インテル® アーキテクチャ対応 インテル® インテグレーテッド・パフォーマンス・プリミティブ（IPP）

リファレンス・マニュアル

第 1 巻 : 信号処理

資料番号 : A24968-4002J

Web : http://www.intel.co.jp/jp/developer/（日本語）
: http://developer.intel.com（英語）

バージョン	バージョン情報	日付
-1001	初版	2000 年 9 月
-1002	インテル® IPP 1.0 最終リリースについて説明。関数 NormDiff、AutoCorr、ZeroMean を追加。導関数の項を改訂。	2001 年 2 月
-1101	インテル® IPP 1.1 ベータ・リリースについて説明。	2001 年 4 月
-2001	インテル® IPP 2.0 ベータ・リリースについて説明。GAC（General Audio Coding）、MP3、超越ベクトル関数を追加。音声認識 API を改訂。	2001 年 8 月
-2002	インテル® IPP 2.0 ゴールド・リリースについて説明。新しいインテル IPP 共通関数を追加。算術関数、ベクトル初期化関数、統計関数、フィルタリング関数を拡張。	2001 年 11 月
-3001	インテル® IPP 3.0 プレベータについて説明。新しい音声コーデック関数を追加。変換関数および窓（Window）関数を拡張。音声認識 API を更新。	2002 年 4 月
-3002	インテル® IPP 3.0 ベータについて説明。	2002 年 6 月
-3003	インテル® IPP 3.0 ベータ・アップデートについて説明。	2002 年 9 月
-3004	インテル® IPP 3.0 ゴールド・リリースについて説明。	2002 年 11 月
-4001	インテル® IPP 4.0 ベータについて説明。クロス・アーキテクチャ開発のための新しい関数を追加。	2003 年 5 月
-4002	インテル® IPP 4.0 ゴールド・リリースについて説明。	2003 年 10 月

【輸出規制に関する告知と注意事項】

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

【資料内容に関する注意事項】

・ 本ドキュメントの内容を予告なしに変更することがあります。

・ インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。

・ インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。

・ 本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。

インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。

・ いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複写することは禁じられています。

インテル、Intel ロゴ、Intel XScale、Itanium、MMX、Pentium は、アメリカ合衆国およびその他の国における Intel Corporation またはその子会社の商標または登録商標です。

* その他の社名、製品名などは、一般に各社の商標または登録商標です。

© 2000-2004, Intel Corporation.

目 次

第 1 章　概要

ソフトウェアの概要 1-1
　ハードウェアとソフトウェアの要件 1-2
　サポートされるプラットフォーム 1-2
　クロスアーキテクチャの統一 1-2
　　クロス・アーキテクチャの概要 1-2
　　従来のコードへの対応 1-3
　技術サポート 1-4
本書について 1-4
　本書の構成 1-5
　関数の説明 1-6
　本書の対象読者 1-6
　オンライン版 1-6
　関連資料 1-6
表記の規則 1-7
　字体の規則 1-7
　信号名の表記規則 1-7
　命名規則 1-7

第 2 章　インテル ® インテグレーテッド・パフォーマンス・プリミティブの概念

基本的な機能 2-1
関数の命名 2-2
　データ領域 2-2
　名前 2-2
　データ・タイプ 2-3
　ディスクリプタ 2-5
　引数 2-6
構造体と列挙子 2-7
　ライブラリ・バージョン構造体 2-7
　複素数データ構造体 2-7
　関数コンテキスト構造体 2-7
　列挙子 2-8
データ範囲 2-9
データ・アライメント 2-10
整数のスケーリング 2-10
エラー・レポート 2-11
コード例 2-17

第 3 章　サポート関数

バージョン情報関数 3-2
GetLibVersion 3-2
メモリ割り当て関数 3-3
ippsMalloc 3-4
ippsFree 3-5
共通の関数 3-6
CoreGetStatusString 3-6
CoreGetCpuType 3-7
CoreGetCpuClocks 3-8
CpuFreqMhz の取得 3-8
CoreSetFlushToZero 3-9
CoreSetDenormAreZeros 3-10
AlignPtr 3-11
ippMalloc 3-11
ippFree 3-12
マージされたライブラリのディスパッチを制御する関数 3-12
StaticInit 3-13
StaticFree 3-14
StaticInitBest 3-14
StaticInitCpu 3-15
バージョン 2.0 との互換性 3-16

第 4 章　ベクトル初期化関数

ベクトル初期化関数 4-2
Copy 4-2
Move 4-4
Set 4-5
Zero 4-6
サンプル生成関数 4-7
トーン生成関数 4-7
ToneInitAllocQ15 4-8
ToneFree 4-9
ToneGetStateSizeQ15 4-9
ToneInitQ15 4-10
ToneQ15 4-11
Tone_Direct 4-11
ToneQ15_Direct 4-13
トライアングル生成関数 4-14
TriangleInitAllocQ15 4-15
TriangleFree 4-17
TriangleGetStateSizeQ15 4-17
TriangleInitQ15 4-18
TriangleQ15 4-19
Triangle_Direct 4-20
TriangleQ15_Direct 4-21
一様分布関数 4-23

intel®

RandUniformInitAlloc 4-23
RandUniformFree 4-24
RandUniformInit 4-24
RandUniformGetSize 4-25
RandUnifrom 4-26
RandUniform_Direct........ 4-27
ガウス分布関数 4-28
RandGaussInitAlloc 4-28
RandGaussFree 4-29
RandGaussGetSize 4-29
RandGaussInit 4-30
RandGauss 4-31
RandGauss_Direct........ 4-32
特殊ベクトル関数 4-33
VectorJaehne 4-33
VectorRamp 4-34

第 5 章　基本的なベクトル関数

論理関数とシフト関数 5-4
AndC 5-4
And........ 5-5
OrC 5-6
Or 5-7
XorC 5-8
Xor........ 5-9
Not........ 5-10
LShiftC 5-11
RShiftC........ 5-12
算術関数 5-14
AddC 5-14
Add........ 5-16
AddProduct 5-18
MulC........ 5-19
Mul 5-21
SubC 5-23
SubCRev 5-25
Sub........ 5-26
DivC 5-28
DivCRev 5-30
Div 5-31
Abs........ 5-33
Sqr........ 5-35
Sqrt 5-37
Cubrt 5-39
Exp........ 5-40
Ln 5-42
10Log10 5-44
SumLn 5-45

Arctan 5-46
Normalize 5-47
変換関数 5-48
SortAscend,
SortDescend 5-49
SwapBytes 5-50
Convert 5-51
Join 5-53
Conj 5-54
ConjFlip 5-55
Magnitude 5-56
MagSquared 5-58
Phase 5-59
PowerSpectr 5-60
Real 5-61
Imag 5-62
RealToCplx 5-63
CplxToReal 5-64
Threshold 5-64
Threshold_LT, Threshold_GT 5-68
Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal 5-71
Threshold_LTInv 5-75
CartToPolar 5-77
PolarToCart 5-79
MaxOrder 5-80
Preemphasize 5-80
Flip 5-81
FindNearestOne 5-82
FindNearest 5-83
ビタビ・デコーダ関数 5-84
GetVarPointDV 5-84
CalcStatesDV 5-85
BuildSymblTableDV4D 5-86
UpdatePathMetricsDV 5-86
窓（Window）関数 5-87
窓関数の概要 5-87
WinBartlett 5-88
WinBlackman 5-90
WinHamming 5-94
WinHann 5-96
WinKaiser 5-97
統計関数 5-100
Sum 5-100
Max 5-101
MaxIndx 5-102
Min 5-103
MinIndx 5-104
MinMax 5-105

intel®

MinMaxIndx 5-106
Mean 5-107
StdDev 5-109
Norm 5-110
NormDiff 5-112
DotProd 5-114
MaxEvery, MinEvery 5-117
サンプリング関数 5-118
SampleUp 5-118
SampleDown 5-120

第 6 章　フィルタリング関数

たたみ込み関数と相関関数 6-1
Conv 6-2
ConvCyclic 6-3
AutoCorr 6-4
CrossCorr 6-6
UpdateLinear 6-8
UpdatePower 6-9
フィルタリング関数 6-10
FIR フィルタ関数 6-12
FIRInitAlloc, FIRMRInitAlloc 6-13
FIRFree 6-17
FIRInit, FIRMRInit 6-18
FIRGetStateSize, FIRMRGetStateSize 6-23
FIRGetTaps, FIRSetTaps 6-25
FIRGetDlyLine, FIRSetDlyLine 6-28
FIROne 6-30
FIR 6-32
FIROne_Direct 6-37
FIR_Direct 6-40
FIRMR_Direct 6-44
FIR フィルタ係数生成関数 6-48
FIRGenLowpass 6-48
FIRGenHighpass 6-50
FIRGenBandpass 6-51
FIRGenBandstop 6-52
シングルレート FIR LMS フィルタ関数 6-53
FIRLMSInitAlloc 6-54
FIRLMSFree 6-55
FIRLMSGetTaps 6-56
FIRLMSGetDlyLine, FIRLMSSetDlyLine 6-57
FIRLMS 6-58
FIRLMSOne_Direct 6-61
マルチレート FIR LMS フィルタ関数 6-63
FIRLMSMRInitAlloc 6-64
FIRLMSMRFree 6-65
FIRLMSMRSetMu 6-66

FIRLMSMRUpdateTaps 6-67
FIRLMSMRGetTaps, FIRLMSMRSetTaps 6-68
FIRLMSMRGetTapsPointer 6-69
FIRLMSMRGetDlyLine, FIRLMSMRSetDlyLine 6-70
FIRLMSMRGetDlyVal 6-71
FIRLMSMRPutVal 6-72
FIRLMSMROne 6-72
FIRLMSMROneVal 6-73
IIR フィルタ関数 6-74
IIRInitAlloc, IIRInitAlloc_BiQuad 6-76
IIRFree 6-80
IIRInit, IIRInit_BiQuad 6-81
IIRGetStateSize, IIRGetStateSize_BiQuad 6-84
IIRSetTaps 6-86
IIRGetDlyLine, IIRSetDlyLine 6-88
IIROne 6-90
IIR 6-92
IIROne_Direct, IIROne_BiQuadDirect 6-97
IIR_Direct, IIR_BiQuadDirect 6-98
メディアン・フィルタ関数 6-100
FilterMedian 6-100

第 7 章　変換関数

フーリエ変換関数 7-3
変換サポート関数 7-3
flag 引数と hint 引数 7-3
Pack 形式 7-4
Perm 形式 7-4
CCS 形式 7-4
パックド・データのアンパック 7-5
ConjPerm 7-5
ConjPack 7-7
ConjCcs 7-9
パックド・データの乗算 7-11
MulPack, MulPerm 7-12
MulPackConj 7-14
高速フーリエ変換関数 7-15
FFTInitAlloc_C, FFT InitAlloc_R 7-16
FFTFree_C, FFTFree_R 7-17
FFT GetBufSize_C, FFTGetBufSize_R 7-18
FFTFwd_CToC, FFTInv_CToC 7-19
FFTFwd_RToPerm, FFTInv_PermToR,
FFTFwd_RToPack, FFTInv_PackToR,
FFTFwd_RToCCS, FFTInv_CCS ToR 7-22
離散フーリエ変換関数 7-25
DFTInitAlloc_C, DFTInitAlloc_R 7-26
DFTFree_C, DFTFree_R 7-27
DFTGetBufSize_C, DFTGetBufSize_R 7-28

intel®

DFTFwd_CToC, DFTInv_CToC 7-29
DFTFwd_RToPerm, DFTInv_PermToR,
DFTFwd_RToPack, DFTInv_PackToR,
DFTFwd_RToCCS, DFTInv_CCS ToR 7-32
DFTOutOrdInitAlloc_C 7-35
DFTOutOrdFree_C 7-36
DFTOutOrdGetBufSize_C 7-37
DFTOutOrdFwd_CToC, DFTOutOrdInv_CToC 7-37
特定周波数用 DFT（Goertzel）関数 7-39
Goertz 7-39
GoertzTwo 7-41
離散コサイン変換関数 7-42
DCTFwdInitAlloc, DCTInvInitAlloc 7-43
DCTFwdFree, DCTInvFree 7-44
DCTFwdGetBufSize, DCTInvGetBufSize 7-45
DCTFwd, DCTInv 7-46
ヒルベルト変換関数 7-48
HilbertInitAlloc 7-49
HilbertFree 7-49
Hilbert 7-50
ウェーブレット変換関数 7-52
固定フィルタバンクに対する変換 7-55
WTHaarFwd, WTHaarInv 7-55
ユーザ・フィルタバンクに対する変換 7-60
WTFwdInitAlloc, WTInvInitAlloc 7-60
WTFwdFree, WTInvFree 7-62
WTFwd 7-63
WTFwdSetDlyLine, WTFwdGetDlyLine 7-67
WTInv 7-69
WTInvSetDlyLine, WTInvGetDlyLine 7-72

第 8 章　音声認識関数

基本算術 8-6
AddAllRowSum 8-7
SumColumn 8-8
SumRow 8-9
SubRow 8-11
CopyColumn_Indirect 8-12
BlockDMatrixInitAlloc 8-13
BlockDMatrixFree 8-14
NthMaxElement 8-15
VecMatMul 8-16
MatVecMul 8-17
特徴処理 8-18
ZeroMean 8-18
CompensateOffset 8-19
SignChangeRate 8-21
LinearPrediction 8-22

Durbin 8-23
Schur 8-25
LPToSpectrum 8-26
LPToCepstrum 8-27
CepstrumToLP 8-28
LPToReflection 8-29
ReflectionToLP 8-30
ReflectionToAR 8-31
ReflectionToTilt 8-32
PitchmarkToF0 8-34
UnitCurve 8-35
LPToLSP 8-36
LSPToLP 8-38
MelToLinear 8-39
LinearToMel 8-40
CopyWithPadding 8-41
MelFBankGetSize 8-42
MelFBankInit 8-43
MelFBankInitAlloc 8-44
MelLinFBankInitAlloc 8-47
EmptyFBankInitAlloc 8-49
FBankFree 8-50
FBankGetCenters 8-50
FBankSetCenters 8-51
FBankGetCoeffs 8-52
FBankSetCoeffs 8-53
EvalFBank 8-54
DCTLifterGetSize_MulC0 8-55
DCTLifterInit_MulC0 8-56
DCTLifterInitAlloc 8-57
DCTLifterFree 8-58
DCTLifter 8-59
NormEnergy 8-62
SumMeanVar 8-63
NewVar 8-65
RecSqrt 8-66
AccCovarianceMatrix 8-67
導関数 8-68
CopyColumn 8-69
EvalDelta 8-70
Delta 8-72
DeltaDelta 8-76
ピッチ超解像度 8-80
CrossCorrCoeffDecim 8-81
CrossCorrCoeff 8-82
CrossCorrCoeffInterpolation 8-83
モデル評価 8-84
AddNRows 8-85

intel®

ScaleLM 8-86
LogAdd 8-87
LogSub 8-88
LogSum 8-89
MahDistSingle 8-90
MahDist 8-91
MahDistMultiMix 8-93
LogGaussSingle 8-94
LogGauss 8-98
LogGaussMultiMix 8-101
LogGaussMax 8-103
LogGaussMaxMultiMix 8-107
LogGaussAdd 8-109
LogGaussAddMultiMix 8-113
LogGaussMixture 8-115
LogGaussMixtureSelect 8-118
BuildSignTable 8-121
FillShortlist_Row 8-123
FillShortlist_Column 8-125
DTW 8-128
モデル推定 8-130
MeanColumn 8-130
VarColumn 8-131
MeanVarColumn 8-133
WeightedMeanColumn 8-134
WeightedVarColumn 8-135
WeightedMeanVarColumn 8-137
NormalizeColumn 8-139
NormalizeInRange 8-141
MeanVarAcc 8-143
GaussianDist 8-144
GaussianSplit 8-145
GaussianMerge 8-146
Entropy 8-147
SinC 8-148
ExpNegSqr 8-149
BhatDist 8-150
UpdateMean 8-152
UpdateVar 8-153
UpdateWeight 8-154
UpdateGConst 8-155
OutProbPreCalc 8-156
DcsClustLAccumulate 8-157
DcsClustLCompute 8-159
モデル適応 8-160
AddMulColumn 8-160
AddMulRow 8-161
QRTransColumn 8-162

DotProdColumn 8-163
MulColumn 8-164
SumColumnAbs 8-165
SumColumnSqr 8-166
SumRowAbs 8-167
SumRowSqr 8-167
SVD 8-168
WeightedSum 8-170
ベクトル量子化 8-171
FormVector 8-171
CdbkGetSize 8-174
CdbkInit 8-175
CdbkInitAlloc 8-177
CdbkFree 8-178
GetCdbkSize 8-179
GetCodebook 8-180
VQ 8-181
VQSingle_Sort, VQSingle_Thresh 8-182
SplitVQ 8-183
FormVectorVQ 8-185
ポリフェーズ・リサンプリング 8-186
ResamplePolyphaseInitAlloc 8-187
ResamplePolyphaseFree 8-189
ResamplePolyphase 8-189
アドバンスト・オーロラ関数 8-192
SmoothedPowerSpectrum_Aurora 8-192
NoiseSpectrumUpdate_Aurora 8-193
WienerFilterDesign_Aurora 8-194
MelFBankInitAlloc_Aurora 8-196
TabsCalculation_Aurora 8-196
ResidualFilter_Aurora 8-197
WaveProcessing_Aurora 8-198
LowHighFilter_Aurora 8-200
HighBandCoding_Aurora 8-201
BlindEqualization_Aurora 8-202
DeltaDelta_Aurora 8-203
VADGetBufSize_Aurora 8-208
VADInit_Aurora 8-208
VADDecision_Aurora 8-209
VADFlush_Aurora 8-209
Ephraim-Malah ノイズ・サプレッサ 8-210
ノイズ・サプレッサ・アーキテクチャ 8-210
Ephraim-Malah ノイズ・サプレッサの詳細 8-211
アルゴリズムのステップ 8-211
データ構造体 8-214
フィルタ更新プリミティブ 8-215
FilterUpdateEMNS 8-215
FilterUpdateWiener 8-217

intel®

ノイズ・フロア推定プリミティブ 8-218
GetSizeMCRA 8-218
InitMCRA 8-219
InitAllocMCRA 8-220
UpdateNoisePSDMCRA 8-221
音響エコー・キャンセラ 8-221
音響エコー・キャンセラ・アーキテクチャ 8-222
周波数領域ブロック NLMS 適応フィルタ 8-223
アルゴリズムのステップ 8-223
計算の複雑さ 8-225
AEC コントローラ 8-226
アルゴリズムの説明 8-227
データ構造体 8-231
フィルタ・プリミティブ 8-232
FilterAECNLMS 8-232
フィルタ更新プリミティブ 8-233
CoefUpdateAECNLMS 8-233
ステップ・サイズ更新プリミティブ 8-235
StepSizeUpdateAECNLMS 8-235
AEC コントローラ・プリミティブ 8-236
ControllerGetSizeAEC 8-236
ControllerInitAEC 8-237
ControllerUpdateAEC 8-238
音声アクティビティ検出（VAD） 8-239
音声アクティビティ検出アーキテクチャ 8-240
音声アクティビティ検出プリミティブ 8-240
FindPeaks 8-240
PeriodicityLSPE 8-241
Periodicity 8-242
バージョン 1.1 との互換性 8-244
バージョン 2.0 との互換性 8-245

第 9 章　音声符号化関数

丸めモード 9-1
表記の規則 9-2
定義 9-3
データ構造体 9-3
共通の関数 9-4
ConvPartial 9-5
Mul_NR 9-6
MulC_NR 9-7
MulPowerC_NR 9-8
AutoScale 9-9
DotProdAutoScale 9-10
InvSqrt 9-11
AutoCorr 9-12
AutoCorrLagMax 9-13
AutoCorr_NormE 9-14

CrossCorr 9-15
CrossCorrLagMax 9-16
SynthesisFilter 9-17
G.729 に関連する関数 9-19
基本関数 9-20
DotProd_G729 9-20
Interpolate_G729 9-21
線形予測分析関数 9-23
AutoCorr_G729 9-23
LevinsonDurbin_G729 9-24
LPCToLSP_G729 9-26
LSFToLSP_G729 9-27
LSFQuant_G729 9-28
LSFDecode_G729 9-29
LSFDecodeErased_G729 9-30
LSPToLPC_G729 9-31
LSPQuant_G729 9-32
LSPToLSF_G729 9-36
LagWindow_G729 9-37
コードブック検索関数 9-38
OpenLoopPitchSearch_G729 9-38
AdaptiveCodebookSearch_G729 9-41
DecodeAdaptiveVector_G729 9-44
FixedCodebookSearch_G729 9-45
ToeplizMatrix_G729 9-49
コードブック・ゲイン関数 9-50
DecodeGain_G729 9-50
GainControl_G729 9-52
GainQuant_G729 9-54
AdaptiveCodebookContribution_G729 9-56
AdaptiveCodebookGain_G729 9-57
フィルタ関数 9-59
ResidualFilter_G729 9-60
SynthesisFilter_G729 9-61
LongTermPostFilter_G729 9-63
ShortTermPostFilter_G729 9-66
TiltCompensation_G729 9-68
HarmonicFilter 9-70
HighPassFilterSize_G729 9-72
HighPassFilterInit_G729 9-72
HighPassFilter_G729 9-73
IIR16s_G729 9-74
PhaseDispersionGetStateSize_G729D 9-75
PhaseDispersionInit_G729D 9-76
PhaseDispersionUpdate_G729D 9-76
PhaseDispersion_G729D 9-77
Preemphasize_G729A 9-78
WinHybridGetStateSize_G729E 9-79

WinHybridInit_G729E 9-79
WinHybrid_G729E 9-80
RandomNoiseExcitation_G729B 9-81
G.723.1 に関連する関数 9-82
線形予測分析関数 9-83
AutoCorr_G723 9-83
AutoCorr_NormE_G723 9-85
LevinsonDurbin_G723 9-86
LPCToLSF_G723 9-87
LSFToLPC_G723 9-88
LSFDecode_G723 9-89
LSFQuant_G723 9-90
コードブック検索関数 9-91
OpenLoopPitchSearch_G723 9-91
ACELPFixedCodebookSearch_G723 9-93
AdaptiveCodebookSearch_G723 9-94
MPMLQFixedCodebookSearch_G723 9-96
ToeplizMatrix_G723 9-97
ゲイン量子化 9-98
GainQuant_G723 9-99
GainControl_G723 9-100
フィルタ関数 9-101
HighPassFilter_G723 9-102
IIR16s_G723 9-102
SynthesisFilter_G723 9-104
TiltCompensation_G723 9-105
HarmonicSearch_G723 9-106
HarmonicNoiseSubtract_G723 9-107
DecodeAdaptiveVector_G723 9-108
PitchPostFilter_G723 9-110
GSM-AMR に関連する関数 9-112
基本関数 9-114
Interpolate_GSMAMR 9-114
FFTFwd_RToPerm_GSMAMR 9-114
LP 分析と量子化プリミティブ 9-115
AutoCorr_GSMAMR 9-116
LevinsonDurbin_GSMAMR 9-118
LPCToLSP_GSMAMR 9-119
LSPToLPC_GSMAMR 9-121
LSFToLSP_GSMAMR 9-122
LSPQuant_GSMAMR 9-123
QuantLSPDecode_GSMAMR 9-125
アダプティブ・コードブック・プリミティブ 9-127
オープン・ループ・ピッチの検索（OLP） 9-127
OpenLoopPitchSearchNonDTX_GSMAMR 9-130
OpenLoopPitchSearchDTXVAD1_GSMAMR 9-132
OpenLoopPitchSearchDTXVAD2_GSMAMR 9-135
ImpulseResponseTarget_GSMAMR 9-138

AdaptiveCodebookSearch_GSMAMR 9-139
AdaptiveCodebookDecode_GSMAMR 9-143
AdaptiveCodebookGain_GSMAMR 9-145
固定コードブック検索 9-147
AlgebraicCodebookSearch_GSMAMR 9-147
FixedCodebookDecode_GSMAMR 9-150
断片的な送信（DTX） 9-151
Preemphasize_GSMAMR 9-151
VAD1_GSMAMR 9-152
VAD2_GSMAMR 9-154
EncDTXSID_GSMAMR 9-156
EncDTXHandler_GSMAMR 9-159
EncDTXBuffer_GSMAMR, DecDTXBuffer_GSMAMR 9-160
後処理 9-161
PostFilter_GSMAMR 9-161
GSM フル・レートに関連する関数 9-164
RPEQuantDecode_GSMFR 9-165
Deemphasize_GSMFR 9-165
ShortTermAnalysisFilter_GSMFR 9-166
ShortTermSynthesisFilter_GSMFR 9-168
HighPassFilter_GSMFR 9-169
Schur_GSMFR 9-170
WeightingFilter_GSMFR 9-170
Preemphasize_GSMFR 9-171
G.722.1 に関連する関数 9-172
DCTFwd_G722, DCTInv_G722 9-173
DecomposeMLTToDCT 9-174
DecomposeDCTToMLT 9-175
HuffmanEncode_G722 9-176
G.726 に関連する関数 9-177
EncodeGetStateSize_G726 9-177
EncodeInit_G726 9-178
Encode_G726 9-179
DecodeGetStateSize_G726 9-180
DecodeInit_G726 9-180
Decode_G726 9-181
G.728 に関連する関数 9-182
IIRGetStateSize_G728 9-183
IIR_Init_G728 9-183
IIR_G728 9-184
SynthesisFilterGetStateSize_G728 9-185
SynthesisFilterInit_G728 9-185
SyntesisFilter_G728 9-186
CombinedFilterGetStateSize_G728 9-187
CombinedFilterInit_G728 9-188
CombinedFilter_G728 9-188
PostFilterGetStateSize_G728 9-190

PostFilterInit_G728 ... 9-190
PostFilter_G728 ... 9-191
WinHybridGetStateSize_G728 ... 9-192
WinHybridInit_G728 ... 9-193
WinHybrid_G728 ... 9-194
LevinsonDurbin_G728 ... 9-196
CodebookSearch_G728 ... 9-197
ImpulseResponseEnergy_G728 ... 9-198

第 10 章　オーディオ符号化関数

インターリーブ形式から複数行形式への変換関数 ... 10-5
Interleave ... 10-5
Deinterleave ... 10-6
スペクトル・データ・プレ量子化関数 ... 10-7
Pow34 ... 10-7
Pow43 ... 10-9
スケール係数計算関数 ... 10-10
CalcSF ... 10-10
仮数変換およびスケーリング関数 ... 10-11
ApplySF_I ... 10-11
MakeFloat ... 10-12
変形離散コサイン変換関数 ... 10-13
MDCTFwdInitAlloc, MDCTInvInitAlloc ... 10-13
MDCTFwdFree, MDCTInvFree ... 10-14
MDCTFwdGetBufSize, MDCTInvGetBufSize ... 10-15
MDCTFwd, MDCTInv ... 10-16
ブロック・フィルタリング関数 ... 10-17
FIRBlockInitAlloc ... 10-18
FIRBlockFree ... 10-19
FIRBlockOne ... 10-19
周波数領域予測関数 ... 10-21
FDPInitAlloc ... 10-22
FDPFree ... 10-23
ResetFDP ... 10-23
ResetFDP_SFB ... 10-24
ResetFDPGroup ... 10-25
FDPFwd ... 10-25
FDPInv ... 10-26
ハフマン・アルゴリズム関数 ... 10-27
GetSizeHDT ... 10-28
BuildHDT ... 10-29
DecodeVLC ... 10-30
GetSizeHET ... 10-32
BuildHET ... 10-33
HuffmanCountBits ... 10-34
EncodeVLC ... 10-35
GetSizeHET_VLC ... 10-36
BuildHET_VLC ... 10-37

CountBits 10-38
EncodeBlock 10-39
ベクトル量子化関数 10-41
CdbkInitAlloc 10-41
CdbkFree 10-42
PreSelect_VQ 10-42
MainSelect_VQ 10-44
IndexSelect_VQ 10-46
VectorReconstruction_VQ 10-48
圧伸関数 10-49
MuLawToLin 10-50
LinToMuLaw 10-51
ALawToLin 10-52
LinToALaw 10-53
MuLawToALaw 10-54
ALawToMuLaw 10-55
MP3 オーディオ符号化関数 10-56
マクロおよび定数 10-57
データ構造体 10-57
フレーム・ヘッダ 10-57
サイド情報 10-58
MP3 心理音響モデル 2 分析 10-59
心理音響モデル 2 ステート 10-59
MP3 ビット貯蓄 10-60
MP3 コーデック列挙型 10-61
MP3 オーディオ・エンコーダ 10-61
AnalysisPQMF_MP3 10-62
MDCTFwd_MP3 10-64
PsychoacousticModelTwo_MP3 10-66
JointStereoEncode_MP3 10-71
Quantize_MP3 10-75
PackScalefactors_MP3 10-81
HuffmanEncode_MP3 10-84
PackFrameHeader_MP3 10-87
PackSideInfo_MP3 10-88
BitReservoirInit_MP3 10-90
MP3 オーディオ・デコーダ 10-91
UnpackFrameHeader_MP3 10-93
UnpackSideInfo_MP3 10-94
UnpackScaleFactors_MP3 10-95
HuffmanDecode_MP3
HuffmanDecodeSfb_MP3
HuffmanDecodeSfbMbp_MP3 10-98
ReQuantize_MP3
ReQuantizeSfb_MP3 10-100
MDCTInv_MP3 10-102
SynthPQMF_MP3 10-104
アドバンスト・オーディオ符号化関数 10-106

グローバル・マクロ . . . 10-108
データ・タイプと構造体 . . . 10-108
ADIF ヘッダ . . . 10-108
ADTS フレーム・ヘッダ . . . 10-109
単一チャネル・サイド情報 . . . 10-110
AAC スケーラブル・メイン要素ヘッダ . . . 10-110
AAC スケーラブル拡張要素ヘッダ . . . 10-111
1 レイヤの TNS 構造体 . . . 10-111
LTP 構造体 . . . 10-112
チャネルのペア要素 . . . 10-112
チャネル情報 . . . 10-112
MPEG-2 AAC プリミティブ . . . 10-114
UnpackADIFHeader_AAC ... 10-114
UnpackADTSFrameHeader_AAC ... 10-115
DecodePrgCfgElt_AAC ... 10-116
DecodeChanPairElt_AAC ... 10-117
NoiselessDecoder_LC_AAC ... 10-119
DecodeDatStrElt_AAC ... 10-122
DecodeFillElt_AAC ... 10-123
QuantInv_AAC ... 10-125
DecodeMsStereo_AAC ... 10-127
DecodeIsStereo_AAC ... 10-129
DeinterleaveSpectrum_AAC ... 10-131
DecodeTNS_AAC ... 10-133
MDCTInv_AAC ... 10-137
MPEG-4 AAC プリミティブ . . . 10-140
DecodeMainHeader_AAC ... 10-140
DecodeExtensionHeader_AAC ... 10-141
DecodePNS_AAC ... 10-142
LongTermReconstruct_AAC ... 10-143
MDCTFwd_AAC ... 10-144
EncodeTNS_AAC ... 10-145
LongTermPredict_AAC ... 10-147
NoiseLessDecode_AAC ... 10-148
LtpUpdate_AAC ... 10-150

第 11 章　文字列関数

Find,
FindRev ... 11-2
FindC
FindRevC ... 11-3
FindCAny,
FindRevCAny ... 11-4
Insert ... 11-5
Remove ... 11-6
Compare ... 11-7
CompareIgnoreCase,
CompareIgnoreCaseLatin ... 11-8

Equal 11-9
TrimC 11-10
TrimCAny
TrimStartCAny
TrimEndCAny 11-11
ReplaceC 11-12
Uppercase,
UppercaseLatin 11-13
Lowercase,
LowercaseLatin 11-14
Hash 11-15
Concat 11-16
ConcatC 11-17
SplitC 11-18

第 12 章　固定精度算術関数

累乗関数と根関数 12-3
Inv 12-3
Div 12-5
Sqrt 12-7
InvSqrt 12-8
Cbrt 12-10
InvCbrt 12-12
Pow 12-14
Powx 12-16
指数関数と対数関数 12-18
Exp 12-18
Ln 12-20
Log10 12-22
三角関数 12-23
Cos 12-23
Sin 12-25
SinCos 12-27
Tan 12-29
Acos 12-30
Asin 12-32
Atan 12-34
Atan2 12-36
双曲線関数 12-38
Cosh 12-38
Sinh 12-40
Tanh 12-41
Acosh 12-43
Asinh 12-45
Atanh 12-46
特殊関数 12-48
Erf 12-48
Erfc 12-50

付録 A　特殊な事例の処理

付録 B　参考文献

用語集

索引

1 概要

本書では、1 次元信号を操作するインテル® アーキテクチャ用インテル® インテグレーテッド・パフォーマンス・プリミティブ（IPP: Integrated Performance Primitives）の構造体、演算、関数について説明する。本書は、インテル® IPP リファレンス・マニュアルの第 1 巻である。画像および動画処理用のインテル IPP は第 2 巻、小行列演算については第 3 巻、暗号化関数については第 4 巻で説明する。
インテル IPP ソフトウェア・パッケージがサポートしている多くの機能は、インテル・アーキテクチャ、特にインテル® MMX® テクノロジとストリーミング SIMD 拡張命令（SSE）向けに最適化されている。インテル® PCA アプリケーション・プロセッサ向けのインテル IPP の実装に関する詳細は、本章の「クロスアーキテクチャの統一」を参照のこと。

信号処理ソフトウェア用のインテル IPP は、1 次元（1D）のデータ配列に対して演算を実行する、高性能でオーバーヘッドが低い関数の集まりである。

本書では、インテル IPP の概念や信号処理の領域で使用する特定のデータ・タイプ定義や演算モデル、信号処理用インテル IPP ソフトウェアに含まれる各種の関数について詳しく説明する。

本章では、インテル IPP ソフトウェアの概要と本書の構成を説明する。

ソフトウェアの概要

インテル® アーキテクチャ・ソフトウェア用のインテル® IPP では、インテル® MMX® テクノロジやストリーミング SIMD 拡張命令（SSE）の中核をなす SIMD（single-instruction, multiple-data）命令の並列処理を活用できる。MMX や SSE は、大量の計算を必要とする信号処理、イメージ処理、動画処理のアプリケーションの性能を高めるのに役立つテクノロジである。

ハードウェアとソフトウェアの要件

インテル® アーキテクチャ・ソフトウェア用のインテル® IPP は、Microsoft* Windows* 2000、Windows ME、Windows XP、または Linux* オペレーティング・システムをサポートしている、IA-32 プロセッサ・ベースまたはインテル® Itanium® アーキテクチャ・プロセッサ・ベースのパーソナル・コンピュータ上で動作する。インテル IPP は、C や C++ で作成した顧客のアプリケーションやライブラリに組み込める。

サポートされるプラットフォーム

インテル® アーキテクチャ・ソフトウェア用のインテル® IPP は、Windows* や Linux* プラットフォーム上で動作する。本書では、関数や変数の宣言で使用するコードやシンタックスを ANSI C スタイルで記述している。ただし、プロセッサやオペレーティング・システムの種類に応じて、必要とされるインテル IPP のバージョンは多少異なる場合がある。

クロスアーキテクチャの統一

クロス・アーキテクチャの概要

インテル® IPP は、さまざまなインテル® アーキテクチャにおけるアプリケーション開発を支援するように設計されている。インテル IPP はこれまで、インテル® Pentium® プロセッサ、インテル® Xeon™ プロセッサおよびインテル® Itanium® プロセッサ対応の製品と、Intel XScale® テクノロジを利用するインテル® PCA プロセッサ対応の製品の 2 つの別製品として提供されていた。これらのパッケージは、バージョン 4.0 以前では機能とインターフェイスでいくつかの相違点があった。

クロス・アーキテクチャ開発に対する関心の高まりに伴い、インテル IPP 開発チームは複数にわたるインテル® プラットフォーム上でのアプリケーション開発が容易になるようにこれらの相違点の解消に取り組んだ。この結果、インテル IPP 4.0 では従来 2 つの製品で対応していたアーキテクチャを 1 つの製品で対応するように統一された。この統一には、両方のアーキテクチャで対応するすべての関数のインターフェイスの統一と完全な API の統一が含まれる。

本リリースで、インテル PCA プロセッサで利用可能な関数は、インテル Pentium プロセッサ、インテル Xeon プロセッサおよびインテル Itanium プロセッサでも利用可能になった。これは、関数の実装は各プロセッサのアーキテクチャを考慮して行われるが、API 定義はすべてのプロセッサで共通になったことを意味する。

単一のクロスアーキテクチャ API の提供により、開発者はインテル® プロセッサ・ベースのデスクトップ、サーバ、モバイルおよび携帯端末などのさまざまなプラットフォーム用にソフトウェア・アプリケーションを容易に開発できるようになった。開発者はコードを一度記述するだけで、多くのプロセッサでアプリケーションのパフォーマンスを最適化することが可能である。

バージョン 3.0 からバージョン 4.0 の API の追加および変更に関する詳細は、製品のリリース・ノートを参照のこと。

各インテル IPP インプリメンテーションに含まれている関数を次の表に示す。

表 1-1 インテル® IPP に含まれている関数

関数グループ	インテル® Pentium® 4 プロセッサ	インテル® Itanium® 2 プロセッサ	インテル® PCA プロセッサ
信号処理	○	○	○
画像処理	○	○	○
JPEG	○	○	○
音声認識	○	○	×
音声コーディング	○	○	○
オーディオ・コーデック	○	○	○
ビデオ・コーデック	○	○	○
行列処理	○	○	×
ベクトル演算	○	○	×
コンピュータ・ビジョン	○	○	×
暗号化	○	○	○

従来のコードへの対応

インテル® Pentium® プロセッサ、インテル® Xeon™ プロセッサおよびインテル® Itanium® プロセッサ用の関数とインテル® PCA プロセッサ用の関数の統一により、API にいくつかの変更が行われたが、インテル® IPP 4.0 は以前のバージョンで使用されていた API との完全な互換性も保っている。このため、開発者はインターフェイスを変更することなくアプリケーションの更新が可能である。しかし、今後の互換性のことを考慮して、現行のインターフェイスと新しい API を使用することを推奨する。変更された関数の一覧は、各章の最後に示す。

技術サポート

インテル® IPP には、製品の特徴、ホワイトペーパー、技術記事など、製品に関する総合的な情報が適宜掲載される製品 Web サイトが用意されている。最新情報については、次のサイトを参照のこと。
http://www.intel.co.jp/jp/developer/software/products/ または、
http://developer.intel.com/software/products/（英語）

インテルでは、使い方のヒント、製品に関する確認済みの問題点、製品のエラッタ、ライセンス情報、ユーザ・フォーラムなどの大量のセルフヘルプ情報を蓄積したサポート Web サイトも用意している（http://support.intel.co.jp/jp/support/ または、http://support.intel.com/support/（英語）を参照のこと）。

ユーザ登録を行うと、インテル® プレミア・サポートによる一年間の技術サポートと製品アップデートのサービスが受けられる。インテル・プレミア・サポートは、以下のサービスを提供する、双方向型の問題管理 / コミュニケーション Web サイトである。

- 問題の送信と問題の状態の検討
- 製品アップデートのダウンロード（1 日 24 時間）

ユーザ登録、インテルへの問い合わせ、または製品サポートについては、次のサイトを参照のこと。http://support.intel.co.jp/jp/support/ (英語)

本書について

本書では、インテル® IPP ソフトウェアで使用される信号処理の概念の背景と、インテル® アーキテクチャ信号処理用インテル IPP 関数の詳細について説明する。

注： 本書のインテル IPP という記述は、特に明記しない限り、インテル・アーキテクチャ用インテル・インテグレーテッド・パフォーマンス・プリミティブのことを意味する。
インテル® PCA プロセッサ用のインテル IPP に関する説明は、Intel XScale® マイクロアーキテクチャ用インテル IPP リファレンス・マニュアルを参照のこと。

インテル IPP 関数は、機能別にグループ分けされている。本書では、グループごとに章を変えて、これらの関数を説明する（第 3 章〜第 12 章）。

本書の構成

本書は、次の章で構成されている。

第 1 章	「概要」。インテル® IPP ソフトウェア、本書の構成、本書で使用する表記の規則を説明する。
第 2 章	「インテル・インテグレーテッド・パフォーマンス・プリミティブの概念」。インテル IPP の信号処理の基本概念を説明し、サポートされるデータ形式と演算モードを説明する。
第 3 章	「サポート関数」。`Copy` や `Set` など、信号の操作で使用する関数を説明する。また、データ変換関数やメモリ割り当て関数も説明する。
第 4 章	「ベクトル初期化関数」。初期化関数とサンプル生成関数を説明する。
第 5 章	「基本的なベクトル関数」。ベクトル操作関数を説明する。
第 6 章	「フィルタリング関数」。線形フィルタと非線形フィルタを使用したフィルタリング演算を説明する。
第 7 章	「変換関数」。領域変換関数（フーリエ、ウェーブレット、コサイン）を説明する。
第 8 章	「音声認識関数」。音声認識のアプリケーションで使用する関数を説明する。
第 9 章	「音声符号化関数」。音声コーデック開発のビルディング・ブロックとして使用する関数を説明する。
第 10 章	「オーディオ符号化関数」。複数のコーデックに使用される汎用関数、MPEG-4 オーディオ・コーデック、MP3 エンコーダ、デコーダ専用の多くの関数、さらに携帯機器向けに最適化された MPEG-4 AAC Main Profile デコーダおよび携帯機器向けに最適化された MPEG-1、2 Layer III エンコーダの開発用の関数を説明する。
第 11 章	「文字列関数」。文字列演算を実行する関数を説明する。
第 12 章	「固定精度算術関数」。リアルタイム・アプリケーションにおけるマルチメディア / 信号処理に最適なインテル IPP 固定精度ベクトル数値演算関数を説明する。

また、本書には「付録」、「用語集」、「参考文献」、「索引」も含まれている。

関数の説明

第 3 章〜第 12 章では、各関数を短い名前で（関数名 ipps のプリフィックスと修飾子を省略）示し、関数の目的について簡単な説明をする。それに続いて、関数呼び出しシーケンス、引数の定義、関数の目的の詳しい説明を記載している。関数の説明には、以下の項目がある。

引数	関数のすべての引数について指定する。
説明	関数の定義を示したその関数によって実行される処理について説明する。この項には、コードの例と記述式も含まれる。
アプリケーション・ノート	アプリケーション・プログラマや関数を使用するユーザが知っていなければならない特別な情報がある場合に記述される。
戻り値	関数によって返される値。通常は、関数によって返されるエラー・コードを記述している。
関連項目	関連タスクを実行する関数がある場合は、その名前が記述される。

本書の対象読者

本書は、信号処理用アプリケーション / ライブラリおよびその他の領域におけるアプリケーションの開発者向けに書かれている。本書を理解するには、C 言語の使用経験と、信号処理に関する用語や原理の知識が必要である。

オンライン版

本書は、PDF で提供される。本書をハードコピーで参照するには、Adobe Acrobat* の印刷機能を使用してファイルを印刷のこと（Adobe Acrobat はドキュメントをオンラインで表示するためのツール）。

関連資料

信号処理の概念やアルゴリズムについてさらに詳しく知りたい場合は、「参考文献」に記載の資料を参照のこと。

表記の規則

本書では、次の表記の規則を使用している。

- フォントの規則（テキストとコードの区別）
- 信号名の規則
- さまざまなアイテムの命名規則

字体の規則

本書では次の字体の規則が使用される。

`THIS TYPE STYLE`	インテル® IPP 定数識別子のテキストに使用する（例 :`IPP_MAX_64S`）。
`This type style`	`IppLibraryVersion` のように、構造体の名前には大文字を含む。関数名やコードの例、呼び出しステートメントでも使用される（例 : `void ippsFree()`）。
`This type style`	引数の説明で変数を表す（例 : *`val`*、*`srcLen`*）。

信号名の表記規則

本書では、通常はベクトルまたは配列を使用して、離散 1D 信号を表している。表記 $x(n)$ は概念的な信号を表し、表記 $x[n]$ は実際のベクトルを表す。これらの表記には、値の有限範囲を示す次の注釈が付けられる。

$x[n],\ 0 \le n < len$

通常、ベクトルの要素の数は *len* で示される。ベクトル名には、ベクトル要素を識別するための角括弧を含む。角括弧内には、現行インデックスの *n* が指定される。

例えば、式 $pDst[n] = pSrc[n] + val$ は、ベクトル *pDst* の各要素 $pDst[n]$ が、各 *n*（0 ～ *len*-1）ごとに計算されるのを意味する。特殊なケースについては、その都度説明を行う。

命名規則

インテル® IPP ソフトウェアは、さまざまなアイテムに対して次の命名規則を使用する。

- 不変識別子は大文字で示す（例 : IPP_MIN_64S）。

- すべての信号処理専用の構造体と列挙子には、`Ipps` プレフィックスが付く。これに対し、インテル IPP ソフトウェア全体に共通の構造体と列挙子には、`Ipp` プリフィックスが付く（例 : `IppsROI`、`IppLibraryVersion`）。
- すべての信号処理関数の名前には、`ipps` プリフィックスが追加される。コードの例では、このプリフィックスにより、インテル IPP インターフェイスの関数とアプリケーションの関数とを区別できる。

注 : 本書のコードの例では、関数名に必ず `ipps` プリフィックスを付けている。ただし、通常、本文中で関数グループを指すときは、このプリフィックスを省略している。

- 関数名の新しい部分は、下線を使用せず、それぞれ大文字で始める（例 : `ippsAddAllRowSum`）。

インテル IPP の関数名の構造の詳細は、2-2 ページを参照のこと。

インテル® インテグレーテッド・パフォーマンス・プリミティブの概念 2

本章では、インテル® インテグレーテッド・パフォーマンス・プリミティブ（インテル® IPP）ソフトウェアの目的と構造について説明し、インテル IPP の信号処理部分で使用されるいくつかの基本的な概念について解説する。また、サポートされるデータ形式と動作モード、および関数の命名規則についても説明する。

基本的な機能

インテル® インテグレーテッド・パフォーマンス・プリミティブは、インテル® パフォーマンス・ライブラリ集の他の製品と同様に、特定の領域の演算を行う高性能コードを集めたものである。インテル® IPP は、低レベルのステートレス・インターフェイスを提供する。

インテル IPP は、インテル・パフォーマンス・ライブラリの開発と使用の経験に基づいて開発され、以下の主な特徴を備えている。

- インテル IPP は、信号処理、画像および動画処理、小行列の演算、暗号化アプリケーションなど、さまざまな領域のアプリケーションを作成するための基本的な低レベルの関数を提供する。
- インテル IPP の関数は、異なるアプリケーション領域を参照するプリミティブについても、共通の命名規則やよく似たプロトタイプの合成など、同じインターフェイス規則を適用する。
- インテル IPP 関数は、アプリケーション・プログラムで優れた性能を発揮するのに最適な抽象レベルを使用している。

プログラムのパフォーマンスを向上させるために、インテル IPP 関数は、インテル® アーキテクチャ・プロセッサの利点をすべて利用するように最適化されている。また、ほとんどのインテル IPP 関数は、他のライブラリのように複雑なデータ構造体を使用しないため、全体的な実行オーバーヘッドが小さくなる。

インテル IPP は、クロスプラットフォーム・アプリケーションに最適である。例えば、IA-32 プラットフォーム用に開発された関数を、Itanium® ベースのプラットフォームおよび StrongARM* テクノロジや Intel XScale® テクノロジのシステムに移植できる。プラットフォームの互換性についての詳細は、第 1 章の「クロスアーキテクチャの統一」を参照のこと。また、それぞれのインテル IPP 関数には、ANSI C

で記述したサンプル・コードを用意している。このリファレンス・コードにより、使用されるアルゴリズムが明確に表現され、異なるオペレーティング・システム間の互換性が実現される。

関数の命名

インテル® IPP 関数の命名規則には、すべての領域で同等のものが使用される。信号処理関数には関数名に ipps プリフィックスが付き、画像および動画処理関数には ippi プリフィックスが付き、小行列の演算に使用される関数には ippm プリフィックスが付く。

インテル IPP の関数名は、次の一般形式で表される。

```
ipp<data-domain><name>_<datatype>[_<descriptor>](<arguments>);
```

この形式の要素は、次の節で説明する。

データ領域

data-domain には、関数が属する機能性のサブセットを表す一文字が入る。インテル® IPP の現行バージョンは、次の領域をサポートする。

s	信号処理の場合（予想されるデータ・タイプは 1 次元信号）
i	画像および動画処理の場合（予想されるデータ・タイプは 2 次元イメージ）
m	行列の場合（予想されるデータ・タイプは行列）

例えば、関数名が ipps で始まる場合、その関数は信号処理に使用される関数である。

名前

name は、関数が実際に実行する基本的な操作を表す省略形である（例えば、Set、Copy、など。場合によっては、その後に関数固有の修飾子が続く）。

```
<name> = <operation>[_modifier]
```

この修飾子は、関数の小さな変更またはバリエーションを表すために指定する。例えば、関数 ippsFFTInv_CToC_32fc の修飾子 CToC は、複素数データを使用して逆高速フーリエ変換を行い、複素数から複素数への（CToC）変換を実行する意味である。

データ・タイプ

datatype フィールドは、関数が使用するデータ・タイプを次の形式で示す。

```
<bit depth><bit interpretation> ,
```

ここで、

`bit depth = <1|8|16|32|64>`

また、次の式が使用される。

`bit interpretation = <u|s|f>[c]`

u は「符号なし整数」、*s* は「符号付き整数」、*f* は「浮動小数点」、*c* は「複素数」を表す。

インテル® IPP の現行バージョンでは、信号処理関数に対し、表 2-1 に示された、ソースおよびデスティネーションのデータ・タイプがサポートされる。

注： 関数の引数のリストでは、`Ipp` プリフィックスをデータ・タイプ内に記述する。例えば、8 ビット符号付きデータは、`Ipp8s` タイプのように示される。これらのインテル IPP 固有のデータ・タイプは、個々のライブラリ・ヘッダ・ファイル内で定義される。

表 2-1　信号処理でインテル® IPP がサポートするデータ・タイプ

タイプ	通常のタイプ C	インテル® IPP タイプ
8u	unsigned char	Ipp8u
8s	signed char	Ipp8s
16u	unsigned short	Ipp16u
16s	signed short	Ipp16s
16sc	complex short	Ipp16sc
32u	unsigned int	Ipp32u
32s	signed int	Ipp32s
32f	float	Ipp32f
32fc	complex float	Ipp32fc
64s	__int64 （Windows の場合）または long long （Linux の場合）	Ipp64s
64f	double	Ipp64f
64fc	complex double	Ipp64fc

単一データ・タイプに関する演算の関数の場合、*datatype* フィールドには上記の値の 1 つだけしか入らない。

異なるデータ・タイプを持つソース信号とデスティネーション信号を関数で演算する場合、それぞれのデータ・タイプ識別子は、ソース、デスティネーションの順で関数名内に記述する。

```
<datatype> = <src1Depth>[src2Depth][dstDepth]
```

例えば、関数 ippsDotProd_16s16sc_Sfs は、16 ビット short ソース・ベクトルと 16 ビット complex short ソース・ベクトルの内積を計算し、その結果を 16 ビット complex short デスティネーション・ベクトルに格納する。関数名に *dstDepth* 修飾子が記述されていないのは、第 2 オペランドと結果が同じタイプだからである。結果はスケーリングおよび飽和される。

インテル IPP で直接サポートされていない 24u、24s、16f データ・タイプは、ライブラリ関数で処理できるデータ・タイプに変換して使用することができる。

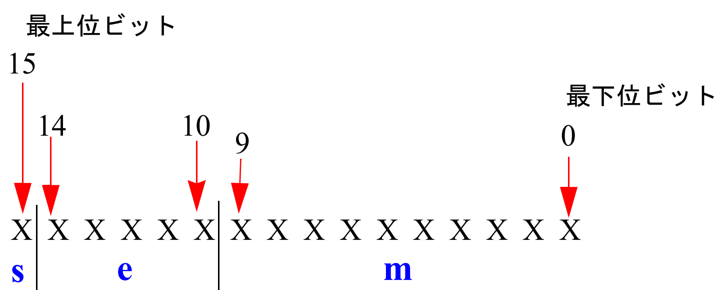
符号なし 24u データの場合、各ベクトル要素は Ipp8u データ・タイプの連続した 3 バイトで構成される。下位バイトが最下位アドレスから順番に格納されている場合、バイト・オーダーはリトルエンディアンである。これらのデータは、インテル IPP 関数 ippsConvert を適切に使用することで、32u データ・タイプまたは 32f データ・タイプと変換することができる。

intel®

符号付き 24s データの場合、各ベクトル要素は Ipp8u データ・タイプの連続した 3 バイトで構成される。下位バイトが最下位アドレスから順番に格納されている場合、バイト・オーダーはリトルエンディアンである。符号は、最上位バイトの最上位ビットで表現される。これらのデータは、インテル IPP 関数 ippsConvert を適切に使用することで、32s データ・タイプまたは 32f データ・タイプと変換することができる。

16f 形式の場合、16 ビットの浮動小数点データ（half タイプ）は、およそ $6.1e^{-5}$ から $6.5e^{4}$ の範囲で、相対誤差が $9.8e^{-4}$ の正および負の値で表現できる。$6.1e^{-5}$ よりも小さい値は、絶対誤差 $6.0e^{-8}$ で表現できる。-2048 から +2048 までの間のすべての整数は正確に表現できる。

次の図は半分の数のビット・レイアウトを示している。



s は符号ビット、**e** は指数、**m** は仮数を示す。

これらのデータは、インテル IPP 関数 ippsConvert を適切に使用することで、16s データ・タイプまたは 32f データ・タイプと変換することができる。

ディスクリプタ

descriptor フィールドには、演算に関連するデータをさらに記述する。ディスクリプタには、暗黙パラメータや追加の必須パラメータを指定できる。関数内のコード分岐数を最小限に抑え、不要な実行オーバーヘッドを減らすため、ほとんどの汎用関数は個々のプリミティブ関数に分割している。そのパラメータのいくつかは、ディスクリプタとしてプリミティブ関数名に指定される。

ただし、関数によっては、内部演算を決定するパラメータを持てるものもある（例 : ippsThreshold）。

信号処理関数では、次のディスクリプタが使用される。

I	演算をインプレースで行う （デフォルトは非インプレース）。

`Sfs`	飽和および固定スケーリング・モード（デフォルトでは飽和処理され、スケーリングは行われない）。
`Dx`	信号は x 次元信号である　（デフォルトは 1 次元）。
`L`	各行につきひとつのポインタを使用する　（2 次元の場合）。

関数名ではディスクリプタの短縮形を常にアルファベット順に記述する。

すべての関数が上記の 2 つの短縮形を持っているわけではない。例えば、copy 演算ではインプレース・モードは意味をなさない。

引数

arguments フィールドには関数の引数を指定する。

引数の順序は、次のとおりである。

1. すべてのソース・オペランド。ベクトルの後に定数が続く。
2. すべてのデスティネーション・オペランド。ベクトルの後に定数が続く。
3. その他、演算固有の引数。

引数名には、次の規則がある。

- ポインタとして定義されたすべての引数は *p* で始まる（例えば、*pPhase*、*pSrc*、*pSeed*）。値として定義された引数は小文字で始まる（例えば、*val*、*src*、*srcLen*）。
- 引数名の新しい部分は、下線を挿入せずに大文字で始まる（例えば、*pSrc*、*lenSrc*、*pDlyLine*）。
- 各引数名にはその機能性を指定する。ソース引数は *pSrc* または *src* と呼ばれ、その後に名前または番号が続くことがある（例えば、*pSrc2*、*srcLen*）。出力引数は *pDst* のように指定するか、*dst* の後に名前や数字を続けて指定する（例 : *pDst*、*dstLen*）。インプレース演算の場合は、入力 / 出力引数に名前 *pSrcDst* が入る。

構造体と列挙子

この項では、信号処理用インテル® IPP で使用される構造体と列挙子を説明する。

ライブラリ・バージョン構造体

`IppLibraryVersion` 構造体は、現在のインテル® IPP ソフトウェア・バージョンを記述する。この構造体の主なフィールドを次に示す。

- 整数フィールドの *major* と *minor*。バージョン番号を格納する。
- 文字列フィールドの *Name*。インテル IPP バージョン名を格納する。例 : "ippsa6"
- 文字列フィールドの *Version*。バージョンの説明を格納する。例 : "v0.0 Alpha 0.0.3.3"

複素数データ構造体

インテル® IPP では複素数を記述する場合、2 つのデータ・タイプ（複素数の実数部と虚数部）を含む構造体を使用する。例えば、単精度複素数は、次のように、`Ipp32fc` 構造体を使用して記述する。

```
typedef struct {
    Ipp32f re;
    Ipp32f im;
} Ipp32fc;
```

複素数データ・タイプは、次のものが定義されている : `Ipp16sc`、`Ipp32fc`、`Ipp64fc`。

関数コンテキスト構造体

フィルタリング、フーリエ変換、ウェーブレット変換などの演算を実行するインテル® IPP 関数の中には、コンテキスト構造体を使用して、関数固有の情報を格納するものがある。

例えば、`IppsFFTSpec` 構造体は、高速フーリエ変換で必要な回転因子とビット反転インデックスを格納する。

2 種類の構造体を使用する。名前に `Spec` サフィックスが付いている構造体は、関数演算中に変更されない名前に `State` サフィックスが付いている構造体は、演算中に変更される。

コンテキストに関連するこれらの構造体はパブリック・ヘッダに定義されてないため、構造体のフィールドにアクセスできない。この理由は、関数コンテキストの解釈がプロセッサごとに異なるからである。したがって、ユーザが使用できるのはコンテキスト関連の関数のみであり、自動変数として関数コンテキストを作成できない。

列挙子

IppStatus 定数は、インテル® IPP 関数により返されるステータス値を列挙し、演算にエラーがなかったかどうかを示す。信号処理関数における有効なステータス値とそれに対応するエラー・メッセージについては、2-11 ページの「エラーの報告」を参照のこと。

IppCmpOp 列挙は、5-64 ページのしきい値関数で使用する関係演算子のタイプを定義する。

```
typedef enum {
    ippCmpLess,
    ippCmpLessEq,
    ippCmpEq,
    ippCmpGreaterEq,
    ippCmpGreater
} IppCmpOp;
```

IppRoundMode 列挙は、5-51 ページの変換関数で使用する丸めモードを定義する。

```
typedef enum {
    ippRndZero,
    ippRndNear
} IppRoundMode;
```

IppHintAlgorithm 列挙は、特定の変換演算で使用するコードのタイプを定義する（すなわち、高速だが精度が低いコード、またはその逆に精度は高いが低速のコード）。この列挙子の詳しい使用方法は、「flag 引数と hint 引数」を参照のこと。

```
typedef enum {
    ippAlgHintNone,
    ippAlgHintFast,
    ippAlgHintAccurate
} IppHintAlgorithm;
```

IppCpuType は、ippGetCpuType 関数によって返されるプロセッサ・タイプを列挙する。3-7 ページを参照のこと。

```
typedef enum {
 /* Enumeration:          Processor:                           */
    ippCpuUnknown = 0x0,
    ippCpuPP,          /* Pentium(R)                          */
    ippCpuPMX,         /* Pentium(R) with MMX(TM)             */
    ippCpuPPR,         /* Pentium(R) Pro                      */
    ippCpuPII,         /* Pentium(R) II                       */
    ippCpuPIII,        /* Pentium(R) III                      */
    ippCpuP4,          /* Pentium(R) 4                        */
    ippCpuP4HT,        /* Pentium(R) 4 with HT enabled        */
    ippCpuITP = 0x10, /* Itanium(R)                          */
    ippCpuITP2         /* Itanium(R) 2                        */
} IppCpuType;
```

IppWinType 列挙は、6-48 ページの FIR フィルタ係数を計算する関数で使用されるウィンドウのタイプを定義する。

```
typedef enum {
    ippWinBartlett,
    ippWinBlackman,
    ippWinHamming,
    ippWinHann,
    ippWinRect
} IppWinType;
```

データ範囲

各データ・タイプで表現可能な値の範囲は、下限と上限で指定される。次の表に、各データ範囲と、範囲の上下限を示すインテル® IPP の不変識別子を示す。

表 2-2　データ・タイプと範囲

データ・タイプ	下限値		上限値	
	識別子	値	識別子	値
8s	IPP_MIN_8S	-128	IPP_MAX_8S	127
8u		0	IPP_MAX_8U	255
16s	IPP_MIN_16S	-32768	IPP_MAX_16S	32767
16u		0	IPP_MAX_16U	65535
32s	IPP_MIN_32S	-2^{31}	IPP_MAX_32S	$2^{31}-1$
32u		0	IPP_MAX_32U	$2^{32}-1$
32f †	IPP_MINABS_32F	$1.175494351e^{-38}$	IPP_MAXABS_32F	$3.402823466e^{38}$
64s	IPP_MIN_64S	-2^{63}	IPP_MAX_64S	$2^{63}-1$

表 2-2　データ・タイプと範囲

データ・タイプ	下限値		上限値	
	識別子	値	識別子	値
64f †	IPP_MINABS_64F	2.2250738585072014e-308	IPP_MAXABS_64F	1.7976931348623158e308

† 絶対値の範囲

データ・アライメント

インテル® IPP はコンパイラ・オプション /Zp16 を使用してビルドされる。/Zp16 は、フィールド・サイズ、または 16 バイト（サイズが 16 バイトを超える場合）で、構造体フィールドを整列する。

インテル IPP では、関数 ippsMalloc を使用して、割り振られたメモリ・ポインタを 32 バイトで整列もできる。

整数のスケーリング

整数データを演算する信号処理関数の中には、内部的に計算された出力結果を整数 *scaleFactor* を使用してスケーリングするものがある（scaleFactor は、関数の引数の一つとして指定される）。これらの関数の名前には、Sfs ディスクリプタが含まれる。

スケール係数には、負の値、正の値、またはゼロを指定できる。スケーリングが適用される理由は、通常内部計算は入力信号や出力信号で使用されるデータ・タイプよりも高い精度で実行されるからである。

注： スケーリングを使用している場合であっても、整数演算の結果は常にデスティネーション・データ・タイプの範囲に飽和される。

整数の結果をスケーリングするには、関数が返される前に、出力ベクトル値に $2^{-scaleFactor}$ を掛ければよい。これにより、出力データの範囲またはその精度が保持される。通常は、シフト演算により、正の係数を指定したスケーリングを実行する。結果は、最も近い整数に切り捨てられる。

例えば、入力値 200 に対する 2 乗演算 ippsSqr の結果である整数 Ipp16s は、40000 ではなく 32767 となる。すなわち、結果は飽和され、正確な値は復元できない。

係数 *scaleFactor* = 1 を使用して出力値をスケーリングすると、20000 の結果が得られる。この結果は飽和処理されないため、20000*2 として正確な値を復元できる。したがって、出力データの範囲は保持される。

次の例では、スケーリングにより精度の一部を保持する方法を示す。

入力値 2 に対して整数平方根演算 ippsSqrt（スケーリングなし）を実行すると、1.414 ではなく、1 という結果が得られる。内部的に計算した出力値を係数 *scaleFactor* = −3 でスケーリングすると、11 の結果が得られる。この方法では、より精度の高い値である $11*2^{-3} = 1.375$ が復元される。

エラー・レポート

インテル® IPP 関数は、実行した演算のステータスを返すことにより、呼び出し側プログラムにエラーと警告を報告する。したがって、エラーに関連する処置やエラーからの回復は、アプリケーション側で行う必要がある。エラー・ステータスの最後の値は保存されないため、ユーザは関数が返された時にそれをチェックするかどうかを決める必要がある。ステータス値は IppStatus タイプであり、グローバル定数整数である。

表 2-3 は、信号処理用インテル IPP により報告されるステータス・コードとそれに対応するメッセージの一覧である。

表 2-3 エラーのステータス、値、メッセージ

ステータス	メッセージ
ippStsCpuNotSupportedErr	The target cpu is not supported （対象とする CPU はサポートされていない）
ippStsMP3FrameHeaderErr	Error in fields IppMP3FrameHeader structure （IppMP3FrameHeader 構造体のフィールド・エラー）
ippStsMP3SideInfoErr	Error in fields IppMP3SideInfo structure （IppMP3SideInfo 構造体のフィールド・エラー）
ippStsAacPrgNumErr	AAC: Invalid number of elements for one program （1 つのプログラムに対する要素の数が無効）
ippStsAacSectCbErr	AAC: Invalid section codebook （セクションのコードブックが無効）
ippStsAacSfValErr	AAC: Invalid scalefactor value （スケール係数が無効）
ippStsAacCoefValErr	AAC: Invalid quantized coefficient value （量子化された係数の値が無効）
ippStsAacMaxSfbErr	AAC: Invalid coefficient index (係数インデックスが無効)
ippStsAacPredSfbErr	AAC: Invalid predicted coefficient index （予測係数インデックスが無効)

表 2-3 エラーのステータス、値、メッセージ （続き）

ippStsAacPlsDataErr	AAC: Invalid pulse data attributes (パルスのデータ属性が無効)
ippStsAacGainCtrErr	AAC: Gain control not supported (ゲイン・コントロールはサポートされていない)
ippStsAacSectErr	AAC: Invalid number of sections (セクションの数が無効)
ippStsAacTnsNumFiltErr	AAC: Invalid number of TNS filters (TNS フィルタの数が無効)
ippStsAacTnsLenErr	AAC: Invalid TNS region length (TNS 領域の長さが無効)
ippStsAacTnsOrderErr	AAC: Invalid order of TNS filter (TNS フィルタの順序が無効)
ippStsAacTnsCoefResErr	AAC: Invalid bit-resolution for TNS filter coefficients (TNS フィルタ係数のビット数が無効)
ippStsAacTnsCoefErr	AAC: Invalid TNS filter coefficients (TNS フィルタ係数が無効)
ippStsAacTnsDirectErr	AAC: Invalid TNS filter direction (TNS フィルタの方向が無効)
ippStsAacTnsProfileErr	AAC: Invalid TNS profile (TNS プロファイルが無効)
ippStsAacErr	AAC: Internal error (内部エラー)
ippStsAacBitOffsetErr	AAC: Invalid current bit offset in bitstream (ビットストリームの現在のビット・オフセット値が無効)
ippStsAacAdtsSyncWordErr	AAC: Invalid ADTS syncword (ADTS 同期ワードが無効)
ippStsAacSmplRateIdxErr	AAC: Invalid sample rate index (サンプル・レートのインデックスが無効)
ippStsAacWinLenErr	AAC: Invalid window length (not short or long) (ウィンドウの長さが無効 (長い))
ippStsAacWinGrpErr	AAC: Invalid number of groups for current window length (現在のウィンドウの長さに対してグループの数が無効)
ippStsAacWinSeqErr	AAC: Invalid window sequence range (ウィンドウ・シーケンスの範囲が無効)
ippStsAacComWinErr	AAC: Invalid common window flag (共通ウィンドウ・フラグが無効)
ippStsAacStereoMaskErr	AAC: Invalid stereo mask (ステレオ・マスクが無効)
ippStsAacChanErr	AAC: Invalid channel number (チャネル番号が無効)
ippStsAacMonoStereoErr	AAC: Invalid mono-stereo flag (モノステレオ・フラグが無効)
ippStsAacStereoLayerErr	AAC: Invalid this Stereo Layer flag (この Stereo Layer フラグは無効)
ippStsAacMonoLayerErr	AAC: Invalid this Mono Layer flag (この Mono Layer フラグは無効)

表 2-3　エラーのステータス、値、メッセージ （続き）

ippStsAacScalableErr	AAC: Invalid scalable object flag （スケーラブル・オブジェクト・フラグが無効）
ippStsAacObjTypeErr	AAC: Invalid audio object type （オーディオ・オブジェクトのタイプが無効）
ippStsAacWinShapeErr	AAC: Invalid window shape （ウィンドウの形状が無効）
ippStsAacPcmModeErr	AAC: Invalid PCM output interleaving indicator （PCM 出力インターリービング・インジケータが無効）
ippStsVLCUsrTblHeaderErr	VLC: Invalid header inside table （テーブル内のヘッダが無効）
ippStsVLCUsrTblUnsupportedFmtErr	VLC: Unsupported table format （テーブル形式がサポートされていない）
ippStsVLCUsrTblEscAlgTypeErr	VLC: Unsupported Ecs-algorithm （Ecs アルゴリズムがサポートされていない）
ippStsVLCUsrTblEscCodeLengthErr	VLC: Incorrect Esc-code length inside table header （テーブルのヘッダ内の Esc コードの長さが正しくない。）
ippStsVLCUsrTblCodeLengthErr	VLC: Unsupported code length inside table （テーブル内のコードの長さがサポートされていない）
ippStsVLCInternalTblErr	VLC: Invalid internal table （内部テーブルが無効）
ippStsVLCInputDataErr	VLC: Invalid input data （入力データが無効）
ippStsVLCAACEscCodeLengthErr	VLC: Invalid AAC-Esc code length （AAC-Esc コードの長さが無効）
ippStsIncorrectLSPErr	Incorrect Linear Spectral Pair values （LSP 値が無効）
ippStsNoRootFoundErr	No roots are found for equation （式に根がない）
ippStsLengthErr	Wrong value of string length （文字列の長さの値が誤っている）
ippStsFBankFreqErr	Incorrect value of the filter bank frequency parameter （フィルタ・バンク周波数パラメータの値が不適当）
ippStsFBankFlagErr	Incorrect value of the filter bank parameter （フィルタ・バンク・パラメータの値が不適当）
ippStsFBankErr	Filter bank is not correctly initialized （フィルタ・バンクが正しく初期化されていない）
ippStsNegOccErr	Negative occupation count （オキュペーション・カウントが負）
ippStsCdbkFlagErr	Incorrect value of the codebook flag parameter （コードブック・フラグ・パラメータの値が不適当）
ippStsSVDCnvgErr	No convergence of SVD algorithm （SVD アルゴリズムが収束しない）
ippStsToneMagnErr	Tone magnitude is less than or equal to zero （トーンの振幅がゼロ以下）
ippStsToneFreqErr	Tone frequency is negative, or greater than or equal to 0.5 （トーンの周波数が負または 0.5 以上）

表 2-3 エラーのステータス、値、メッセージ （続き）

ippStsTonePhaseErr	Tone phase is negative, or greater than or equal to 2*PI （トーンの位相が負または 2*π 以上）
ippStsTrnglMagnErr	Triangle magnitude is less than or equal to zero （トライアングルの振幅がゼロ以下）
ippStsTrnglFreqErr	Triangle frequency is negative, or greater than or equal to 0.5 （トライアングルの周波数が負または 0.5 以上）
ippStsTrnglPhaseErr	Triangle phase is negative, or greater than or equal to 2*PI （トライアングルの位相が負または 2*π 以上）
ippStsTrnglAsymErr	Triangle asymmetry is less than -PI, or greater than or equal to PI （トライアングルの非対称性が -π より小さいか、π 以上）
ippStsHugeWinErr	Incorrect size of the Kaiser window （Kaiser window のサイズが不適切）
ippStsJaehneErr	Magnitude value is negative （振幅の値が負）
ippStsStepErr	Step value is less than or equal to zero （ステップ値がゼロ以下）
ippStsDlyLineIndexErr	Invalid value of the delay line sample index （遅延線のサンプル・インデックスの値が無効）
ippStsStrideErr	Stride value is less than the row length （ストライド値が行のサイズよりも小さい）
ippStsEpsValErr	Negative epsilon value error （イプシロン値が負であることを示すエラー）
ippStsScaleRangeErr	Scale bounds are out of range （スケール境界が範囲外）
ippStsThresholdErr	Invalid threshold bounds （しきい値境界が無効）
ippStsWtOffsetErr	Invalid offset value of the wavelet filter （ウェーブレット・フィルタのオフセット値が無効）
ippStsAnchorErr	Anchor point is outside the mask （アンカー・ポイントがマスク外）
ippStsMaskSizeErr	Invalid mask size （マスク・サイズが無効）
ippStsShiftErr	Shift value is less than zero （シフト値がゼロより小さい）
ippStsSampleFactorErr	Sampling factor is less than or equal to zero （サンプリング係数がゼロ以下）
ippStsSamplePhaseErr	Phase value is out of range, 0 ≤ phase < factor （位相の値が範囲外、0 ≤ 位相 < 係数）
ippStsFIRMRFactorErr	MR FIR sampling factor is less than or equal to zero （MR FIR サンプリング係数がゼロ以下）
ippStsFIRMRPhaseErr	MR FIR sampling phase parameter is negative, or greater than or equal to the sampling factor （MR FIR サンプリング位相パラメータが負、またはサンプリング係数以上）

表 2-3　エラーのステータス、値、メッセージ （続き）

ippStsRelFreqErr	Relative frequency value is out of range （相対周波数の値が範囲外）
ippStsFIRLenErr	Length of a FIR filter is less than or equal to zero （FIR フィルタのサイズがゼロ以下）
ippStsIIROrderErr	Order of an IIR filter is less than or equal to zero （IIR フィルタの次数がゼロ以下）
ippStsResizeFactorErr	Resize factor(s) is less than or equal to zero （サイズ変更係数がゼロ以下）
ippStsDivByZeroErr	An attempt to divide by zero （ゼロで除算しようとした）
ippStsInterpolationErr	Invalid interpolation mode （補完モードが無効）
ippStsMirrorFlipErr	Invalid flip mode （フリップ・モードが無効）
ippStsMoment00ZeroErr	Moment value M(0,0) is too small to continue calculations （モーメント値 M（0,0）が小さすぎて、計算を続けられない）
ippStsThreshNegLevelErr	Negative value of the level in the threshold operation （しきい値演算のレベルの値が負）
ippStsContextMatchErr	Context parameter doesn't match the operation （コンテキスト・パラメータが演算に一致しない）
ippStsFftFlagErr	Invalid value of the FFT flag parameter （FFT フラグ・パラメータの値が無効）
ippStsFftOrderErr	Invalid value of the FFT order parameter （FFT 次数パラメータの値が無効）
ippStsMemAllocErr	Not enough memory allocated for the operation （演算用に十分なメモリが割り当てられていない）
ippStsNullPtrErr	Null pointer error （ヌル・ポインタ・エラー）
ippStsSizeErr	Wrong value of the data size （データ・サイズの値が誤っている）
ippStsBadArgErr	Function argument/parameter is bad （関数の引数 / パラメータが不適当）
ippStsErr	Unknown/unspecified error （未知 / 未指定のエラー）
ippStsNoErr	No error, it's OK （エラーなし、OK）
ippStsNoOperation	No operation has been executed （演算が実行されなかった）
ippStsMisalignedBuf	Misaligned pointer in operation in which it must be aligned （アライメントが必要であるにもかかわらず、アライメントが合っていないポインタが演算中に見つかった）
ippStsSqrtNegArg	Negative value(s) of the argument in the function Sqrt （関数 Sqrt の引数の値が負）
ippStsInvByZero	Inf result.Zero value was met by InvThresh with zero level （結果が Inf。ゼロ・レベルを指定した InvThresh がゼロの値を検出した）

表 2-3　エラーのステータス、値、メッセージ （続き）

ippStsEvenMedianMaskSize	Even size of the Median Filter mask was replaced by the odd one （偶数サイズのメディアン・フィルタ・マスクが、奇数サイズのマスクで置き換えられた）
ippStsDivByZero	Zero value(s) of the divisor in the function Div （関数 Div 内の除数の値がゼロ）
ippStsLnZeroArg	Zero value(s) of the argument in the function Ln （関数 Ln 内の引数の値がゼロ）
ippStsLnNegArg	Negative value(s) of the argument in the function Ln （関数 Ln 内の引数の値が負）
ippStsNanArg	Not a Number (NaN) argument value warning （引数の値が NaN（Not a Number）の警告）
ippStsResFloor	All result values are floored （すべての演算結果の値が下限）
ippStsOverflow	Overflow occurred in the operation （演算中にオーバーフローが発生した）
ippStsZeroOcc	Zero occupation count （オキュペーション・カウントがゼロ）
ippStsUnderflow	Underflow occurred in the operation （演算中にアンダーフローが発生した）
ippStsSingularity	Singularity occurred in the operation （演算中に特異点が発生した）
ippStsDomain	Argument is out of the function domain （引数が関数領域外）
ippStsNotIntelCpu	The target cpu is not Genuine Intel （対象とする CPU がインテル純正でない）
ippStsCpuMismatch	The library for given cpu cannot be set （指定した CPU のライブラリを設定できない）
ippStsNotIppFunctionFound	Application does not contain IPP functions calls （アプリケーションに IPP 関数呼び出しが含まれていない）
ippStsDllNotFoundBestUsed	The newest version of IPP DLL's not found by dispatcher （ディスパッチャが IPP DLL の最新バージョンを見つけられない）
ippStsNoOperationInDll	The function does nothing in the dynamic version of the library （この関数はダイナミック版のライブラリ内では何も実行しない）
ippStsOvermuchStrings	Number of destination strings is more than expected （デスティネーション文字列の数が予想以上に多い）
ippStsOverlongString	Length of one of the destination strings is more than expected （デスティネーション文字列の長さが予想以上に長い）

Err で終わるステータス・コード（ippStsNoErr ステータスは除く）は、該当するコードの整数値が負であるというエラーを示す。エラーが発生すると、関数の実行は中断される。
それ以外のステータス・コードはすべて警告を示す。特定の問題を検出すると、関数の実行の完了後に、対応する警告ステータスが返される。

例えば、整数関数 ippsDiv_8u は、正の値をゼロで除算しようとする演算を検出しても、関数の実行を中断しない。演算の結果は、ソース・データ・タイプで表現可能な最大の値に設定され、関数は警告ステータス ippStsDivByZero を返す。この動作は、ベクトル - ベクトル演算 ippsDiv の場合に適用される。ベクトル - スカラ除算 ippsDivC の場合は、関数の動作はこれとは異なる。すなわち、定数の除数が 0 である場合、関数の実行は中止され、ただちにエラー・ステータス ippStsDivByZeroErr が返される。

コード例

本書には、インテル® IPP 関数を使用するコード例が多数記載されている。これらの例は、プリミティブの機能とプリミティブの呼び出し方法を示している。これらのコード例の多くは、エラー状態または警告状態が検出された場合、演算結果のデータと一緒に、ステータス・コードとそれに関連するメッセージを出力する。

コード例を簡単にするために、print 文の特殊な定義を使用して、出力文字列によって各種の形式の実数と複素数の演算結果がわかりやすく表現され、プリント・ステータス・コードおよびメッセージも出力されるように指定する。

以下に示すコードの定義を使用して、コードのコピーと貼り付けを直接行うと、本書に記載されているコード例を作成できる。

```
/// the functions providing simple output of the result
/// they are for real and complex data

#define genPRINT(TYPE,FMT) \
void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus
st ) { \
   int n; \
   if( st > ippStsNoErr ) \
      printf( "-- warning %d, %s\n", st, ippGetStatusString( st )); \
   else if( st < ippStsNoErr ) \
      printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
   printf(" %s ", msg ); \
   for( n=0; n<len; ++n ) printf( FMT, buf[n] ); \
   printf("\n" ); \
}

#define genPRINTcplx(TYPE,FMT) \
```

```
void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus
st ) { \
   int n; \
   if( st > ippStsNoErr ) \
      printf( "-- warning %d, %s\n", st, ippGetStatusString( st )); \
   else if( st < ippStsNoErr ) \
      printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
   printf(" %s ", msg ); \
   for( n=0; n<len; ++n ) printf( FMT, buf[n].re, buf[n].im ); \
   printf("\n" ); \
}
genPRINT( 64f, " %f" )
genPRINT( 32f, " %f" )
genPRINT( 16s, " %d" )

genPRINTcplx( 64fc, " {%f,%f}" )
genPRINTcplx( 32fc, " {%f,%f}" )
genPRINTcplx( 16sc, " {%d,%d}" )
```

サポート関数

3

本章では、以下の目的で使用されるインテル® IPP サポート関数について説明する。

- インテル IPP ソフトウェアの現在のバージョンに関する情報を取得する
- 他のインテル IPP 関数の操作に必要なメモリの割り当てと解放を行う
- 特定の補助的操作を実行する

すべてのサポート関数を表 3-1 に示す。

表 3-1 インテル® IPP サポート関数

関数の基本名	操作
バージョン情報関数	
GetLibVersion	アクティブなライブラリのバージョンに関する情報を返す。
メモリ割り当て関数	
ippsMalloc	32 バイト境界にアライメントされたメモリを割り当てる。
ippsFree	関数 ippsMalloc により割り当てられたメモリを解放する。
共通の関数	
CoreGetStatusString	ステータス・コードをメッセージに変換する
CoreGetCpuType	プロセッサのタイプを返す。
CoreGetCpuClocks	タイムスタンプ・カウンタ（TSC）レジスタの現在値を返す。
CpuFreqMhz の取得	プロセッサの周波数を推定する。
CoreSetDenormAreZeros	ゼロ・フラッシュ・モードをイネーブルまたはディスエーブルにする。
AlignPtr	デノーマル・ゼロ・モードをイネーブルまたはディスエーブルにする。
ippMalloc	指定されたバイト数に合わせてポインタをアライメントする。
ippFree	32 バイト境界にアライメントされたメモリを割り当てる。
CoreGetStatusString	関数 ippMalloc によって割り当てられたメモリを解放する。
ディスパッチャ制御関数	
StaticInit	マージされたライブラリのスマート・ディスパッチを実行する。
StaticFree	関数 ippStaticInit によって使用されたリソースを解放する。

表 3-1　インテル® IPP サポート関数 （続き）

関数の基本名	操作
StaticInitBest	最も妥当なスタティック・コードを初期化する。
StaticInitCpu	指定されたバージョンのスタティック・コードを初期化する。

バージョン情報関数

バージョン情報関数は、アクティブなインテル® IPP ソフトウェアのバージョン番号などの情報を返す。

GetLibVersion

アクティブなインテル® IPP 信号処理ソフトウェアのバージョンに関する情報を返す。

```
const IppLibraryVersion* ippsGetLibVersion(void);
```

説明

関数 ippsGetLibVersion は、ipps.h ファイルで宣言される。この関数は、現在のバージョンの信号処理用インテル® IPP ソフトウェアに関する情報が入っている静的なデータ構造体 IppLibraryVersion へのポインタを返す。返されるポインタは静的な変数を指すため、このポインタによって参照されるメモリを解放する必要はない。IppLibraryVersion 構造体の以下のフィールドが利用可能である。

major	現在のライブラリのバージョン番号（メジャー）。
minor	現在のライブラリのバージョン番号（マイナー）。
Name	現在のライブラリのバージョン名。
Version	ライブラリのバージョン文字列。
BuildDate	ライブラリのバージョンの実際のビルド日付。

例えば、ライブラリのバージョンが “v1.2 Beta”、ライブラリ名が “ippsm6”、ビルド日付が “Jul 20 99” の場合、この構造体の一連のフィールドは次のように設定される。

major=1, *minor*=2, *Name*=“ippsm6”,
Version = "v1.2 Beta", *BuildDate* = "Jul 20 99"

intel®

例 3-1 は、関数 ippsGetLibVersion の使用例を示している。

例 3-1　ippsGetLibVersion 関数の使用例

```
void libinfo(void) {
        const IppLibraryVersion* lib = ippsGetLibVersion();
        printf("%s %s %d.%d.%d.%d\n", lib->Name, lib->Version,
             lib->major, lib->minor, lib->majorBuild, lib->build);
}

Output:
    ippsa6 v0.0 Alpha 0.0.5.5
```

注： 信号処理領域で使用される各サブライブラリは、アクティブなライブラリのバージョンに関する情報を取得するために、ippsGetLibVersion によく似た関数を持っている。
これらの関数は、ippGetLibVersion、ippsrGetLibVersion、ippscGetLibVersion、ippvmGetLibVersion、ippmpGetLibVersion、ippacGetLibVersion である。これらの関数は、それぞれヘッダ・ファイル ippcore.h、ippsr.h、ippsc.h、ippvm.h、ippmp.h、ippac.h で宣言され、上で説明した関数と同じインターフェイスを持つ。

メモリ割り当て関数

この項では、整列したメモリ・ブロックを必須タイプのデータに割り当てたり、以前に割り当てたメモリを解放したりする信号処理用インテル® IPP 関数について説明する。割り当てるメモリ・サイズは、割り当てる要素の数（*len*）で指定される。

注： メモリ割り当て関数により割り当てられたメモリを解放できるのは、ippsFree() 関数だけである。

ippsMalloc

32 バイト境界にアライメントされたメモリを割り当てる。

```
Ipp8u* ippsMalloc_8u(int len);
Ipp16u* ippsMalloc_16u(int len);
Ipp32u* ippsMalloc_32u(int len);
Ipp8s* ippsMalloc_8s(int len);
Ipp16s* ippsMalloc_16s(int len);
Ipp32s* ippsMalloc_32s(int len);
Ipp64s* ippsMalloc_64s(int len);
Ipp32f* ippsMalloc_32f(int len);
Ipp64f* ippsMalloc_64f(int len);
Ipp8sc* ippsMalloc_8sc(int len);
Ipp16sc* ippsMalloc_16sc(int len);
Ipp32sc* ippsMalloc_32sc(int len);
Ipp64sc* ippsMalloc_64sc(int len);
Ipp32fc* ippsMalloc_32fc(int len);
Ipp64fc* ippsMalloc_64fc(int len);
```

引数

len	割り当てる要素の数。

説明

関数 ippsMalloc は、ipps.h ファイルで宣言される。この関数は、さまざまなデータ・タイプの要素に対し、32 バイト境界に整列されたメモリ・ブロックを割り当てる。

戻り値

ippsMalloc の戻り値は、整列されたメモリ・ブロックへのポインタである。使用可能なメモリがシステムにない場合は、NULL 値が返される。このブロックを解放するには、関数 ippsFree を使用する。

ippsFree

関数 ippsMalloc により割り当てられた
メモリを解放する。

```
void ippsFree(void* ptr);
```

引数

ptr	解放するメモリ・ブロックへのポインタ。*ptr* で指示されるメモリ・ブロックは、関数 ippsMalloc により割り当てられる。

説明

関数 ippsFree は、ipps.h ファイルで宣言される。この関数は、関数 ippsMalloc により割り当てられる整列済みメモリ・ブロックを解放する。

注 : malloc や calloc などの標準関数で割り当てられたメモリは、関数 ippsFree で解放することはできない。また、ippsMalloc で割り当てられたメモリは、free で解放できない。

共通の関数

この項では、すべての領域に共通の特殊な操作を実行するインテル® IPP 関数について説明する。例えば、関数 ippCoreGetStatusString は、現在のインテル IPP ソフトウェアによって返されたステータス・コードの簡単な説明を取得する。関数 ippCoreGetCpuType は、システム内のプロセッサのタイプを取得する。関数 ippCoreGetCpuClocks は、プロセッサのクロック周波数の現在の数値を取得する（この値は操作のタイミングに広く使用される）。関数 ippCoreSetFlushToZero と ippCoreSetDenormAreZeros は、特殊なプロセッサ・モードをイネーブルまたはディスエーブルにする。関数 ippAlignPtr は、ポインタのアライメントを実行する。関数 ippMalloc と ippFree は、32 バイト境界にアライメントされたメモリ・ブロックの割り当てと解放を行う。ディスパッチャ制御関数と呼ばれる特殊な関数サブセットは、マージされたスタティック・ライブラリのディスパッチを制御する。

この項で説明する関数はすべて、ippcore という名前の別個のサブライブラリにまとめられている。

CoreGetStatusString

ステータス・コードをメッセージに変換する。

```
const char* ippCoreGetStatusString(IppStatus StsCode);
```

引数

StsCode	ステータスのタイプを示すコード（表 2-2 を参照)。

説明

関数 ippCoreGetStatusString は、ippcore.h ファイルで宣言される。この関数は、タイプ IppStatus のステータス・コードに関連付けられているテキスト文字列へのポインタを返す。この関数を使用して、ユーザ向けのエラー・メッセージと警告メッセージを生成できる。返されるポインタは、内部スタティック・バッファへのポインタであるため、解放する必要はない。

以下のコード例は、関数 ippCoreGetStatusString の使用法を示している。インテル® IPP 関数を（この場合、NULL ポインタを指定して ippsAddC_16s_I を）呼び出すと、エラー・コード -8 が返される。ステータス情報関数は、このコードを、対応するメッセージ "Null Pointer Error" に変換する。

例 3-2 ippCoreGetStatusString 関数の使用例

```
void statusinfo(void) {
        IppStatus st = ippsAddC_16s_I (3, 0, 0);
        printf("%d : %s\n", st, ippCoreGetStatusString(st));
}

Output:
    -8, Null Pointer Error
```

CoreGetCpuType

プロセッサのタイプを返す。

```
IppCpuType ippCoreGetCpuType (void);
```

説明

関数 ippCoreGetCpuType は、ippcore.h ファイルで宣言される。この関数は、コンピュータ・システム内で使用されているプロセッサのタイプを検出し、該当する IppCpuType 変数値を返す。表 3-2 は、可能な値とその意味を示している。

表 3-2 インテル® IPP で検出されるプロセッサのタイプ

返される変数値	プロセッサのタイプ
ippCpuPP	インテル® Pentium® プロセッサ
ippCpuPMX	インテル® MMX® テクノロジ Pentium® プロセッサ
ippCpuPPR	インテル® Pentium® Pro プロセッサ
ippCpuPII	インテル® Pentium® II プロセッサ
ippCpuPIII	インテル® Pentium® III プロセッサ
ippCpuP4	インテル® Pentium® 4 プロセッサ
ippCpuP4HT	ハイパー・スレッディング・テクノロジ 対応インテル® Pentium® 4 プロセッサ
ippCpuITP	インテル® Itanium® プロセッサ

表 3-2　インテル® IPP で検出されるプロセッサのタイプ （続き）

返される変数値	プロセッサのタイプ
ippCpuITP2	インテル® Itanium® 2 プロセッサ
ippCpuUnknown	未知のプロセッサ

CoreGetCpuClocks

タイムスタンプ・カウンタ（TSC）レジスタの
現在値を返す。

```
Ipp64u ippCoreGetCpuClocks (void);
```

説明

関数 ippCoreGetCpuClocks は、ippcore.h ファイルで宣言される。この関数は、タイムスタンプ・カウンタ（TSC）レジスタの現在の状態を読み取り、カウンタの値を返す。現在使用しているチップセットが TSC の読み取りをサポートしていない場合は、ハードウェア例外が発生する可能性がある。

CpuFreqMhz の取得

プロセッサの周波数を推定する。

```
IppStatus ippGetCpuFreqMhz(int* pMhz);
```

引数

pMhz	出力結果へのポインタ。

説明

関数 ippGetCpuFreqMhz は、ippcore.h ファイルで宣言される。この関数は、プロセッサが動作する周波数を推定し、*pMhz* に格納される整数として、MH z で値を返す。
予測値はプロセッサのワークロードに基づいて変化するので、周波数の正確な値は保証されないことに注意しなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pMhz* ポインタが NULL。

CoreSetFlushToZero

**ゼロ・フラッシュ・モードをイネーブル
またはディスエーブルにする。**

```
IppStatus ippCoreSetFlushToZero(int value, unsigned int*
pUMask);
```

引数

value	MXCSR レジスタの対応するビットをセットまたはクリアするスイッチ。
pUMask	現在のアンダーフロー例外マスクへのポインタ。NULL に設定できる。

説明

関数 ippCoreSetFlushToZero は、ippcore.h ファイルで宣言される。この関数は、ストリーミング SIMD 拡張命令（SSE）をサポートするプロセッサのゼロ・フラッシュ（FTZ）モードをイネーブル（*value* が 0 でない場合）またはディスエーブル（*value* が 0 の場合）にする。FTZ モードは、SIMD 浮動小数点アンダーフロー状態に対するマスク応答を制御する。FTZ モードは、主にパフォーマンス上の理由で用意されている。アンダーフローがよく発生するアプリケーションで、アンダーフローの演算結果をゼロに丸めてもかまわない場合、FTZ モードをイネーブルにすると、精度が多少低下する代わりに、アプリケーションの実行が高速化される。

FTZ モードは、マスク・レジスタが特定の状態になっている場合にのみ使用できる。ippSetFlushToZero 関数は、マスク・レジスタの状態をチェックし、必要に応じてその状態を変更する。FTZ モードをディスエーブルにした後、マスク・レジスタの初期状態を復元できる。これを行うには、アプリケーション内で unsigned integer 型の変数を宣言し、ippFlushToZero 関数の *pUMask* パラメータ内でその変数を指定する必要がある。マスク・レジスタの初期状態はこの位置に保存され、後で復元できる。マスクの初期状態を復元する必要がない場合は、*pUMask* ポインタを NULL に設定できる。

戻り値

ippStsNoErr	エラーなし。
ippStsCpuNotSupportedErr	プロセッサが FTZ モードをサポートしていない場合のエラー状態を示す。

CoreSetDenormAreZeros

デノーマル・ゼロ・モードをイネーブル
またはディスエーブルにする。

```
IppStatus ippCoreSetDenormAreZeros(int value);
```

引数

value	MXCSR レジスタの対応するビットをセットまたはクリアするスイッチ。

説明

関数 ippCoreSetDenormAreZeros は、ippcore.h ファイルで宣言される。この関数は、ストリーミング SIMD 拡張命令（SSE）をサポートするプロセッサのデノーマル・ゼロ（DAZ）モードをイネーブル（*value* が 0 でない場合）またはディスエーブル（*value* が 0 の場合）にする。DAZ モードは、SIMD 浮動小数点デノーマル・オペランド状態に対するプロセッサの応答を制御する。DAZ フラグがセットされている場合、プロセッサは、すべてのデノーマル・ソース・オペランドを 0 に変換し、ソース・データの演算を実行する前の元のオペランドの符号を付ける。DAZ モードは、デノーマル・オペランドを 0 に丸めても処理されたデータの品質にそれほど影響を与えない、ストリーミング・メディア処理などのアプリケーションで、プロセッサのパフォーマンスを向上させるために用意されている。

戻り値

ippStsNoErr	エラーなし。
ippStsCpuNotSupportedErr	プロセッサが DAZ モードをサポートしていない場合のエラー状態を示す。

intel®

AlignPtr

指定されたバイト数に合わせてポインタを
アライメントする。

```
void* ippAlignPtr(void* ptr, int alignBytes);
```

引数

ptr	プロセッサのタイプ。
alignBytes	アライメントのバイト数。指定できる値は、2 の累乗、すなわち、2、4、8、16 などである。

説明

関数 ippAlignPtr は、ippcore.h ファイルで宣言される。この関数は、指定されたバイト数 *alignBytes* に合わせてアライメントされたポインタ *ptr* を返す。*alignBytes* で指定できる値は、2 の累乗である。この関数は、このパラメータの有効性をチェックしない。

注： この関数によって返されるポインタを解放してはならない。元のポインタを解放すること。

ippMalloc

32 バイト境界にアライメントされたメモリを
割り当てる。

```
void* ippMalloc(int length);
```

引数

len	割り当てる要素の数。

説明

関数 ippMalloc は、ippcore.h ファイルで宣言される。この関数は、32 バイト境界にアライメントされたメモリ・ブロックを割り当てる。

戻り値

ippMalloc の戻り値は、アライメントされたメモリ・ブロックへのポインタである。このブロックを解放するには、関数 ippFree を使用する必要がある。

ippFree

関数 ippMalloc によって割り当てられたメモリを解放する。

```
void ippFree(void* ptr);
```

引数

ptr	解放するメモリ・ブロックへのポインタ。

説明

関数 ippFree は、ippcore.h ファイルで宣言される。この関数は、関数 ippMalloc によって割り当てられた、アライメントの合ったメモリ・ブロックを解放する。

注： malloc など、ippMalloc 以外の関数によって割り当てられたメモリは、関数 ippFree では解放できない。また、ippMalloc によって割り当てられたメモリは、free では解放できない。

マージされたライブラリのディスパッチを制御する関数

この項では、マージされたスタティック・ライブラリのディスパッチャを制御するインテル® IPP 関数について説明する。

StaticInit

マージされたライブラリのスマート・
ディスパッチを実行する。

```
IppStatus ippStaticInit(void);
```

説明

関数 ippStaticInit は、ippcore.h ファイルで宣言される。この関数は、「スマート・ディスパッチ」と呼ばれる機能を実行する。スマート・ディスパッチとは、ディスパッチャがインテル® IPP DLL の最新バージョンを検索するという意味である。ディスパッチャが DLL の最新バージョンを見つけると、対応する DLL がロードされる。
DLL の最新バージョンが見つからない場合は、ディスパッチャは、関数 ippStaticInitBest を呼び出して、インテル IPP ソフトウェアの最も妥当なスタティック・コードを設定する。

ドライバのコード内では、関数 ippStaticInit を使用してはならない。

戻り値

ippStsNoErr	インテル IPP DLL の最新バージョンが正常に検出され、ロードされたことを示す。
ippStsNoIppFunctionFound	アプリケーションがインテル IPP 関数への呼び出しを含んでいないか、またはマージされていないインテル IPP ライブラリを使用して作成されたことを示す。
IppStsDllNotFoundBestUsed	ディスパッチャがインテル IPP DLL の最新バージョンを見つけられなかったため、最も妥当なスタティック・コードが使用されることを示す。
ippStsNoOperationInDll	ライブラリのダイナミック・バージョン内に、このような操作が存在しないことを示す。

StaticFree

関数 ippStaticInit によって使用された
リソースを解放する。

```
IppStatus ippStaticFree(void);
```

説明

関数 ippStaticFree は、ippcore.h ファイルで宣言される。この関数は、関数 ippStaticInit によって使用されたリソースを解放し、関数 ippStaticInitBest を呼び出す。関数 ippStaticFree は、繰り返し呼び出すことが可能である。

ドライバのコード内では、関数 ippStaticFree を使用してはならない。

戻り値

ippStsNoErr	リソースが解放され、関数 ippStaticInitBest が正常に呼び出されたことを示す。
ippStsNonIntelCpu	インテル・アーキテクチャ向け汎用コードのスタティック・バージョンが設定されたことを示す。
ippStsNoOperationInDll	ライブラリのダイナミック・バージョン内に、このような操作が存在しないことを示す。

StaticInitBest

最も妥当なスタティック・コードを初期化する。

```
IppStatus ippStaticInitBest(void);
```

説明

関数 ippStaticInitBest は、ippcore.h ファイルで宣言される。この関数は、ユーザのコンピュータ・システム内で使用されているプロセッサのタイプを検出し、インテル® IPP ソフトウェアのプロセッサ固有のスタティック・コードのうち最も妥当なものを設定する。

戻り値

ippStsNoErr	インテル IPP ソフトウェアの最も妥当なスタティック・コードが正常に設定されたことを示す。
ippStsNonIntelCpu	インテル・アーキテクチャ向け汎用コードのスタティック・バージョンが設定されたことを示す。
ippStsNoOperationInDll	ライブラリのダイナミック・バージョン内に、このような操作が存在しないことを示す。

StaticInitCpu

指定されたバージョンのスタティック・コードを初期化する。

```
IppStatus ippStaticInitCpu(IppCpuType cpu);
```

引数

cpu	プロセッサのタイプ。

説明

関数 `ippStaticInitCpu` は、`ippcore.h` ファイルで宣言される。この関数は、指定されたプロセッサのタイプ *cpu* に従って、インテル® IPP ライブラリのプロセッサ固有のスタティック・コードを設定する。

戻り値

ippStsNoErr	必要なプロセッサ固有のスタティック・コードが正常に設定されたことを示す。
ippStsCpuMismatch	指定されたプロセッサのタイプが有効でないため、以前に設定されたコード・バージョンが使用されることを示す。
ippStsNoOperationInDll	ライブラリのダイナミック・バージョン内に、このような操作が存在しないことを示す。

バージョン 2.0 との互換性

インテル® IPP バージョン 2.0 の共通関数の一部は、インテル IPP バージョン 3.0 の ippcore サブライブラリの関数で置き換えられた。互換性を維持するために、バージョン 3.0 では、これらの古い関数の使用をサポートしている。表 3-3 は、古い（置き換えられた）関数と新しい関数の両方を示している。

表 3-3　インテル® IPP 3.0 で置き換えられた関数

置き換えられた関数	新しい関数
ippGetStatusString	ippCoreSetStatusString
ippGetCpuType	ippCoreGetCpuType
ippGetCpuClocks	ippCoreGetCpuClocks
ippSetFlushToZero	ippCoreSetFlushToZero
ippSetDenormAreZeros	ippCoreSetDenormAreZeros

intel®

ベクトル初期化関数

4

本章では、定数、他のベクトルの内容、または生成された信号でベクトルを初期化するインテル® IPP 関数について説明する。

このグループのすべての関数を表 4-1 に示す。

表 4-1　インテル® IPP ベクトル初期化関数

関数の基本名	操作
ベクトル初期化関数	
Copy	ベクトルの内容を別のベクトルにコピーする。
Move	ベクトルの内容を別のベクトルに移動する。
Set	指定したコモン値にベクトル要素を初期化する。
Zero	ベクトルをゼロに初期化する。
トーンおよびトライアングル生成関数	
ToneInitAllocQ15	メモリを割当てて、トーン・ジェネレータ・ステートを初期化する。
ToneFree	関数 ippsToneInitAlloc によって割り当てられたメモリを解放する。
ToneGetStateSizeQ15	トーン・ジェネレータ構造体の長さを計算する。
ToneInitQ15	トーン・ジェネレータ指定構造体を初期化する。
ToneQ15	トーン・ジェネレータ指定に基づいてトーンを生成する。
Tone_Direct	特定の周波数、位相、大きさを持つトーンを生成する。
ToneQ15_Direct	特定の周波数、位相、大きさを持つトーンを生成する。
TriangleInitAllocQ15	メモリを割当てて、トライアングル・ジェネレータ・ステートを初期化する。
TriangleFree	関数 ippsTriangleInitAlloc によって割り当てられたメモリを解放する。
TriangleGetStateSizeQ15	トライアングル・ジェネレータ構造体の長さを計算する。
TriangleInitQ15	トライアングル・ジェネレータ指定構造体を初期化する。
TriangleQ15	トライアングル・ジェネレータ指定に基づいてトーンを生成する。
Triangle_Direct	特定の周波数、位相、大きさを持つトライアングルを生成する。

表 4-1 インテル® IPP ベクトル初期化関数 （続き）

関数の基本名	操作
Triangle015_Direct	特定の周波数、位相、大きさを持つトライアングルを生成する。
一様分布関数	
RandUniformInitAlloc	メモリを割り当て、ノイズ・ジェネレータを一様分布で初期化する。
RandUniformFree	一様分布ジェネレータ・ステートを終了する。
RandUniformGetSize	一様分布ジェネレータ構造体の長さを計算する。
RandUniformInit	ノイズ・ジェネレータを一様分布で初期化する。
RandUnifrom	一様分布を持つ疑似ランダム・サンプルを生成する。
RandUniform_Direct	一様分布を持つ疑似ランダム・サンプルを直接モードで生成する。
ガウス分布関数	
RandGaussInitAlloc	メモリを割り当て、ノイズ・ジェネレータをガウス分布で初期化する。
RandGaussFree	ガウス分布ジェネレータ・ステートを終了する。
RandGaussGetSize	ガウス分布ジェネレータ・ステートの長さを計算する。
RandGaussInit	ノイズ・ジェネレータをガウス分布で初期化する。
RandGauss	ガウス分布を持つ疑似ランダム・サンプルを生成する。
RandGauss_Direct	ガウス分布を持つ疑似ランダム・サンプルを直接モードで生成する。
特殊ベクトル関数	
VectorJaehne	Jaehne ベクトルを生成する。
VectorRamp	ランプ・ベクトルを生成する。

ベクトル初期化関数

この項では、ベクトル要素の値を初期化する関数を説明する。
ベクトル要素はすべて初期値ゼロ、または他の指定された値に初期化できる。また、ベクトル要素は 2 番目のベクトル要素の各値に初期化もできる。

Copy

ベクトルの内容を別のベクトルにコピーする。

```
IppStatus ippsCopy_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsCopy_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsCopy_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
```


intel®

```
IppStatus ippsCopy_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCopy_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCopy_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len );
IppStatus ippsCopy_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

引数

pSrc	*pDst* を初期化するのに使用するソース・ベクトルへのポインタ。
pDst	初期化するデスティネーション・ベクトルへのポインタ。
len	コピーする要素の数。

説明

関数 ippsCopy は、ipps.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc* からデスティネーション・ベクトル *pDst* に、最初の *len* 個の要素をコピーする。

注： これらの関数は、上記のコピー操作だけを実行するだけで、データを移動するためのものではない。ソース・バッファとデスティネーション・バッファが重なる場合、これらの関数の動作は予測不可能である。
データを移動するには、関数 ippsMove を使用する必要がある。

例 4-1 は、関数 ippsCopy の使用例を示している。

例 4-1 ippsCopy 関数の使用例

```
IppStatus copy(void) {
      char src[] = "to be copied\0";
      char dst[256];
      return ippsCopy_8u(src, dst, strlen(src)+1);
}
```

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。*pDst* ポインタまたは *pSrc* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Move

ベクトルの内容を別のベクトルに移動する。

```
IppStatus ippsMove_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsMove_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMove_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsMove_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMove_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsMove_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsMove_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

引数

pSrc	*pDst* を初期化するのに使用するソース・ベクトルへのポインタ。
pDst	初期化するデスティネーション・ベクトルへのポインタ。
len	移動する要素の数。

説明

関数 ippsMove は、ipps.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc* からデスティネーション・ベクトル *pDst* に最初の *len* 個の要素を移動する。ソース・ベクトルとデスティネーション・ベクトルの一部が重なる場合、この関数は、重なる部分の元のソース・バイトがデスティネーション・ベクトルの該当する部分に移動される（つまり、上書きされる前にコピーされる）ことを保証する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタまたは *pSrc* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Set

指定したコモン値にベクトル要素を
初期化する。

```
IppStatus ippsSet_8u(Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsSet_16s(Ipp16s val, Ipp16s* pDst, int len);
IppStatus ippsSet_16sc(Ipp16sc val, Ipp16sc* pDst, int len);
IppStatus ippsSet_32s(Ipp32s val, Ipp32s* pDst, int len);
IppStatus ippsSet_32f(Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsSet_32sc(Ipp32sc val, Ipp32sc* pDst, int len);
IppStatus ippsSet_32fc(Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsSet_64s(Ipp64s val, Ipp64s* pDst, int len);
IppStatus ippsSet_64f(Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsSet_64sc(Ipp64sc val, Ipp64sc* pDst, int len);
IppStatus ippsSet_64fc(Ipp64fc val, Ipp64fc* pDst, int len);
```

引数

pDst	初期化するベクトルへのポインタ。
len	初期化する要素の数。
val	ベクトル *pDst* を初期化するのに使用する値。

説明

関数 ippsSet は、ipps.h ファイルで宣言される。この関数は、実数または複素数ベクトル *pDst* の最初の *len* 個の要素を同じ値、*val* に初期化する。

例 4-2 は、関数 ippsSet の使用例を示している。

例 4-2　ippsSet 関数の使用例

```
IppStatus set(void) {
      char src[] = "set";
      return ippsSet_8u('0', src, strlen(src));
}
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Zero

ベクトルをゼロに初期化する。

```
IppStatus ippsZero_8u(Ipp8u* pDst, int len);
IppStatus ippsZero_16s(Ipp16s* pDst, int len);
IppStatus ippsZero_16sc(Ipp16sc* pDst, int len);
IppStatus ippsZero_32f(Ipp32f* pDst, int len);
IppStatus ippsZero_32fc(Ipp32fc* pDst, int len);
IppStatus ippsZero_64f(Ipp64f* pDst, int len);
IppStatus ippsZero_64fc(Ipp64fc* pDst, int len);
```

引数

pDst	ゼロに初期化するベクトルへのポインタ。
len	初期化する要素の数。

説明

関数 ippsZero は、ipps.h ファイルで宣言される。この関数は、ベクトル *pDst* の最初の *len* 個の要素をゼロに初期化する。*pDst* が複素数ベクトルの場合は、実数部と虚数部がともにゼロに初期化される。

例 4-3 は、関数 ippsZero の使用例を示している。

例 4-3 ippsZero 関数の使用例

```
IppStatus zero(void) {
   char src[] = "zero";
   return ippsZero_8u(src, strlen(src));
}
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

サンプル生成関数

本章では、トーン・サンプル、トライアングル・サンプル、一様分布を持つ疑似ランダム・サンプル、ガウス分布を持つ疑似ランダム・サンプル、特定のテスト・サンプルを生成するインテル® IPP 関数について説明する。

トーン生成関数

この項では、特定の周波数、位相、大きさを持つトーン（サイン・カーブ）を生成するための関数について説明する。トーンは、アナログ信号の基本要素である。そのため、サンプリングしたトーンは、テスト信号として、またより複雑な信号を生成するための基本要素として、信号処理システムで非常に役に立つ。

多くのアプリケーションでは、トーン関数によく似た C 演算ライブラリの sin() 関数よりも、トーン関数の方がよく使用される。これは、インテル® IPP 関数では、標準の sin() や cos() と比較してはるかに高速に、前のサンプルの計算で得られた情報を次のサンプルの計算に使用できるからである。

ToneInitAllocQ15

メモリを割り当て、ジェネレータ指定構造体を
初期化する。

```
IppStatus ippsToneInitAllocQ15_16s(IppToneState_16s**
     pToneState, Ipp16s magn, Ipp16s rFreqQ15, Ipp32s phaseQ15);
```

引数

pToneState	トーン・ジェネレータ指定構造体へのポインタ。
magn	トーンの大きさ。すなわち、波形の最大値を指定する。
phaseQ15	Q16.15 形式のコサイン波に対するトーンの位相。値は [0, 205886] の範囲内に収まっていなければならない。
rFreqQ15	Q0.15 形式のサンプリング周波数に対するトーンの周波数。値は [0, 16383] の範囲内に収まっていなければならない。

説明

関数 ippsToneInitAllocQ15 は、ipps.h ファイルで宣言される。この関数は、メモリを割り当て、指定された周波数 *rFreqQ15*、位相 *phaseQ15*、大きさ *magn* を持つ *pToneState* トーン・ジェネレータ構造体を初期化する。
Q15 形式のデータは、相対周波数値が [0, 0.5)、位相が [0, 2π) の範囲にある対応する浮動小数点データ・タイプに変換される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pToneState* ポインタが NULL。
ippStsToneMagnErr	エラー。*magn* がゼロ以下。
ippStsToneFreqErr	エラー。*rFreqQ15* が負、または 16383 より大きい。
ippStsTonePhaseErr	エラー。*phaseQ15* 値が負、または 205886 より大きい。

ToneFree

関数 ippsToneInitAllocQ15 によって
割り当てられたメモリを解放する。

```
IppStatus ippsToneFree(IppToneState_16s* pToneState);
```

引数

pToneState	トーン・ジェネレータ指定構造体へのポインタ。

説明

関数 ippsToneFree は、ipps.h ファイルで宣言される。この関数は、ippsippsToneInitAllocQ15 により生成された構造体に関連付けられているメモリをすべて解放してトーン・ジェネレータ・ステートをクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pToneState* ポインタが NULL。

ToneGetStateSizeQ15

トーン・ジェネレータ構造体の長さを
計算する。

```
IppStatus ippsToneGetStateSizeQ15_16s(int* pToneStateSize);
```

引数

pToneStateSize	ジェネレータ指定構造体のバイト単位で計算されたサイズ値へのポインタ。

説明

関数 ippsToneGetStateSizeQ15 は、ipps.h ファイルで宣言される。この関数は、トーン・ジェネレータ構造体の *pToneStateSize* の長さ（バイト単位）を計算する。関数 ippsToneGetStateSizeQ15 は、ippsippsToneInitQ15 関数を呼び出す前に呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pToneStateSize* ポインタが NULL。

ToneInitQ15

トーン・ジェネレータ指定構造体を
初期化する。

```
IppStatus ippsToneInitQ15_16s(IppToneState_16s* pToneState,
     Ipp16s magn, Ipp16s rFreqQ15, Ipp32s phaseQ15);
```

引数

pToneState	トーン・ジェネレータ指定構造体へのポインタ。
magn	トーンの大きさ。すなわち、波形の最大値を指定する。
phaseQ15	Q16.15 形式のコサイン波に対するトーンの位相。値は [0, 205886] の範囲内に収まっていなければならない。
rFreqQ15	Q0.15 形式のサンプリング周波数に対するトーンの周波数。値は [0, 16383] の範囲内に収まっていなければならない。

説明

関数 ippsToneInitQ15 は、ipps.h ファイルで宣言される。この関数は、指定された周波数 *rFreqQ15*、位相 *phaseQ15*、大きさ *magn* を持つ *pToneState* トーン・ジェネレータ構造体を初期化する。
この構造体は ippsippsToneGetStateSizeQ15 関数により求められるサイズの外部バッファに割り当てられる。Q15 形式のデータは、相対周波数値が [0, 0.5)、位相が [0, 2π) の範囲にある対応する浮動小数点データ・タイプに変換される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pToneState* ポインタが NULL。
ippStsToneMagnErr	エラー。*magn* がゼロ以下。
ippStsToneFreqErr	エラー。*rFreqQ15* が負、または 16383 より大きい。

intel®

ippStsTonePhaseErr	エラー。*phaseQ15* 値が負、または 205886 より大きい。

ToneQ15

トーン・ジェネレータ構造体で指定された周波数、
位相、大きさを持つトーンを生成する。

```
IppStatus ippsToneQ15_16s(Ipp16s* pDst, int len,
     IppToneState_16s* pToneState);
```

引数

pDst	サンプルを格納する配列へのポインタ。
len	計算するサンプルの数。
pToneState	トーン・ジェネレータ指定構造体へのポインタ。

説明

関数 ippsToneQ15 は、ipps.h ファイルで宣言される。この関数は、前に作成された *pToneState* 構造体で指定された周波数、位相、大きさの各パラメータを持つトーンを生成する。この関数は、*len* 個のトーン・サンプルを計算し、それらを配列 *pDst* に格納する。生成された値、x[*n*] は実数のトーンの計算に用いられた ippsTone_Direct 関数と同じ式で計算される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタまたは *pSeed* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Tone_Direct

特定の周波数、位相、大きさを持つトーンを
生成する。

```
IppStatus ippsTone_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn,
     float rFreq, float* pPhase, IppHintAlgorithm hint);
```

```
IppStatus ippsTone_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn,
      float rFreq, float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_32f(Ipp32f* pDst, int len, float magn,
      float rFreq, float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_32fc(Ipp32fc* pDst, int len, float magn,
      float rFreq, float* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_64f(Ipp64f* pDst, int len, double magn,
      double rFreq, double* pPhase, IppHintAlgorithm hint);
IppStatus ippsTone_Direct_64fc(Ipp64fc* pDst, int len, double magn,
      double rFreq, double* pPhase, IppHintAlgorithm hint);
```

引数

magn	トーンの大きさ。すなわち、波形の最大値を指定する。
pPhase	コサイン波に対するトーンの位相へのポインタ。値は [0.0, 2π) の範囲内に収まっていなければならない。戻り値は、次の連続するデータ・ブロックの計算に使用できる。
rFreq	サンプリング周波数に対するトーンの周波数。値は、実数のトーンの場合は [0.0, 0.5) 間隔、複素数のトーンの場合は [0.0, 1.0) 間隔でなければならない。
pDst	サンプルを格納する配列へのポインタ。
len	計算するサンプルの数。
hint	特別のコードの使用を指定する。*hint* 引数の値については、「flag 引数と hint 引数」を参照のこと。

説明

関数 `ippsTone_Direct` は、`ipps.h` ファイルで宣言される。この関数は、指定された周波数 *rFreq*、位相 *pPhase*、大きさ *magn* を持つトーンを生成する。この関数は、*len* 個のトーン・サンプルを計算し、それらを配列 *pDst* に格納する。実数のトーンの場合、それぞれの生成される値 $x[n]$ は次のように定義される。

$$x[n] = magn \cdot \cos(2\pi n \cdot rFreq + phase)$$

複素数のトーンの場合、$x[n]$ は次のように定義される。

$$x[n] = magn \cdot (\cos(2\pi n \cdot rFreq + phase) + j \cdot \sin(2\pi n \cdot rFreq + phase))$$

hint 引数には、特別なコード（計算は高速だが精度が低い、または計算の精度は高いが低速）の使用を指定する。*hint* 引数に指定可能な値は、表 7-3 に記載されている。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタまたは *pPhase* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsToneMagnErr	エラー。*magn* がゼロ以下。
ippStsToneFreqErr	エラー。*rFreq* が負、または実数のトーンで 0.5 以上および複素数のトーンで 1.0 以上。
ippStsTonePhaseErr	エラー。*pPhase* の値が負、または IPP_2PI 以上。

ToneQ15_Direct

特定の周波数、位相、大きさを持つトーンを生成する。

```
IppStatus ippsToneQ15_Direct_16s(Ipp16s* pDst, int len, Ipp16s
      magn, Ipp16s rFreqQ15, Ipp32s phaseQ15);
```

引数

pDst	サンプルを格納する配列へのポインタ。
magn	トーンの大きさ。すなわち、波形の最大値を指定する。
rFreqQ15	Q0.15 形式のサンプリング周波数に対するトーンの周波数。値は [0, 16383] の範囲内に収まっていなければならない。
phaseQ15	Q16.15　形式のコサイン波に対するトーンの位相。値は [0, 205886] の範囲内に収まっていなければならない。

説明

関数 ippsToneQ15_Direct は、ipps.h ファイルで宣言される。この関数は、指定された周波数 *rFreqQ15*、位相 *pPhaseQ15*、大きさ *magn* を持つトーンを生成する。Q15 形式のデータは、相対周波数値が [0, 0.5)、位相が [0, 2π) の範囲にある対応する浮動小数点データ・タイプに変換される。この関数は、*len* 個のトーン・サンプルを計算し、それらを配列 *pDst* に格納する。生成された値、x[*n*] は実数のトーンの計算に用いられた ippsTone_Direct 関数と同じ式で計算される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsToneMagnErr	エラー。*magn* がゼロ以下。
ippStsToneFreqErr	エラー。*rFreqQ15* が負、または 16383 より大きい。
ippStsTonePhaseErr	エラー。*phaseQ15* 値が負、または 205886 より大きい。

トライアングル生成関数

この項では、特定の周波数、位相、大きさ、非対称性の三角波形を持つ周期的な信号（トライアングルと呼ばれる）を生成する関数について説明する。

アプリケーション・ノート

特定の周波数 *rFreq*、位相 *phase*、大きさ *magn*、非対称性 *h* の三角波形を持つ実数の周期的信号 *x*[*n*]（実数のトライアングルと呼ばれる）は、次のように定義される。

$$x[n] = magn \cdot \mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase),\ \ n = 0, 1, 2, \ldots$$

特定の周波数 *rFreq*、位相 *phase*、大きさ *mag*、非対称性 *h* の三角波を持つ複素数の周期的信号 *x*[*n*]（複素数のトライアングルと呼ばれる）は、次のように定義される。

$$x[n] = magn \cdot (\mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase) + j \cdot \mathbf{st}_h(2\pi \cdot rFreq \cdot n + phase)),\ \ n = 0, 1, 2, \ldots$$

$\mathbf{ct}_h(\ )$ 関数は、次のように決定される。

$$H = p + h$$

$$\mathbf{ct}_h(\alpha) = \begin{cases} -\frac{2}{H} \cdot \left(\alpha - \frac{H}{2}\right), & 0 \le \alpha \le H \\ \frac{2}{2\pi - H} \cdot \left(\alpha - \frac{2\pi + H}{2}\right), & H \le \alpha \le 2\pi \end{cases}$$

$$\mathbf{ct}_h(\alpha + k \cdot 2\pi) = \mathbf{ct}_h(\alpha),\ \ k = 0, \pm 1, \pm 2, \ldots$$

intel®

$H=\pi$、非対称性 h=0 の場合、関数 $_{ct}$h() は対称になり、また cos() 関数の三角相似になる。次の式に注意する必要がある。

$$\mathbf{ct}_h(H/2 + k\cdot\pi) = 0,\ k = 0, \pm1, \pm2, \ldots$$

$$\mathbf{ct}_h(k\cdot 2\pi) = 1,\ k = 0, \pm1, \pm2, \ldots$$

$$\mathbf{ct}_h(H + k\cdot 2\pi) = -1,\ k = 0, \pm1, \pm2, \ldots$$

$\mathbf{st}_h()$ 関数は、次のように決定される。

$$\mathbf{st}_h(\alpha) = \begin{cases} \dfrac{2}{2\pi - H}\cdot\alpha,\ 0 \le \alpha \le \dfrac{2\pi - H}{2} \\ -\dfrac{2}{H}\cdot(\alpha - \pi),\ \dfrac{2\pi - H}{2} \le \alpha \le \dfrac{2\pi + H}{2} \\ \dfrac{2}{2\pi - H}\cdot(\alpha - 2\pi),\ \dfrac{2\pi + H}{2} \le \alpha \le 2\pi \end{cases}$$

$$\mathbf{st}_h(\alpha + k\cdot 2\pi) = \mathbf{st}_h(\alpha),\ k = 0, \pm1, \pm2, \ldots$$

$H=\pi$、非対称性 h=0 の場合、関数 $_{st}$h() は正弦関数の三角相似になる。次の式に注意する必要がある。

$$\mathbf{st}_h(\alpha) = \mathbf{ct}_h(\alpha + (3\pi + h)/2)$$

$$\mathbf{st}_h(\pi k) = 0,\ k = 0, \pm1, \pm2, \ldots$$

$$\mathbf{st}_h((\pi - h)/2 + 2\pi k) = 1,\ k = 0, \pm1, \pm2, \ldots$$

$$\mathbf{st}_h((3\pi + h)/2 + 2\pi k) = -1,\ k = 0, \pm1, \pm2, \ldots$$

TriangleInitAllocQ15

メモリを割り当て、トライアングル指定構造体を初期化する。

```
IppStatus ippsTriangleInitAllocQ15_16s(IppTriangleState_16s**
     pTriangleState, Ipp16s magn, Ipp16s rFreqQ15, Ipp32s
     phaseQ15, Ipp32s asymQ15);
```

引数

pTriangleState	トライアングル指定構造体へのポインタ。
magn	トーンの大きさ。すなわち、波形の最大値を指定する。
rFreqQ15	Q0.15 形式のサンプリング周波数に対するトーンの周波数。値は [0, 16383] の範囲内に収まっていなければならない。
phaseQ15	Q16.15 形式のコサイン波に対するトーンの位相。値は [0, 205886] の範囲内に収まっていなければならない。
asymQ15	Q16.15 形式のトライアングルの非対称性 *h*。値は [-102943, 102943] の範囲内に収まっていなければならない。*h*=0 の場合、トライアングルは対称になり、トーンの直接相似になる。

説明

関数 `ippsTriangleInitAllocQ15` は、`ipps.h` ファイルで宣言される。この関数は、メモリを割り当て、指定された周波数 *rFreqQ15*、位相 *phaseQ15*、非対称性 *asymQ15*、大きさ *magn* を持つ *pTriangleState* トライアングル・ジェネレータ構造体を初期化する。
Q15 形式のデータは、相対周波数値が [0, 0.5)、位相が [0, 2π)、非対称性が (-π, π) の範囲にある対応する浮動小数点データ・タイプに変換される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。*pTriandleState* ポインタが `NULL`。
`ippStsTriangleMagnErr`	エラー。*magn* がゼロ以下。
`ippStsTriangleFreqErr`	エラー。*rFreqQ15* が負、または 16383 より大きい。
`ippStsTrianglePhaseErr`	エラー。*phaseQ15* 値が負、または205886 より大きい。
`ippStsTriangleAsymErr`	エラー。*asymQ15* 値が -102943 よりも小さい、または 102943 よりも大きい。

TriangleFree

関数 ippsTriangleInitAlloc によって
割り当てられたメモリを解放する。

```
IppStatus ippsTriangleFree(IppTriangleState_16s* pTriangleState);
```

引数

pTriangleState	トライアングル・ジェネレータ指定構造体へのポインタ。

説明

関数 ippsTriangleFree は、ipps.h ファイルで宣言される。この関数は、ippsippsTriangleInitAllocQ15 により生成された構造体に関連付けられているメモリをすべて解放して、トライアングル・ジェネレータ・ステートをクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pTriangleState* ポインタが NULL。

TriangleGetStateSizeQ15

トライアングル・ジェネレータ構造体の
長さを計算する。

```
IppStatus ippsTriangleGetStateSizeQ15_16s(int* pTriangleStateSize);
```

引数

pTriangleStateSize	ジェネレータ指定構造体のバイト単位で計算されたサイズ値へのポインタ。

説明

関数 ippsTriangleGetStateSizeQ15 は、ipps.h ファイルで宣言される。この関数は、トーン・ジェネレータ構造体の *pTriangleStateSize* の長さ（バイト単位）を計算する。関数 ippsTriangleGetStateSizeQ15 は、ippsippsTriangleInitQ15 関数を呼び出す前に呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pTriangleStateSize* ポインタが NULL。

TriangleInitQ15

トライアングル・ジェネレータ指定構造体を初期化する。

```
IppStatus ippsTriangleInitQ15_16s(IppTriangleState_16s* pTriangleState,
      Ipp16s magn, Ipp16s rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

引数

pTriangleState	トライアングル・ジェネレータ指定構造体へのポインタ。
magn	トーンの大きさ。すなわち、波形の最大値を指定する。
rFreqQ15	Q0.15 形式のサンプリング周波数に対するトーンの周波数。値は[0,16383]の範囲内に収まっていなければならない。
phaseQ15	Q16.15　形式のコサイン波に対するトーンの位相。値は [0, 205886] の範囲内に収まっていなければならない。
asymQ15	Q16.15 形式のトライアングルの非対称性 *h*。値は [-102943, 102943] の範囲内に収まっていなければならない。*h*=0 の場合、トライアングルは対称になり、トーンの直接相似になる。

説明

関数 ippsTriangleInitQ15 は、ipps.h ファイルで宣言される。この関数は、指定された大きさ *magn*、周波数 *rFreqQ15*、位相 *phaseQ15*、非対称性 *asymQ15* を持つ *pTriangleState* トライアングル・ジェネレータ構造体を初期化する。この構造体は ippsippsTriangleGetStateSizeQ15 関数により求められるサイズの外

部バッファに割り当てられる。

Q15 形式のデータは、相対周波数値が [0, 0.5)、位相が [0, 2π)、非対称性が (-π, π) の範囲にある対応する浮動小数点データ・タイプに変換される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。*pTriandleState* ポインタが NULL。
`ippStsTriangleMagnErr`	エラー。*magn* がゼロ以下。
`ippStsTriangleFreqErr`	エラー。*rFreqQ15* が負、または 16383 より大きい。
`ippStsTrianglePhaseErr`	エラー。*phaseQ15* 値が負、または205886 より大きい。
`ippStsTriangleAsymErr`	エラー。*asymQ15* 値が -102943 よりも小さい、または 102943 よりも大きい。

TriangleQ15

トライアングル・ジェネレータ構造体で指定された周波数、位相、大きさを持つトライアングルを生成する。

```
IppStatus ippsTriangleQ15_16s(Ipp16s* pDst, int len,
     IppTriangleState_16s* pTriangleState);
```

引数

pDst	生成されたサンプルを格納する配列へのポインタ。
len	計算するサンプルの数。
pTriangleState	トライアングル・ジェネレータ指定構造体へのポインタ。

説明

関数 `ippsTriangleQ15` は、`ipps.h` ファイルで宣言される。この関数は、前に作成された *pTriangleState* 構造体で指定された周波数、位相、大きさ、非対称性の各パラメータを持つトライアングルを生成する。この関数は、*len* 個のトライアングル・サンプルを計算し、それらを配列 *pDst* に格納する。生成された値、$x[n]$ は実数のトライアングルの計算に用いられた `ippsTriangle_Direct` 関数と同じ式で計算される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタまたは *pSeed* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Triangle_Direct

特定の周波数、位相、大きさを持つ
トライアングルを生成する。

```
IppStatus ippsTriangle_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn,
      float rFreq, float asym, float* pPhase);
IppStatus ippsTriangle_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn,
      float rFreq, float asym, float* pPhase);
IppStatus ippsTriangle_Direct_32f(Ipp32f* pDst, int len, float magn,
      float rFreq, float asym, float* pPhase);
IppStatus ippsTriangle_Direct_32fc(Ipp32fc* pDst, int len, float magn,
      float rFreq, float asym, float* pPhase);
IppStatus ippsTriangle_Direct_64f(Ipp64f* pDst, int len, double magn,
      double rFreq, double asym, double* pPhase);
IppStatus ippsTriangle_Direct_64fc(Ipp64fc* pDst, int len, double magn,
      double rFreq, double asym, double* pPhase);
```

引数

rFreq	サンプリング周波数に対するトライアングルの周波数。値は [0.0, 0.5) の範囲内に収まっていなければならない。
pPhase	余弦相似三角波に対するトライアングルの位相へのポインタ。値は [0.0, 2π) の範囲内に収まっていなければならない。戻り値は、次の連続するデータ・ブロックの計算に使用できる。
magn	トライアングルの大きさ。すなわち、波形の最大値を指定する。
asym	トライアングルの非対称性 h。値は [-π, π) の範囲内に収まっていなければならない。h=0 の場合、トライアングルは対称になり、トーンの直接相似になる。
pDst	サンプルを格納する配列へのポインタ。

len 計算するサンプルの数。

説明

関数 ippsTriangle は、ipps.h ファイルで宣言される。この関数は、指定された周波数 *rFreq*、*pPhase* で指示された位相、大きさ *magn* を持つトライアングルを生成する。この関数は、*len* 個のトライアングル・サンプルを計算し、それらを配列 *pDst* に格納する。実数のトライアングルの場合、x[*n*] は次のように定義される。

$$x[n] = magn \cdot \mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase),\quad n = 0, 1, 2, \ldots$$

複素数のトライアングルの場合、x[*n*] は次のように定義される。

$$x[n] = magn \cdot (\mathbf{ct}_h(2\pi \cdot rFreq \cdot n + phase) + j \cdot \mathbf{st}_h(2\pi \cdot rFreq \cdot n + phase)),\quad n = 0, 1, 2, \ldots$$

関数 $\mathbf{ct}_h$ and $\mathbf{st}_h$ の定義については、4-14 ページの「アプリケーション・ノート」を参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタまたは *pPhase* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsTrnglMagnErr	エラー。*magn* がゼロ以下。
ippStsTrnglFreqErr	エラー。*rFreq* が負、または 0.5 以上。
ippStsTrnglPhaseErr	エラー。*pPhase* の値が負、または IPP_2PI 以上。
ippStsTrnglAsymErr	エラー。*asym* が -IPP_PI より小さい、または IPP_PI 以上。

TriangleQ15_Direct

特定の周波数、位相、大きさを持つ
トライアングルを生成する。

```
IppStatus ippsTriangleQ15_Direct_16s(Ipp16s* pDst, int len,
    Ipp16s magn, Ipp16s rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

引数

pDst	サンプルを格納する配列へのポインタ。
magn	トーンの大きさ。すなわち、波形の最大値を指定する。
rFreqQ15	Q0.15 形式のサンプリング周波数に対するトーンの周波数。値は [0, 16383] の範囲内に収まっていなければならない。
phaseQ15	Q16.15 形式のコサイン波に対するトーンの位相。値は [0, 205886] の範囲内に収まっていなければならない。
asymQ15	Q16.15 形式のトライアングルの非対称性 *h*。値は [-102943, 102943] の範囲内に収まっていなければならない。*h*=0 の場合、トライアングルは対称になり、トーンの直接相似になる。

説明

関数 ippsTriangleQ15_Direct は、ipps.h ファイルで宣言される。この関数は、指定された大きさ magn、周波数 *rFreqQ15*、位相 *pPhaseQ15*、非対称性 *asymQ15* を持つ *pTriangleState* トライアングルを生成する。Q15 形式のデータは、相対周波数値が [0, 0.5)、位相が [0, 2π)、非対称性が [-π, π) の範囲にある対応する浮動小数点データ・タイプに変換される。この関数は、*len* 個のトーン・サンプルを計算し、それらを配列 *pDst* に格納する。生成された値、*x*[*n*] は実数のトライアングルの計算に用いられた ippsTriangle_Direct 関数と同じ式で計算される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsToneMagnErr	エラー。*magn* がゼロ以下。
ippStsToneFreqErr	エラー。*rFreqQ15* が負、または 16383 より大きい。
ippStsTonePhaseErr	エラー。*phaseQ15* 値が負、または205886 より大きい。
ippStsTriangleAsymErr	エラー。*asymQ15* 値が -102943 よりも小さい、または 102943 よりも大きい。

一様分布関数

この項では、一様分布を持つ疑似ランダム・サンプルを生成する関数について説明する。

RandUniformInitAlloc

メモリを割り当て、ノイズ・ジェネレータを一様分布で初期化する。

```
IppStatus ippsRandUniformInitAlloc_8u(IppsRandUniState_8u**
      pRandUniState, Ipp8u low, Ipp8u high, unsigned int seed);
IppStatus ippsRandUniformInitAlloc_16s(IppsRandUniState_16s**
      pRandUniState, Ipp16s low, Ipp16s high, unsigned int seed);
IppStatus ippsRandUniformInitAlloc_32f(IppsRandUniState_32f**
      pRandUniState, Ipp32f low, Ipp32f high, unsigned int seed);
```

引数

pRandUniState	ノイズ・ジェネレータのパラメータを保持する構造体へのポインタ。
low	一様分布範囲の下限。
high	一様分布範囲の上限。
seed	疑似乱数生成アルゴリズムが使用するシード値。

説明

関数 ippsRandUniformInitAlloc は、ipps.h ファイルで宣言される。この関数は、メモリを割り当てて、疑似ランダム・ジェネレータ・ステート *pRandUniState* を初期化する。一様分布範囲は、下限 *low* と上限 *high* によって指定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pRandUniState* ポインタが NULL。
ippStsMemAllocErr	エラー。演算用のメモリが不足している。

RandUniformFree

一様分布ジェネレータ・ステートを終了する。

```
IppStatus ippsRandUniformFree_8u(IppsRandUniState_8u*
pRandUniState);
IppStatus ippsRandUniformFree_16s(IppsRandUniState_16s*
pRandUniState);
IppStatus ippsRandUniformFree_32f(IppsRandUniState_32f*
pRandUniState);
```

引数

pRandUniState	ノイズ・ジェネレータのパラメータを保持する構造体へのポインタ。

説明

関数 ippsRandUniformFree は、ipps.h ファイルで宣言される。この関数は、*ippsRandUniInitAlloc* 関数によって割り当てられたすべてのメモリを解放して、ノイズ・ジェネレータ・ステート pRandUniState を終了する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pRandUniState* ポインタが NULL。
ippStsContextMatchErr	エラー。ステート識別子が正しくない。

RandUniformInit

ノイズ・ジェネレータを一様分布で
初期化する。

```
IppStatus ippsRandUniformInit_16s(IppsRandUniState_16s* pRandUniState,
     Ipp16s low, Ipp16s high, unsigned int seed);
```

引数

pRandUniState	ノイズ・ジェネレータのパラメータを保持する構造体へのポインタ。
low	一様分布範囲の下限。
high	一様分布範囲の上限。
seed	疑似乱数生成アルゴリズムが使用するシード値。

説明

関数 ippsRandUniformInit は、ipps.h ファイルで宣言される。この関数は、外部バッファの疑似ランダム・ジェネレータ・ステート *pRandUniState* を初期化する。このバッファのサイズは事前に ippsRandUniformGetSize 関数を呼び出すことにより計算されている。一様分布範囲は、下限 *low* と上限 *high* によって指定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pRandUniState* ポインタが NULL。
ippStsMemAllocErr	エラー。演算用のメモリが不足している。

RandUniformGetSize

一様分布ジェネレータ構造体の長さを計算する。

```
IppStatus ippsRandUniformGetSize_16s(int* pRandUniStateSize))
```

引数

pRandUniStateSize	ジェネレータ指定構造体のバイト単位で計算されたサイズ値へのポインタ。

説明

関数 ippsRandUniformGetSize は、ipps.h ファイルで宣言される。この関数は、一様分布ジェネレータ構造体の *pRandUniStateSize* の長さ（バイト単位）を計算する。関数 ippsRandUniformGetSize は関数 ippsRandUniformInit よりも前に呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pRandUniStateSize* ポインタが NULL。

RandUnifrom

一様分布を持つ疑似ランダム・サンプルを
生成する。

```
IppStatus ippsRandUniform_8u(Ipp8u* pDst,  int len,
      IppsRandUniState_8u* pRandUniState);
IppStatus ippsRandUniform_16s(Ipp16s* pDst, int len,
      IppsRandUniState_16s* pRandUniState);
IppStatus ippsRandUniform_32f(Ipp32f* pDst, int len,
      IppsRandUniState_32f* pRandUniState);
```

引数

pDst	サンプルを格納する配列へのポインタ。
len	計算するサンプルの数。
pRandUniState	ノイズ・ジェネレータのパラメータを保持する構造体へのポインタ。

説明

関数 ippsRandUniform は、ipps.h ファイルで宣言される。この関数は、一様分布を持つ *len* 個の疑似ランダム・サンプルを生成し、それらを配列 *pDst* に格納する。ジェネレータの初期パラメータは、ジェネレータ・ステート構造体 *pRandUniState* 内で設定される。
ippsRandUniform を呼び出す前に、ippsRandUniformInitAlloc 関数を呼び出して、ジェネレータ・ステートを初期化しなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pRandUniState* が NULL。
ippStsContextMatchErr	エラー。ステート識別子が正しくない。

RandUniform_Direct

一様分布を持つ疑似ランダム・サンプルを
直接モードで生成する。

```
IppStatus ippsRandUniform_Direct_16s(Ipp16s* pDst, int len, Ipp16s
      low, Ipp16s high, unsigned int* pSeed);
IppStatus ippsRandUniform_Direct_32f(Ipp32f* pDst, int len, Ipp32f
      low, Ipp32f high, unsigned int* pSeed);
IppStatus ippsRandUniform_Direct_64f(Ipp64f* pDst, int len, Ipp64f
      low, Ipp64f high, unsigned int* pSeed);
```

引数

pSeed	疑似乱数生成アルゴリズムが使用するシード値へのポインタ。
low	一様分布範囲の下限。
high	一様分布範囲の上限。
pDst	サンプルを格納する配列へのポインタ。
len	計算するサンプルの数。

説明

関数 ippsRandUniform_Direct は、ipps.h ファイルで宣言される。この関数は、一様分布を持つ *len* 個の疑似ランダム・サンプルを生成し、それらを配列 *pDst* に格納する。この関数では、事前にジェネレータ・ステート構造体を初期化する必要はない。疑似乱数ジェネレータのすべてのパラメータは、関数内で直接に設定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSeed* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

ガウス分布関数

この項では、ガウス分布を持つ疑似ランダム・サンプルを生成する関数について説明する。

RandGaussInitAlloc

メモリを割り当て、ノイズ・ジェネレータをガウス分布で初期化する。

```
IppStatus ippsRandGaussInitAlloc_8u(IppsRandGaussState_8u**
      pRandGaussState, Ipp8u mean, Ipp8u stdDev, unsigned int seed);
IppStatus ippsRandGaussInitAlloc_16s(IppsRandGaussState_16s**
      pRandGaussState,Ipp16smean,Ipp16sstdDev,unsignedintseed);
IppStatus ippsRandGaussInitAlloc_32f(IppsRandGaussState_32f**
      pRandGaussState,Ipp32fmean,Ipp32fstdDev,unsignedintseed);
```

引数

pRandGaussState	ノイズ・ジェネレータのパラメータを保持する構造体へのポインタ。
mean	ガウス分布の平均。
stdDev	ガウス分布の標準偏差。
seed	疑似乱数生成アルゴリズムが使用するシード値。

説明

関数 ippsRandGaussInitAlloc は、ipps.h ファイルで宣言される。この関数は、メモリを割り当てて、疑似ランダム・ジェネレータ・ステート構造体 *pRandGaussState* を初期化する。この構造体は、*mean*、*stdDev*、*seed* の値で指定される、必要なノイズ・ジェネレータのパラメータを保持する。

戻り値

ppStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pRandGaussState* ポインタが NULL。
ippStsMemAllocErr	エラー。演算用のメモリが不足している。

RandGaussFree

ガウス分布ジェネレータ・ステートを
終了する。

```
IppStatus ippsRandGaussFree_8u(IppsRandGaussState_8u*
  pRandGaussState);
IppStatus ippsRandGaussFree_16s(IppsRandGaussState_16s*
  pRandGaussState);
IppStatus ippsRandGaussFree_32f(IppsRandGaussState_32f*
  pRandGaussState);
```

引数

pRandGaussState	ノイズ・ジェネレータのパラメータを保持する構造体へのポインタ。

説明

関数 ippsRandGaussFree は、ipps.h ファイルで宣言される。この関数は、*ippsRandGaussInitAlloc* 関数によって割り当てられたすべてのメモリを解放して、ノイズ・ジェネレータ・ステート pRandGaussState を終了する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pRandGaussState* ポインタが NULL。
ippStsContextMatchErr	エラー。ステート識別子が正しくない。

RandGaussGetSize

ガウス分布ジェネレータ構造体の長さを
計算する。

```
IppStatus ippsRandGaussGetSize_16s(int* pRandGaussStateSize);
```

引数

pRandGaussStateSize	ジェネレータ指定構造体のバイト単位で計算されたサイズ値へのポインタ。

説明

関数 ippsRandGaussGetSize は、ipps.h ファイルで宣言される。この関数は、一様分布ジェネレータ構造体の *pRandGaussStateSize* の長さ（バイト単位）を計算する。関数 ippsRandGaussGetSize は、関数 ippsRandGaussInit を呼び出す前に呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pRandGaussStateSize* ポインタが NULL。

RandGaussInit

ノイズ・ジェネレータをガウス分布で初期化する。

```
IppStatus ippsRandGaussInit_16s(IppsRandGaussState_16s*
    pRandGaussState,Ipp16smean,Ipp16sstdDev,unsignedintseed);
```

引数

pRandGaussState	ノイズ・ジェネレータのパラメータを保持する構造体へのポインタ。
mean	ガウス分布の平均。
stdDev	ガウス分布の標準偏差。
seed	疑似乱数生成アルゴリズムが使用するシード値。

説明

関数 ippsRandGaussInit は、ipps.h ファイルで宣言される。この関数は、外部バッファの疑似ランダム・ジェネレータ・ステート構造体 *pRandGaussState* を初期化する。このバッファのサイズは事前に ippsRandGaussGetSize 関数を呼び出すことにより計算されている。この構造体は、*mean*、*stdDev*、*seed* の値で指定される、必要なノイズ・ジェネレータのパラメータを保持する。

戻り値

ppStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pRandGaussState* ポインタが NULL。
ippStsMemAllocErr	エラー。演算用のメモリが不足している。

RandGauss

ガウス分布を持つ疑似ランダム・サンプル
を生成する。

```
IppStatus ippsRandGauss_8u(Ipp8u* pDst,  int len,
      IppsRandGaussState_8u*  pRandGaussState);
IppStatus ippsRandGauss_16s(Ipp16s* pDst, int len,
      IppsRandGaussState_16s* pRandGaussState);
IppStatus ippsRandGauss_32f(Ipp32f* pDst, int len,
      IppsRandGaussState_32f* pRandGaussState);
```

引数

pDst	サンプルを格納する配列へのポインタ。
len	計算するサンプルの数。
pRandGaussState	ノイズ・ジェネレータのパラメータを保持する構造体へのポインタ。

説明

関数 ippsRandGauss は、ipps.h ファイルで宣言される。この関数は、ガウス分布を持つ *len* 個の疑似ランダム・サンプルを生成し、それらを配列 *pDst* に格納する。ジェネレータの初期パラメータは、ジェネレータ・ステート構造体 *pRandGaussState* 内で設定される。
ippsRandGauss を呼び出す前に、ippsRandGaussInitAlloc 関数を呼び出して、ジェネレータ・ステートを初期化しなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pRandGaussState* ポインタが NULL。
ippStsContextMatchErr	エラー。ステート識別子が正しくない。

RandGauss_Direct

ガウス分布を持つ疑似ランダム・サンプルを直接モードで生成する。

```
IppStatusippsRandGauss_Direct_16s(Ipp16s*pDst,intlen,Ipp16smean,
      Ipp16s stdev, unsigned int* pSeed);
IppStatusippsRandGauss_Direct_32f(Ipp32f*pDst,intlen,Ipp32fmean,
      Ipp32f stdev, unsigned int* pSeed);
IppStatusippsRandGauss_Direct_64f(Ipp64f*pDst,intlen,Ipp64fmean,
      Ipp64f stdev, unsigned int* pSeed);
```

引数

pDst	サンプルを格納する配列へのポインタ。
pSeed	疑似乱数生成アルゴリズムが使用するシード値へのポインタ。
len	計算するサンプルの数。
mean	ガウス分布の平均。
stdev	ガウス分布の標準偏差。

説明

関数 ippsRandGauss は、ipps.h ファイルで宣言される。この関数は、ガウス分布を持つ *len* 個の疑似ランダム・サンプルを生成し、それらを配列 *pDst* に格納する。この関数では、事前にジェネレータ・ステート構造体を初期化する必要はない。疑似乱数ジェネレータのすべてのパラメータは、関数内で直接に設定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSeed* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

特殊ベクトル関数

この項で説明する関数は、各種の信号処理関数を適用した結果を調べるためのテスト信号として使用できる、特殊なベクトルを生成する。

VectorJaehne

Jaehne ベクトルを生成する。

```
IppStatus ippsVectorJaehne_8u(Ipp8u* pDst, int len, Ipp8u magn);
IppStatus ippsVectorJaehne_8s(Ipp8s* pDst, int len, Ipp8s magn);
IppStatus ippsVectorJaehne_16u(Ipp16u* pDst, int len, Ipp16u magn);
IppStatus ippsVectorJaehne_16s(Ipp16s* pDst, int len, Ipp16s magn);
IppStatus ippsVectorJaehne_32u(Ipp32u* pDst, int len, Ipp32u magn);
IppStatus ippsVectorJaehne_32s(Ipp32s* pDst, int len, Ipp32s magn);
IppStatus ippsVectorJaehne_32f(Ipp32f* pDst, int len, Ipp32f magn);
IppStatus ippsVectorJaehne_64f(Ipp64f* pDst, int len, Ipp64f magn);
```

引数

pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
magn	生成する信号の大きさ。

説明

関数 `ippsVectorJaehne` は、`ipps.h` ファイルで宣言される。この関数は Jaehne ベクトルを生成し、結果を *pDst* に格納する。大きさ *magn* には、正の値を指定する。この関数は、可変周波数を持つサイン・カーブを生成する。計算は、次のように行われる。

$$pDst[n] = magn \cdot \sin\left(\frac{0.5\pi n^2}{len}\right), \quad 0 \le n < len$$

例 4-4 は、関数 ippsVectorJaehne の使用例を示している。

例 4-4 ippsVectorJaehne 関数の使用例

```
IppStatus Jaehne (void) {
    Ipp16s buf[100] ;
    return ippsVectorJaehne_16s ( buf, 100, 255 );
}
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsJaehneErr	エラー。*magn* が負。

VectorRamp

ランプ・ベクトルを生成する。

```
IppStatus ippsVectorRamp_8u(Ipp8u* pDst, int len, float offset,
    float slope);
IppStatus ippsVectorRamp_8s(Ipp8s* pDst, int len, float offset,
    float slope);
IppStatus ippsVectorRamp_16u(Ipp16u* pDst, int len, float offset,
    float slope);
IppStatus ippsVectorRamp_16s(Ipp16s* pDst, int len, float offset,
    float slope);
IppStatus ippsVectorRamp_32u(Ipp32u* pDst, int len, float offset,
    float slope);
IppStatus ippsVectorRamp_32s(Ipp32s* pDst, int len, float offset,
    float slope);
IppStatus ippsVectorRamp_32f(Ipp32f* pDst, int len, float offset,
    float slope);
IppStatus ippsVectorRamp_64f(Ipp64f* pDst, int len, float offset,
    float slope);
```


intel®

引数

pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
offset	オフセットの値。
slope	勾配係数。

説明

関数 ippsVectorRamp は、ipps.h ファイルで宣言される。この関数は、ランプ・ベクトルを生成し、結果を *pDst* に格納する。デスティネーション・ベクトルの要素は、次の公式に従って計算される。

pDst[n] = *offset* + *slope***n*, $0 \leq n < len$.

ただし、すべての関数タイプで、線形変換係数 *offset* と *slope* は浮動小数点値を持つ。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

基本的なベクトル関数 5

本章では、論理演算、シフト演算、算術演算、変換演算、圧伸演算、窓（window）演算、統計演算を実行するインテル® IPP 関数をそれぞれ説明する。

このグループのすべての関数を表 5-1 に示す。

表 5-1　インテル® IPP 基本的なベクトル関数

関数の基本名	操作
論理関数とシフト関数	
AndC	スカラー値とベクトルの各要素のビット単位 AND を計算する。
And	2 つのベクトルのビット単位 AND を計算する。
OrC	スカラー値とベクトルの各要素のビット単位 OR を計算する。
Or	2 つのベクトルのビット単位 OR を計算する。
XorC	スカラー値とベクトルの各要素のビット単位 XOR を計算する。
Xor	2 つのベクトルのビット単位 XOR を計算する。
Not	ベクトル要素のビット単位 NOT を計算する。
LShiftC	ベクトル要素のビットを左にシフトする。
RShiftC	ベクトル要素のビットを右にシフトする。
算術関数	
AddC	ベクトルの各要素に定数値を加える。
Add	2 つのベクトルの要素を足し合わせる。
AddProduct	2 つのベクトルの内積をアキュムレータ・ベクトルに加算する。
MulC	ベクトルの各要素に定数値を掛ける。
Mul	2 つのベクトルの要素を掛け合わせる。
SubC	ベクトルの各要素から定数値を引く。
SubCRev	定数値からベクトルの各要素を引く。
Sub	2 つのベクトルの要素の差を求める。
DivC	ベクトルの各要素を定数値で割る。
DivCRev	定数値をベクトルの各要素で割る。
Div	2 つのベクトルの要素を除算する。
Abs	ベクトル要素の絶対値を計算する。
Sqr	ベクトルの各要素の 2 乗を計算する。

表 5-1 インテル® IPP 基本的なベクトル関数 （続き）

関数の基本名	操作
Sqrt	ベクトルの各要素の平方根を計算する。
Cubrt	ベクトルの各要素の立方根を計算する。
Exp	e をベクトルの各要素で累乗した値を計算する。
Ln	ベクトルの各要素の自然対数を計算する。
10Log10	ベクトルの各要素を十進法の対数で計算し、10 を掛ける
SumLn	ベクトルの各要素の自然対数を合計する。
Arctan	ベクトルの各要素の逆正接を計算する。
Normalize	オフセット演算と除算演算を使用して、実数ベクトルまたは複素数ベクトルの要素を正規化する。
変換関数	
SortAscend, SortDescend	ベクトルのすべての要素を並べ替える。
SwapBytes	ベクトルのバイト順序を逆にする。
Convert	ベクトルのデータ・タイプを変換し、その結果を 2 番目のベクトルに格納する。
Join	複数のベクトルの浮動小数点データを整数データに変換し、その結果を 1 つのベクトルに格納する。
Conj	ベクトルの共役複素数の値を 2 番目のベクトルに格納する（インプレース方式）。
ConjFlip	ベクトルの共役複素数を計算し、その結果を逆順で格納する。
Magnitude	複素数ベクトルの要素の大きさを計算する。
MagSquared	複素数ベクトルの要素の 2 乗の大きさを計算する。
Phase	複素数入力ベクトルの要素の位相角度を 2 番目のベクトルに返す。
PowerSpectr	複素数ベクトルのパワー・スペクトラムを計算する。
Real	複素数ベクトルの実数部を 2 番目のベクトルに返す。
Imag	複素数ベクトルの虚数部を 2 番目のベクトルに返す。
RealToCplx	2 つの実数ベクトルの実数部と虚数部から構築した複素数ベクトルを返す。
CplxToReal	複素数ベクトルの実数部と虚数部を 2 つのベクトルにそれぞれ返す。
Threshold Threshold_LT, Threshold_GT Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal	要素の値を *level* で制限しながら、ベクトルの要素に対するしきい値演算を実行する。
Threshold_LTInv	下限でベクトルの各要素の大きさを制限した後で、ベクトルの各要素の逆を計算する。
CartToPolar	複素数ベクトルの要素を極座標形式に変換する。
PolarToCart	実数部 / 虚数部の組からなる入力ベクトルを複素数形式から極座標形式に変換する。
MaxOrder	ベクトルの最大次数を計算する。

intel®

表 5-1 インテル® IPP 基本的なベクトル関数 （続き）

関数の基本名	操作
Preemphasize	単精度実数信号のプリエンファシスを計算する（インプレース方式）。
Flip	ベクトル内の要素の順序を逆にする。
FindNearestOne	指定された値に最も近いテーブルの要素を見つける。
FindNearest	指定されたベクトルの要素に最も近いテーブルの要素を見つける。
ビタビ・デコーダ関数	
GetVarPointDV	受け取った点に最も近い点に関する情報を配列に格納する。
CalcStatesDV	ビタビ・デコーダのステートを計算する。
BuildSymblTableDV4D	可能な 4D シンボルに関する情報を配列に格納する。
UpdatePathMetricsDV	最小パス・メトリックを持つステートを検索する。
窓（Windowing）関数	
WinBartlett	ベクトルに Bartlett（バートレット）窓関数を掛ける。
WinBlackman	ベクトルに Blackman（ブラックマン）窓関数を掛ける。
WinHamming	ベクトルに Hamming（ハミング）窓関数を掛ける。
WinHann	ベクトルに Hann（ハニング）窓関数を掛ける。
WinKaiser	ベクトルに Kaiser（カイザー）窓関数を掛ける。
統計関数	
Sum	ベクトルの要素の合計を計算する。
Max	ベクトルの最大値を返す。
MaxIndx	ベクトルの最大値と、最大要素のインデックスを返す。
Min	ベクトルの最小値を返す。
MinIndx	ベクトルの最小値と、最小要素のインデックスを返す。
MinMax	ベクトルの最大値と、最小値を返す。
MinMaxIndx	ベクトルの最大値と最小値、それに対応する要素のインデックスを返す。
Mean	ベクトルの平均値を計算する。
StdDev	ベクトルの標準偏差値を計算する。
Norm	ベクトルのノルム C、L1、または L2 を計算する。
NormDiff	2 つのベクトルの差のノルム C、L1、または L2 を計算する。
DotProd	2 つのベクトルの内積を計算する。
MaxEvery, MinEvery	2 つのベクトルの要素の各ペアの最大値または最小値を計算する。
サンプリング関数	
SampleUp	整数の係数でサンプリング・レートを概念的に上げて信号をアップ・サンプリングする。
SampleDown	整数の係数でサンプリング・レートを概念的に下げて信号をダウン・サンプリングする。

論理関数とシフト関数

この項では、ベクトルに対し論理演算またはシフト演算を実行するインテル® IPP（信号処理用）関数について説明する。論理関数とシフト関数は、整数引数に対してのみ定義される。

AND、OR、XOR の各バイナリ論理演算用には、次の関数を用意している。
ベクトル - スカラー演算用 : `AndC`、`OrC`、`XorC`
ベクトル - ベクトル演算用 : `And`、`Or`、`Xor`

AndC

スカラー値とベクトルの各要素のビット単位 AND を計算する。

```
IppStatus ippsAndC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
    int len);
IppStatus ippsAndC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst,
    int len);
IppStatus ippsAndC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst,
    int len);
IppStatus ippsAndC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsAndC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsAndC_32u_I(Ipp32u val, Ipp32u* pSrcDst, int len);
```

引数

val	入力スカラー値。
pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsAndC` は、`ipps.h` ファイルで宣言される。この関数は、スカラー値 *val* とベクトル *pSrc* の各要素に対してビット単位 AND を計算し、その結果を *pDst* に格納する。

関数 *ippsAndC* はインプレース演算で、スカラー値 *val* とベクトル *pSrcDst* の各要素に対してビット単位で AND 演算を行い、その結果を *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

And

2 つのベクトルのビット単位 AND を計算する。

```
IppStatus ippsAnd_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
    Ipp8u* pDst, int len);
IppStatus ippsAnd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
    Ipp16u* pDst, int len);
IppStatus ippsAnd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2,
    Ipp32u* pDst, int len);
IppStatus ippsAnd_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsAnd_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsAnd_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

引数

pSrc1, pSrc2	2 つのソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrc	インプレース演算用のソース・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsAnd は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc1* とベクトル *pSrc2* の対応する各要素に対してビット単位 AND を計算し、その結果をベクトル *pDst* に格納する。

関数 ippsAnd はインプレース演算で、ベクトル *pSrc* とベクトル *pSrcDst* の対応する各要素に対してビット単位で AND 演算を行い、その結果をベクトル *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

OrC

スカラー値とベクトルの各要素のビット単位 OR を計算する。

```
IppStatus ippsOrC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int
    len);
IppStatus ippsOrC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int
    len);
IppStatus ippsOrC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int
    len);
IppStatus ippsOrC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsOrC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsOrC_32u_I(Ipp16u val, Ipp32u* pSrcDst, int len);
```

引数

val	入力スカラー値。
pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsOrC は、ipps.h ファイルで宣言される。この関数は、スカラー値 *val* とベクトル *pSrc* の各要素に対してビット単位 OR を計算し、その結果を *pDst* に格納する。

関数 ippsOrC はインプレース演算で、スカラー値 *val* とベクトル *pSrcDst* の各要素に対してビット単位で OR 演算を行い、その結果を *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Or

2 つのベクトルのビット単位 OR を計算する。

```
IppStatus ippsOr_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
    Ipp8u* pDst, int len);
IppStatus ippsOr_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
    Ipp16u* pDst, int len);
IppStatus ippsOr_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2,
    Ipp32u* pDst, int len);
IppStatus ippsOr_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsOr_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsOr_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

引数

pSrc1, pSrc2	2 つのソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrc	インプレース演算用のソース・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsOr は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc1* とベクトル *pSrc2* の対応する各要素に対してビット単位 OR を計算し、その結果をベクトル *pDst* に格納する。

関数 ippsOr はインプレース演算で、ベクトル *pSrc* とベクトル *pSrcDst* の対応する各要素に対してビット単位で OR 演算を行い、その結果をベクトル *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

XorC

スカラー値とベクトルの各要素のビット単位 XOR を計算する。

```
IppStatus ippsXorC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int
    len);
IppStatus ippsXorC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst,
    int len);
IppStatus ippsXorC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst,
    int len);
IppStatus ippsXorC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsXorC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsXorC_32u_I(Ipp32u val, Ipp32u* pSrcDst, int len);
```

引数

val	入力スカラー値。
pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsXorC は、ipps.h ファイルで宣言される。この関数は、スカラー値 *val* とベクトル *pSrc* の各要素に対してビット単位 XOR を計算し、その結果を *pDst* に格納する。

関数 ippsXorC はインプレース演算で、スカラー値 *val* とベクトル *pSrcDst* の各要素に対してビット単位で XOR 演算を行い、その結果を *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Xor

2 つのベクトルのビット単位 XOR を計算する。

```
IppStatus ippsXor_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
    Ipp8u* pDst, int len);
IppStatus ippsXor_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
    Ipp16u* pDst, int len);
IppStatus ippsXor_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2,
    Ipp32u* pDst, int len);
IppStatus ippsXor_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsXor_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsXor_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

引数

pSrc1, pSrc2	2 つのソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrc	インプレース演算用のソース・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsXor は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc1* とベクトル *pSrc2* の対応する各要素に対してビット単位 XOR を計算し、その結果をベクトル *pDst* に格納する。

関数 ippsXor はインプレース演算で、ベクトル *pSrc* とベクトル *pSrcDst* の対応する各要素に対してビット単位で XOR 演算を行い、その結果をベクトル *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Not

ベクトル要素のビット単位 NOT を計算する。

```
IppStatus ippsNot_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsNot_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsNot_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsNot_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsNot_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsNot_32u_I(Ipp32u* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsNot は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の対応する各要素に対してビット単位 NOT を計算し、その結果をベクトル *pDst* に格納する。

関数 ippsNot はインプレース演算で、ベクトル *pSrcDst* の対応する各要素に対してビット単位で NOT 演算を行い、その結果をベクトル *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

LShiftC

ベクトル要素のビットを左にシフトする。

```
IppStatus ippsLShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int
    len);
IppStatus ippsLShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst,
    int len);
IppStatus ippsLShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst,
    int len);
IppStatus ippsLShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst,
    int len);
IppStatus ippsLShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
IppStatus ippsLShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippsLShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippsLShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);
```

引数

val	この関数がベクトル *pSrc* またはベクトル *pSrcDst* の各要素をシフトするビット数。
pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsLShiftC は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素を *val* ビットだけ左にシフトし、その結果を *pDst* に格納する。

関数 ippsLShiftC はインプレース演算で、ベクトル *pSrcDst* の各要素を *val* ビットだけ左にシフトし、その結果を *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

RShiftC

ベクトル要素のビットを右にシフトする。

```
IppStatus ippsRShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int
    len);
IppStatus ippsRShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst,
    int len);
IppStatus ippsRShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst,
    int len);
IppStatus ippsRShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst,
    int len);
IppStatus ippsRShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
IppStatus ippsRShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippsRShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippsRShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);
```

引数

val	この関数がベクトル *pSrc* またはベクトル *pSrcDst* の各要素をシフトするビット数。
pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsRShiftC` は、`ipps.h` ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素を *val* ビットだけ右にシフトし、その結果を *pDst* に格納する。

関数 `ippsRShiftC` はインプレース演算で、ベクトル *pSrcDst* の各要素を *val* ビットだけ右にシフトし、その結果を *pSrcDst* に格納する。

算術シフトは符号付きデータに適用され、論理シフトは符号なしデータに適用される点に注意しなければならない。

例 5-1 は、飽和演算における論理関数とシフト関数の使用例を示している。データは、[0...255] の範囲の符号なし文字型に変換される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
`ippStsSizeErr`	エラー。*len* がゼロ以下。

例 5-1　論理関数とシフト関数の使用例

```
void saturate(void) {
      Ipp16s x[8] = {1000, -257, 127, 4, 5, 0, 7, 8}, lo[8], hi[8];
      IppStatus status = ippsNot_16u((Ipp16u*)x, (Ipp16u*)lo, 8);
      ippsRShiftC_16s_I(15, lo, 8);
      ippsCopy_16s(x, hi, 8);
      ippsSubCRev_16s_ISfs(255, hi, 8, 0);
      ippsRShiftC_16s_I(15, hi, 8);
      ippsAnd_16u_I((Ipp16u*)lo, (Ipp16u*)x, 8);
      ippsOr_16u_I((Ipp16u*)hi, (Ipp16u*)x, 8);
      ippsAndC_16u_I(255, (Ipp16u*)x, 8);
      printf_16s("saturate =", x, 8, status);
}
Output:
      saturate =  255 0 127 4 5 0 7 8
```

算術関数

この項では、ベクトル上でベクトル算術演算を実行するインテル® IPP（信号処理用）関数について説明する。算術関数には、ベクトル間の基本的な算術演算を要素単位で実行するものだけではなく、ベクトル要素の絶対値、2 乗、平方根、自然対数、指数の計算など、より複雑な計算を行うものもある。

インテル IPP ソフトウェアでは、各関数ごとに、2 つのバージョンを用意している。1 つのバージョンは演算をインプレースで実行し、もう 1 つのバージョンは演算の結果を異なるデスティネーション・ベクトルに格納する（つまり、アウトオブプレース演算を実行する）。

AddC

ベクトルの各要素に定数値を加える。

```
IppStatus ippsAddC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
    int len);
IppStatus ippsAddC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
    int len);
IppStatus ippsAddC_32fc(const Ipp32fc* pSrc, Ipp32fc val,Ipp32fc* pDst,
    int len);
IppStatus ippsAddC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc*
    pDst, int len);
IppStatus ippsAddC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsAddC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsAddC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsAddC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsAddC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsAddC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsAddC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsAddC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s*
    pDst, int len, int scaleFactor);
IppStatus ippsAddC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc*
    pDst, int len, int scaleFactor);
IppStatus ippsAddC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc*
    pDst, int len, int scaleFactor);
```

```
IppStatus ippsAddC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsAddC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAddC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsAddC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsAddC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
    int scaleFactor);
```

引数

val	ベクトル *pSrc* またはベクトル *pSrcDst* の各要素を増分するために使用するスカラー値。
pSrc	ソース・ベクトルへのポインタ。
pDst	加算 *pSrc[n]* + *val* の結果を格納するデスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsAddC は、ipps.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc* の各要素に値 *val* を加え、その結果をデスティネーション・ベクトル *pDst* に格納する。

関数 ippsAddC はインプレース演算で、ベクトル *pSrcDst* の各要素に値 *val* を加え、その結果を *pSrcDst* に格納する。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値に対しスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Add

2 つのベクトルの要素を足し合わせる。

```
IppStatus ippsAdd_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
IppStatus ippsAdd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
    Ipp16u* pDst, int len);
IppStatus ippsAdd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2,
    Ipp32u* pDst, int len);
IppStatus ippsAdd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsAdd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsAdd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
    Ipp32fc* pDst, int len);
IppStatus ippsAdd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
    Ipp64fc* pDst, int len);
IppStatus ippsAdd_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
    Ipp16u* pDst, int len);
IppStatus ippsAdd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsAdd_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsAdd_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsAdd_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsAdd_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int
    len);
IppStatus ippsAdd_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int
    len);
IppStatus ippsAdd_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u*
    pDst, int len, int scaleFactor);
IppStatus ippsAdd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsAdd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
    Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsAdd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc*
    pSrc2,Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsAdd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
    Ipp32sc* pDst, int len, int scaleFactor);
```

```
IppStatus ippsAdd_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsAdd_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int
    len, int scaleFactor);
IppStatus ippsAdd_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int
    len,  int scaleFactor);
IppStatus ippsAdd_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int
    len, int scaleFactor);
IppStatus ippsAdd_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int
    len, int scaleFactor)
```

引数

pSrc1、*pSrc2*	要素を足し合わせるベクトルへのポインタ。
pDst	加算 *pSrc2[n]+pSrc1[n]* の結果を格納するデスティネーション・ベクトルへのポインタ。
pSrc	*pSrcDst* の要素にインプレースで要素を加えるソース・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsAdd は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc1* の要素とベクトル *pSrc2* の要素を加算し、その結果を *pDst* に格納する。

関数 ippsAdd はインプレース演算で、ベクトル *pSrc* の要素とベクトル *pSrcDst* の要素を加算し、その結果を *pSrcDst* に格納する。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値に対しスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和される。

例 5-2 は、スケーリング演算の使用により、大きな数値を加えてもオーバーフローが発生しない様子を示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-2　ippsAdd 関数の使用例

```
IppStatus add(void) {
        Ipp16s x[4] = {-1, 32767, 2, 30000};
        IppStatus st = ippsAdd_16s_ISfs(x, x, 4, 1);
        printf_16s("add =", x, 4, st);
        return st;
}
Output:
        add = -1 32767 2 30000
```

AddProduct

2 つのベクトルの内積をアキュムレータ・ベクトルに足す。

```
IppStatus ippsAddProduct_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pSrcDst, int len);
IppStatus ippsAddProduct_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pSrcDst, int len);
IppStatus ippsAddProduct_32fc(const Ipp32fc* pSrc1, const Ipp32fc*
    pSrc2, Ipp32fc* pSrcDst, int len);
IppStatus ippsAddProduct_64fc(const Ipp64fc* pSrc1, const Ipp64fc*
    pSrc2, Ipp64fc* pSrcDst, int len);
IppStatus ippsAddProduct_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddProduct_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s*
    pSrc2, Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsAddProduct_16s32s_Sfs(const Ipp16s* pSrc1,
    const Ipp16s* pSrc2, Ipp32s* pSrcDst, int len, int scaleFactor);
```

引数

pSrc1, pSrc2	ソース・ベクトルへのポインタ。
pSrcDst	デスティネーション・アキュムレータ・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	「整数のスケーリング」を参照。

intel®

説明

関数 ippsAddProduct は、ipps.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc1* の各要素をそれに対応するベクトル *pSrc2* の要素で掛けて、結果をそれに対応するアキュムレータ・ベクトル *pSrcDst* の各要素に足す。

$pSrcDst[n] = pSrcDst[n] + pSrc1[n] * pSrc2[n], \quad 0 \le n < len$

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値のスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

MulC

ベクトルの各要素に定数値を掛ける。

```
IppStatus ippsMulC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
    int len);
IppStatus ippsMulC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
    int len);
IppStatus ippsMulC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc*
    pDst, int len);
IppStatus ippsMulC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc*
    pDst, int len);
IppStatus ippsMulC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsMulC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsMulC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsMulC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsMulC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsMulC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsMulC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsMulC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s*
    pDst, int len, int scaleFactor);
```

```
IppStatus ippsMulC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc*
    pDst, int len, int scaleFactor);
IppStatus ippsMulC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc*
    pDst, int len, int scaleFactor);
IppStatus ippsMulC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsMulC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsMulC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsMulC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsMulC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsMulC_32f16s_Sfs(const Ipp32f* pSrc, Ipp32f val, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsMulC_Low_32f16s(const Ipp32f* pSrc, Ipp32f val, Ipp16s*
    pDst, int len);
```

引数

val	ベクトル *pSrc* またはベクトル *pSrcDst* の各要素に掛けるスカラー値。
pSrc	ソース・ベクトルへのポインタ。
pDst	乗算 *pSrc[n] * val* の結果を格納するデスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsMulC は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素に値 *val* を掛け、その結果を *pDst* に格納する。

関数 ippsMulC はインプレース演算で、ベクトル *pSrcDst* の各要素に値 *val* を掛け、その結果を *pSrcDst* に格納する。

Low サフィックスの付いた関数タイプでは、積 *pSrc*val* の各値が Ipp32s データ・タイプの範囲を超えてはならない。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値に対しスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Mul

2 つのベクトルの要素を掛け合わせる。

```
IppStatus ippsMul_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
IppStatus ippsMul_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsMul_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsMul_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
    Ipp32fc* pDst, int len);
IppStatus ippsMul_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
    Ipp64fc* pDst, int len);
IppStatus ippsMul_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
    Ipp16u* pDst, int len);
IppStatus ippsMul_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsMul_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMul_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMul_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsMul_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsMul_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsMul_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u*
    pDst, int len, int scaleFactor);
IppStatus ippsMul_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsMul_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
    Ipp32s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsMul_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
    Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsMul_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
    Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsMul_16u16s_Sfs(const Ipp16u* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsMul_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsMul_32s32sc_Sfs(const Ipp32s* pSrc1, const Ipp32sc* pSrc2,
    Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsMul_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsMul_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int
    len, int scaleFactor);
IppStatus ippsMul_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int
    len, int scaleFactor);
IppStatus ippsMul_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int
    len, int scaleFactor);
IppStatus ippsMul_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int
    len, int scaleFactor);
IppStatus ippsMul_32s32sc_ISfs(const Ipp32s* pSrc, Ipp32sc* pSrcDst,
    int len, int scaleFactor);
```

引数

pSrc1, pSrc2	要素が互いに掛け合わされるベクトルへのポインタ。
pDst	乗算 *pSrc1[n] * pSrc2[n]* の結果を格納するデスティネーション・ベクトルへのポインタ。
pSrc	*pSrcDst* の要素をインプレースに掛け合わされる要素を持つベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsMul は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc1* の要素とベクトル *pSrc2* の要素を掛け合わせ、その結果を *pDst* に格納する。

関数 ippsMul はインプレース演算で、ベクトル *pSrc* の要素とベクトル *pSrcDst* の要素を掛け合わせ、その結果を *pSrcDst* に格納する。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値に対しスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

SubC

ベクトルの各要素から定数値を引く。

```
IppStatus ippsSubC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
    int len);
IppStatus ippsSubC_32fc(const Ipp32fc* pSrc, Ipp32fc val,
    Ipp32fc* pDst, int len)
IppStatus ippsSubC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
    int len);
IppStatus ippsSubC_64fc(const Ipp64fc* pSrc, Ipp64fc val,
    Ipp64fc* pDst, int len);
IppStatus ippsSubC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
IppStatus ippsSubC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsSubC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsSubC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSubC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val,
    Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsSubC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val,
    Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsSubC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val,
    Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsSubC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int
    scaleFactor);
```

```
IppStatus ippsSubC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSubC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsSubC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
    int scaleFactor);
```

引数

val	ベクトル *pSrc* またはベクトル *pSrcDst* の各要素を減分するために使用するスカラー値。
pSrc	ソース・ベクトルへのポインタ。
pDst	減算 *pSrc[n] - val* の結果を格納するデスティネーション・ベクトルへのポインタ。
pSrcDst	値 *val* で減算し、結果がインプレース演算用の *pSrcDst* に格納される要素を持つベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsSubC は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素から値 *val* を引き、その結果を *pDst* に格納する。

関数 ippsSubC はインプレース演算で、ベクトル *pSrcDst* の各要素から値 *val* を引き、その結果を *pSrcDst* に格納する。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値に対しスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

intel®

SubCRev

定数値からベクトルの各要素を引く。

```
IppStatus ippsSubCRev_32f(const Ipp32f* pSrc, Ipp32f val,
    Ipp32f* pDst, int len);
IppStatus ippsSubCRev_64f(const Ipp64f* pSrc, Ipp64f val,
    Ipp64f* pDst, int len);
IppStatus ippsSubCRev_32fc(const Ipp32fc* pSrc, Ipp32fc val,
    Ipp32fc* pDst, int len);
IppStatus ippsSubCRev_64fc(const Ipp64fc* pSrc, Ipp64fc val,
    Ipp64fc* pDst, int len);
IppStatus ippsSubCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsSubCRev_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsSubCRev_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsSubCRev_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
IppStatus ippsSubCRev_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
    int len, int scaleFactor);
IppStatus ippsSubCRev_16s_Sfs(const Ipp16s* pSrc, Ipp16s val,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSubCRev_32s_Sfs(const Ipp32s* pSrc, Ipp32s val,
    Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsSubCRev_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val,
    Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsSubCRev_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val,
    Ipp32sc* pDst, int len, int scaleFactor))
IppStatus ippsSubCRev_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsSubCRev_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsSubCRev_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsSubCRev_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
    int scaleFactor);
IppStatus ippsSubCRev_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len,
    int scaleFactor);
```

引数

val	ベクトル要素によって減算されるスカラー値。

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	値 *val* から減算される要素を持つベクトルへのポインタ（インプレース演算の場合)。 減算 *val - pSrcDst[n]* の結果を格納するデスティネーション・ベクトル。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsSubCRev は、ipps.h ファイルで宣言される。この関数は、値 *val* からベクトル *pSrc* の各要素を減算し、その結果を *pDst* に格納する。

インプレース方式の ippsSubCRev は、値 *val* からベクトル *pSrcDst* の各要素を引き、その結果を *pSrcDst* に格納する。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値に対しスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Sub

2 つのベクトルの要素の差を求める。

```
IppStatus ippsSub_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len);
IppStatus ippsSub_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsSub_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsSub_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
    Ipp32fc* pDst, int len);
```

```
IppStatus ippsSub_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
    Ipp64fc* pDst, int len);
IppStatus ippsSub_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsSub_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsSub_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsSub_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsSub_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int
    len);
IppStatus ippsSub_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int
    len);
IppStatus ippsSub_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
    Ipp8u* pDst, int len, int scaleFactor);
IppStatus ippsSub_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsSub_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
    Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsSub_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
    Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsSub_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
    Ipp32sc* pDst, int len, int scaleFactor);
IppStatus ippsSub_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst,
    int len, int scaleFactor);
IppStatus ippsSub_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
    int len, int scaleFactor);
IppStatus ippsSub_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst,
    int len, int scaleFactor);
IppStatus ippsSub_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst,
    int len, int scaleFactor);
IppStatus ippsSub_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst,
    int len, int scaleFactor)
```

引数

pSrc1	*pSrc2* から減算する要素を持つソース・ベクトルへのポインタ。
pSrc2	*pSrc1* の要素で減算し要素を持つソース・ベクトルへのポインタ。
pDst	減算 *pSrc2[n]* - *pSrc1[n]* の結果を格納するデスティネーション・ベクトルへのポインタ。
pSrc	*pSrcDst* の要素からインプレースで減算する要素を持つベクトルへのポインタ。

pSrcDst	*pSrc* の要素で減算され、結果がインプレース演算用の *pSrcDst* に格納される要素を持つベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsSub は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc2* の要素からベクトル *pSrc1* の要素を引き、その結果を *pDst* に格納する。

関数 ippsSub はインプレース演算で、ベクトル *pSrcDst* の要素からベクトル *pSrc* の要素を引き、その結果を *pSrcDst* に格納する。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値に対しスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

DivC

ベクトルの各要素を定数値で割る。

```
IppStatus ippsDivC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
    int len);
IppStatus ippsDivC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst,
    int len);
IppStatus ippsDivC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc*
    pDst, int len);
IppStatus ippsDivC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc*
    pDst, int len);
IppStatus ippsDivC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
IppStatus ippsDivC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
IppStatus ippsDivC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
IppStatus ippsDivC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```


intel®

```
IppStatus ippsDivC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst,
    int len, int ScaleFactor);
IppStatus ippsDivC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s*
    pDst, int len, int ScaleFactor);
IppStatus ippsDivC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc*
    pDst, int len, int ScaleFactor);
IppStatus ippsDivC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len,
    int ScaleFactor);
IppStatus ippsDivC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int
    ScaleFactor);
IppStatus ippsDivC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len,
    int ScaleFactor);
```

引数

val	ベクトル *pSrc* またはベクトル *pSrcDst* の各要素を割るために使用するスカラー値。
pSrc	ソース・ベクトルへのポインタ。
pDst	除算 *pSrc[n] / val* の結果を格納するデスティネーション・ベクトルへのポインタ。
pSrcDst	値 *val* で除算し、結果がインプレース演算用の *pSrcDst* に格納される要素を持つベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDivC は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素を値 *val* で割り、その結果を *pDst* に格納する。

関数 ippsDivC はインプレース演算で、ベクトル *pSrcDst* の各要素を値 *val* で割り、その結果を *pSrcDst* に格納する。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値に対しスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsDivByZeroErr	エラー。*val* がゼロ。

DivCRev

定数値をベクトルの各要素で割る。

```
IppStatus ippsDivCRev_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst,
    int len);
IppStatus ippsDivCRev_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst,
    int len);
IppStatus ippsDivCRev_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsDivCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
```

引数

val	演算の被除数として使用される定数値。
pSrc	除数として使用される要素を持つソース・ベクトルへのポインタ
pDst	除算 *val / pSrc[n]* の結果を格納するデスティネーション・ベクトルへのポインタ。
pSrcDst	除数（インプレース演算の場合）と得られた商の格納の両方に使用される要素を持つベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsDivCRev は、ipps.h ファイルで宣言される。この関数は、定数値 *val* をベクトル *pSrc* の各要素で割り、その結果を *pDst* に格納する。インプレース方式の ippsDivC は、定数値 *val* をベクトル *pSrcDst* の各要素で割り、その結果を *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsDivByZeroErr	エラー。*val* がゼロ。

intel®

Div

2 つのベクトルの要素を除算する。

```
IppStatus ippsDiv_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsDiv_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsDiv_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
    Ipp32fc* pDst, int len);
IppStatus ippsDiv_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
    Ipp64fc* pDst, int len);
IppStatus ippsDiv_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsDiv_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsDiv_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsDiv_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsDiv_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u*
    pDst, int len, int scaleFactor);
IppStatus ippsDiv_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsDiv_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
    Ipp32s* pDst, int len, int scaleFactor);
IppStatus ippsDiv_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc*
    pSrc2,Ipp16sc* pDst, int len, int scaleFactor);
IppStatus ippsDiv_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst,
    int len, int ScaleFactor);
IppStatus ippsDiv_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int
    len, int ScaleFactor);
IppStatus ippsDiv_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int
    len, int ScaleFactor);
IppStatus ippsDiv_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int
    len, int ScaleFactor);
```

引数

pSrc1	*pSrc2* の除数として使用される要素を持つベクトルへのポインタ。
pSrc2	*pSrc1* の要素で割られる要素を持つベクトルへのポインタ。

pDst	除算 *pSrc2[n]* / *pSrc1[n]* の結果を格納するデスティネーション・ベクトル *pDst* へのポインタ。
pSrc	インプレース演算時に *pSrcDst* の除数として使用される要素を持つベクトルへのポインタ。
pSrcDst	*pSrc* の要素で除算し、結果がインプレース演算用の *pSrcDst* に格納される要素を持つベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDiv は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc2* の要素をベクトル *pSrc1* の要素で割り、その結果を *pDst* に格納する。

関数 ippsDiv はインプレース演算で、ベクトル *pSrcDst* の要素をベクトル *pSrc* の要素で割り、その結果を *pSrcDst* に格納する。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値に対しスケーリングを実行する。出力値がデータ範囲を超えると、結果は飽和される。

例 5-3 は、整数関数でスケーリング係数を使用すると演算精度が高くなる様子を示している。例 5-4 は、ゼロ除算例外（x / 0, 0 / 0）について検討している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsDivByZero	警告。ベクトルの除数要素がゼロである。演算の実行は中断されない。浮動小数点演算におけるデスティネーション・ベクトル要素の値は、次のようになる。

NaN	被除数ベクトル要素がゼロ値の場合。
+Inf	被除数ベクトル要素が正の場合。
-Inf	被除数ベクトル要素が負の場合。

例 5-3 ippsDiv_16s_ISfs 関数の使用例

```
IppStatus div16s( void ) {
      Ipp16s x[4] = { -3, 2, 0, 300 };
      Ipp16s y[4] = { -2, 2, 0, 0 };
      IppStatus st = ippsDiv_16s_ISfs( y, x, 4, -1 );
      printf_16s(“div16s =”, x, 4, st );
      return st;
}
Output:
     -- warning 6, Zero value(s) in the divisor of the function Div
     div16s =  3 2 0 32767
```

例 5-4 ippsDiv_32f_I 関数の使用例

```
IppStatus div32f( void ) {
      Ipp32f x[4] = { -3, 2, 0, 300 };
      Ipp32f y[4] = { -2, 2, 0, 0 };
      IppStatus st = ippsDiv_32f_I( y, x, 4 );
      printf_32f( "div32f =", x, 4, st );
      return st;
}
Output:
      -- warning 6, Zero value(s) in the divisor of the function Div
     div32f =  1.500000 1.000000 1.#IND00 1.#INF00
```

Abs

ベクトル要素の絶対値を計算する。

```
IppStatus ippsAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAbs_16s_I(Ipp16s* pSrcDst, int len);
```

```
IppStatus ippsAbs_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsAbs_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsAbs_64f_I(Ipp64f* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsAbs は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素の絶対値を計算し、その結果を *pDst* に格納する。

関数 ippsAbs はインプレース演算で、ベクトル *pSrcDst* の各要素の絶対値を計算し、その結果を *pSrcDst* に格納する。

複素数データの絶対値を計算するには、関数 ippsMagnitude を使用する。

例 5-5 は、関数 ippsAbs_32f_I の呼び出し方法を示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-5　ippsAbs 関数の使用例

```
void abs32f(void) {
      Ipp32f x[4] = {-1, 1, 0, 0};
      x[3] *= (-1);
      ippsAbs_32f_I(x, 4);
      printf_32f("abs =", x, 4, ippStsNoErr);
}
Output:
      abs =  1.000000 1.000000 0.000000 0.000000
```

Sqr

ベクトルの各要素の 2 乗を計算する。

```
IppStatus ippsSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqr_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqr_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqr_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqr_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqr_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqr_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    int scaleFactor);
IppStatus ippsSqr_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16sc_ISfs(Ipp16sc* pSrcDst, int len, int
    scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsSqr は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素の 2 乗を計算し、その結果を *pDst* に格納する。計算は、次のように行われる。

$pDst[n] = pSrc[n]^2$

関数 ippsSqr はインプレース演算で、ベクトル *pSrcDst* の各要素の 2 乗を計算し、その結果を *pSrcDst* に格納する。計算は、次のように行われる。

$pSrcDst[n] = pSrcDst[n]^2$

整数の 2 乗を計算すると、出力結果がデータ範囲を超え、飽和する場合がある。正確な結果を得るには、スケール係数を使用する。例 5-6 は、200^2 の値を得る方法を示している。スケーリングを使用しないと、この結果は 32767 にクリップされる。スケーリングを使用すると出力データ範囲を保持するが、下位ビットの精度は損なわれる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-6 ippsSqr 関数の使用例

```
IppStatus sqr(void) {
      Ipp16s x[4] = {-3, 2, 30, 200};
      IppStatus st = ippsSqr_16s_ISfs(x, 4, 1);
      printf_16s("sqr =", x, 4, st);
      return st;
}
Output:
      sqr =  4 2 450 20000
```

Sqrt

ベクトルの各要素の平方根を計算する。

```
IppStatus ippsSqrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqrt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqrt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqrt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqrt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqrt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqrt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqrt_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqrt_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsSqrt_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsSqrt_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len,
    int scaleFactor);
IppStatus ippsSqrt_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    len, int scaleFactor);
IppStatus ippsSqrt_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int
    len, int scaleFactor);
IppStatus ippsSqrt_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16sc_ISfs(Ipp16sc* pSrcDst, int len, int
    scaleFactor);
IppStatus ippsSqrt_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsSqrt は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素の平方根を計算し、その結果を *pDst* に格納する。計算は、次のように行われる。

$$pDst[n] = \sqrt{pSrc[n]}$$

関数 ippsSqrt はインプレース演算で、ベクトル *pSrcDst* の各要素の平方根を計算し、その結果を *pSrcDst* に格納する。計算は、次のように行われる。

$$pSrcDst[n] = \sqrt{pSrcDst[n]}$$

複素数ベクトルの要素の平方根は、次のように計算する。

$$\sqrt{a + j \cdot b} = \sqrt{\frac{\sqrt{a^2 + b^2} + a}{2}} + j \cdot sign(b) \cdot \sqrt{\frac{\sqrt{a^2 + b^2} - a}{2}}$$

アプリケーション・ノート

関数 ippsSqrt で負の入力値を検出すると、警告ステータスが返される。演算の実行は中断されない。整数出力の精度を上げるには、スケール係数を使用する。

例 5-7 は、関数 ippsSqrt_16s_ISfs の呼び出し方法を示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsSqrtNegArg	警告。演算の実行は中断されない。デスティネーション・ベクトルの要素は、次の値をとる。

NaN	浮動小数点演算の入力ベクトルの要素が負の場合。
0	整数演算の入力ベクトルの要素が負の場合。

例 5-7 ippsSqrt 関数の使用例

```
IppStatus sqrt(void) {
      Ipp16s x[4] = {-3, 2, 30, 300};
      IppStatus st = ippsSqrt_16s_ISfs(x, 4, -1);
      printf_16s("sqrt =", x, 4, st);
      return st;
}
Output:
     -- warning 3, Negative value(s) in the argument of the function Sqrt
    sqrt = 0 3 11 35
```

Cubrt

ベクトルの各要素の立方根を計算する。

```
IppStatus ippsCubrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCubrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int
   len, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsCubrt は、ipps.h ファイルで宣言される。この関数は、*pSrc* の各要素の立方根を計算し、その結果を *pDst* の対応する要素に格納する。

計算は、次のように行われる。

$$pDst[n] = \sqrt[3]{pSrc[n]}, 0 \le n < len$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Exp

e をベクトルの各要素で累乗した値を
計算する。

```
IppStatus ippsExp_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsExp_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsExp_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsExp_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    int scaleFactor);
IppStatus ippsExp_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len,
    int scaleFactor);
IppStatus ippsExp_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsExp_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippsExp_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトル *pSrcDst* へのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsExp は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素の指数関数を計算し、その結果を *pDst* に格納する。

計算は、次のように行われる。

$pDst[n] = e^{pSrc[n]}$

関数 ippsExp はインプレース演算で、ベクトル *pSrcDst* の各要素の指数関数を計算し、その結果を *pSrcDst* に格納する。

計算は、次のように行われる。

$pSrcDst[n] = e^{pSrcDst[n]}$

整数の累乗を計算すると、出力結果がデータ範囲を超え、飽和する場合がある。スケーリング を行うと出力データ範囲を保持するが、下位ビットで結果の精度が損なわれる。関数 ippsExp_32f64f は、データ範囲内でより高い精度の出力結果を計算する。

例 5-8 は関数 ippsExp_16s_ISfs の呼び出し方法を示し、例 5-9 は関数 ippsExp_64f_I の呼び出し方法を示している。

アプリケーション・ノート

関数 ippsExp と関数 ippsLn の場合、結果はスケーリング後に、一番近い整数に丸められる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-8 ippsExp_16s_ISfs 関数の使用例

```
IppStatus exp16s(void) {
      Ipp16s x[4] = {-1, 2, 30, 0};
      IppStatus st = ippsExp_16s_ISfs(x, 4, -1);
      printf_16s("exp16s =", x, 4, st);
      return st;
}
Output:
      exp16s =  1 15 32767 2
```

例 5-9 ippsExp_64f_I 関数の使用例

```
IppStatus exp64f(void) {
      Ipp64f x[4] = {-1, 2, 1, log(1.234567)};
      IppStatus st = ippsExp_64f_I(x, 4);
      printf_64f("exp64f =", x, 4, st);
      return st;
}
Output:  exp64f =  0.367879 7.389056 2.718282 1.234567
```

Ln

ベクトルの各要素の自然対数を計算する。

```
IppStatus ippsLn_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsLn_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsLn_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsLn_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int
    scaleFactor);
```

```
IppStatus ippsLn_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsLn_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsLn_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsLn は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素の自然対数を計算し、その結果を与えられた *pDst* に格納する。

$pDst[n] = \log_e(pSrc[n])$

関数 ippsLn はインプレース演算で、ベクトル *pSrcDst* の各要素の自然対数を計算し、その結果を与えられた *pSrcDst* に格納する。

$pSrcDst[n] = \log_e(pSrcDst[n])$

例 5-10 は、関数 ippsLn_32f_I の呼び出し方法を示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsLnZeroArg	警告。入力ベクトルの要素がゼロである。演算の実行は中断されない。浮動小数点演算の場合、デスティネーション・ベクトルの要素の値は -Inf となる。
ippStsLnNegArg	警告。入力ベクトルの要素が負である。演算の実行は中断されない。浮動小数点演算の場合、デスティネーション・ベクトルの要素の値は NaN となる。

例 5-10　ippsLn 関数の使用例

```
IppStatus ln32f(void) {
        Ipp32f x[4] = {-1, (float)IPP_E, 0, (float)(exp(1.234567))};
        IppStatus st = ippsLn_32f_I(x, 4);
        printf_32f("Ln =", x, 4, st);
        return st;
}
Output:
      -- warning 8, Negative value(s) of argument in the Ln function
     Ln = -1.#IND00 1.000000 -1.#INF00 1.234567
```

10Log10

ベクトルの各要素の 10 進対数を計算し、
それに 10 を掛ける。

```
IppStatus ipps10Log10_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst,
    int len, int scaleFactor);
IppStatus ipps10Log10_32s_ISfs(Ipp32s* pSrcDst, int len,
    int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースおよびデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	「整数のスケーリング」を参照。

説明

関数 ipps10Log10 は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素の 10 進対数に 10 を掛けた値を計算し、その結果を与えられた *pDst* に格納する。

$pDst[n] = 10*\log_{10}(pSrc[n])$

関数 ipps10Log10 はインプレース演算で、ベクトル *pSrcDst* の各要素の 10 進対数に 10 を掛けた値を計算し、その結果を *pSrcDst* に格納する。

$pSrcDst[n] = 10*\log_{10}(pSrcDst[n])$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsLnZeroArg	警告。入力ベクトルの要素がゼロである。演算の実行は中断されない。
ippStsLnNegArg	警告。入力ベクトルの要素が負である。演算の実行は中断されない。

SumLn

ベクトルの各要素の自然対数を合計する。

```
IppStatus ippsSumLn_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum);
IppStatus ippsSumLn_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_32f64f(const Ipp32f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pSum);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pSum	出力結果のポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsSumLn は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素の自然対数を合計し、得られた値を *pSum* に格納する。合計は次の式から得られる。

$$sum = \sum_{n=0}^{len-1} \ln(pSrc[n])$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pSum* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsLnZeroArg	警告。入力ベクトルの要素がゼロである。演算の実行は中断されない。浮動小数点演算の場合、デスティネーション・ベクトルの要素の値は -Inf となる。
ippStsLnNegArg	警告。入力ベクトルの要素が負である。演算の実行は中断されない。浮動小数点演算の場合、デスティネーション・ベクトルの要素の値は NaN となる。

Arctan

ベクトルの各要素の逆正接を計算する。

```
IppStatus ippsArctan_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsArctan_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsArctan_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsArctan_64f_I(Ipp64f* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトル *pSrcDst* へのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsAtan は、ipps.h ファイルで宣言される。この関数は、*pSrc* の各要素の逆正接を計算し、その結果を *pDst* の対応する要素に格納する。

計算は、次のように行われる。

$pDst[n] = \arctan(pSrc[n]),\ 0 \le n < len.$

intel®

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeEr	エラー。*len* がゼロ以下。

Normalize

オフセット演算と除算演算を使用して、実数ベクトルまたは複素数ベクトルの要素を正規化する。

```
IppStatus ippsNormalize_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f vsub, Ipp32f vdiv);
IppStatus ippsNormalize_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f vsub, Ipp64f vdiv);
IppStatus ippsNormalize_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len, Ipp32fc vsub, Ipp32f vdiv);
IppStatus ippsNormalize_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int len, Ipp64fc vsub, Ipp64f vdiv);
IppStatus ippsNormalize_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, Ipp16s vsub, int vdiv, int scaleFactor);
IppStatus ippsNormalize_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len, Ipp16sc vsub, int vdiv, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
vsub	減数値。
vdiv	分母の値。
pDst	正規化した要素を格納するベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsNormalize は、ipps.h ファイルで宣言される。この関数は、入力ベクトル *pSrc* の要素から *vsub* を引き、その差を *vdiv* で割り、結果を *pDst* に格納する。計算は、次のように行われる。計算は、次のように行われる。

$$pDst[n] = \frac{pSrc[n] - vsub}{vdiv}$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsDivByZeroErr	エラー。*vdiv* がゼロか、正の最小浮動小数点数より小さい。

変換関数

この項では、ベクトルに対し次の変換演算を実行する関数について説明する。

- ベクトルのすべての要素を並べ替える。
- データ・タイプを変換する（浮動小数点データから整数データへの変換、整数データから浮動小数点データへの変換など）。
- 複数のベクトルを結合する。
- 複素数ベクトルから成分を抽出し、複素数ベクトルを構築する。
- ベクトルの共役複素数を計算する。
- 直交座標から極座標へ、極座標から直交座標へ変換する。

この項では、複素数ベクトルから実数成分と虚数成分を抽出するインテル® IPP 関数や、実数成分と虚数成分から複素数ベクトルを構築するインテル IPP 関数についても説明する。
関数 ippsReal と関数 ippsImag は、複素数ベクトルの実数部と虚数部をそれぞれ個別のベクトルに返す。
関数 ippsRealToCplx は、2 つのベクトルに個々に格納されている実数成分と虚数成分から、1 つの複素数ベクトルを構築する。
関数 ippsCplxToReal は、複素数ベクトルの実数部と虚数部を、2 つのベクトルに個々に返す。
関数 ippsMagnitude は、複素数ベクトルの要素の大きさを計算する。

また、この項では、V34 レシーバのビタビ・デコーディングを実行する関数について説明する。

SortAscend, SortDescend

ベクトルのすべての要素を並べ替える。

```
IppStatus ippsSortAscend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortAscend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortAscend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortAscend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortAscend_64f_I(Ipp64f* pSrcDst, int len);

IppStatus ippsSortDescend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortDescend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortDescend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortDescend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortDescend_64f_I(Ipp64f* pSrcDst, int len);
```

引数

pSrcDst	ソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素数。

説明

関数 `ippsSortAscend` と関数 `ippsSortDescend` は、`ipps.h` ファイルで宣言される。これらの関数は、指定されたベクタ *pSrcDst* 内のすべての要素を、それぞれ昇順または降順に並べ替え、その結果をデスティネーション・ベクトル *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

SwapBytes

ベクトルのバイト・オーダーを逆にする。

```
IppStatus ippsSwapBytes_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsSwapBytes_24u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsSwapBytes_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsSwapBytes_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSwapBytes_24u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSwapBytes_32u_I(Ipp32u* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースおよびデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素数。

説明

関数 ippsSwapBytes は、ipps.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc*（インプレース演算の場合は *pSrcDst*）のエンディアン・オーダー（バイト・オーダー）を逆にして、その結果を pDst（*pSrcDst*）に格納する。
下位バイトがメモリの最下位アドレスから順番に格納されている場合、リトルエンディアン・オーダーが、上位バイトがメモリの最上位アドレスから順番に格納されている場合、ビッグエンディアン・オーダーが実装されている。関数 ippsSwapBytes を使用して、オーダーを切り替えることができる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Convert

ベクトルのデータ・タイプを変換し、
その結果を２番目のベクトルに格納する。

```
IppStatus ippsConvert_8s16s(const Ipp8s* pSrc,Ipp16s* pDst, int len);
IppStatus ippsConvert_8s32f(const Ipp8s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsConvert_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16u32f(const Ipp16u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s16s(const Ipp32s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsConvert_32s32f(const Ipp32s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s64f(const Ipp32s* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32f_Sfs(const Ipp16s* pSrc, Ipp32f* pDst, int
    len, int scaleFactor);
IppStatus ippsConvert_16s64f_Sfs(const Ipp16s* pSrc, Ipp64f* pDst, int
    len, int scaleFactor);
IppStatus ippsConvert_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int
    len, int scaleFactor);
IppStatus ippsConvert_32s32f_Sfs(const Ipp32s* pSrc, Ipp32f* pDst, int
    len, int scaleFactor);
IppStatus ippsConvert_32s64f_Sfs(const Ipp32s* pSrc, Ipp64f* pDst, int
    len, int scaleFactor);
IppStatus ippsConvert_32f8s_Sfs(const Ipp32f* pSrc, Ipp8s* pDst,
    int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f8u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int
    len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16s_Sfs(const Ipp32f* pSrc, Ipp16s* pDst, int
    len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16u_Sfs(const Ipp32f* pSrc, Ipp16u* pDst, int
    len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f32s_Sfs(const Ipp32f* pSrc, Ipp32s* pDst, int
    len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_64f32s_Sfs(const Ipp64f* pSrc, Ipp32s* pDst, int
    len, IppRoundMode rndMode, int scaleFactor);
```

```
IppStatus ippsConvert_24u32u(const Ipp8u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsConvert_24u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32u24u_Sfs(const Ipp32u* pSrc, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsConvert_32f24u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len,
    int scaleFactor);

IppStatus ippsConvert_24s32s(const Ipp8u* pSrc, Ipp32s* pDst, int len);
IppStatus ippsConvert_24s32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s24s_Sfs(const Ipp32s* pSrc, Ipp8u* pDst, int len,
    int scaleFactor);
IppStatus ippsConvert_32f24s_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len,
    int scaleFactor);

IppStatus ippsConvert_16s16f(const Ipp16s* pSrc, Ipp16f* pDst, int len,
    IppRoundMode rndMode);
IppStatus ippsConvert_32f16f(const Ipp32f* pSrc, Ipp16f* pDst, int len,
    IppRoundMode rndMode);
IppStatus ippsConvert_16f16s_Sfs(const Ipp16f* pSrc, Ipp16s* pDst,
    int len, IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_16f32f(const Ipp16f* pSrc, Ipp32f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
rndMode	丸めモード。*ippRndZero* または *ippRndNear* を指定する。 *ippRndZero* 浮動小数点値をゼロに切り捨てる場合に指定する。 *ippRndNear* 浮動小数点値を一番近い整数に丸める場合に指定する。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsConvert は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* 内の整数データを浮動小数点データに変換し、その結果を *pDst* に格納する。

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値のスケーリングを実行する。出力データ範囲を超えた変換結果は飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Join

複数のベクトルの浮動小数点データを整数データに変換し、その結果を 1 つのベクトルに格納する。

```
IppStatus ippsJoin_32f16s_D2L(const Ipp32f** pSrc, int nChannels,
    int chanLen, Ipp16s* pDst);
```

引数

pSrc	入力ベクトルへのポインタの配列へのポインタ。
pDst	出力ベクトルへのポインタ。
nChannels	ベクトルの数。
chanlen	各入力ベクトル内の要素の数。

説明

関数 ippsJoin は、ipps.h ファイルで宣言される。この関数は、配列 *pSrc* 内に格納された *nChannels* 入力ベクトルの浮動小数点データを整数データ・タイプに変換し、その結果をデスティネーション・ベクトル *pDst* に書き込む。要素の書き込みは、配列内の最初のベクトルの最初の要素、2 番目のベクトルの最初の要素、...、最後のベクトルの最初の要素、最初のベクトルの 2 番目の要素、2 番目のベクトルの 2 番目の要素、... という順序で行われる。
変換された値が出力データ範囲を超えた場合は、その値は飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。指定されたポインタのうち少なくとも 1 つが NULL。
ippStsSizeErr	エラー。*nChannels* または *chanlen* がゼロ以下

Conj

ベクトルの共役複素数の値を 2 番目の
ベクトルに格納する（インプレース方式）。

```
IppStatus ippsConj_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsConj_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsConj_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsConj_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsConj_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsConj_64fc_I(Ipp64fc* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsConj は、ipps.h ファイルで宣言される。この関数は、複素数ベクトル *pSrc* の共役を要素単位で *pDst* に格納する。ベクトルの要素単位の共役は、次のように定義される。

pDst[n].re = pSrc[n].re
pDst[n].im = - pSrc[n].im

関数 ippsConj はインプレース演算で、複素数ベクトル *pSrcDst* の共役を要素単位で *pSrcDst* に格納する。

ベクトルの要素単位の共役は、次のように定義される。

```
pSrcDst[n].re = pSrcDst[n].re
pSrcDst[n].im = - pSrcDst[n].im
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

ConjFlip

ベクトルの共役複素数を計算し、その結果を逆順で格納する。

```
IppStatus ippsConjFlip_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len);
IppStatus ippsConjFlip_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len);
IppStatus ippsConjFlip_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int len);
```

引数

pSrc	ソース・ベクトル *pSrc* へのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsConjFlip は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の共役を計算し、その結果を逆順で *pDst* に格納する。

逆順で格納される共役複素数は、次のように定義される。

```
pDst[n] = conj(pSrc[len - n - 1])
```

関数 ippsConjFlip はインプレース演算で、ベクトル *pSrcDst* の共役を計算し、その結果を逆順で *pSrcDst* に格納する。

逆順で格納される共役複素数は、次のように定義される。

pSrcDst[n] = conj(pSrcDst[len - n - 1])

pSrc と *pDst* がメモリ内でオーバーラップすると、予期せぬ結果が返されるのに注意しなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Magnitude

複素数ベクトルの要素の大きさを計算する。

```
IppStatus ippsMagnitude_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDst, int len);
IppStatus ippsMagnitude_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDst, int len);
IppStatus ippsMagnitude_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int
    len);
IppStatus ippsMagnitude_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int
    len);
IppStatus ippsMagnitude_16s32f(const Ipp16s* pSrcRe, const Ipp16s*
    pSrcIm, Ipp32f* pDst, int len);
IppStatus ippsMagnitude_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst,
    int len);
IppStatus ippsMagnitude_16s_Sfs(const Ipp16s* pSrcRe,
    const Ipp16s* pSrcIm, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsMagnitude_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsMagnitude_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst,
    int len, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pSrcRe	複素数の要素の実数部を持つベクトルへのポインタ。
pSrcIm	複素数の要素の虚数部を持つベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsMagnitude は、ipps.h ファイルで宣言される。この関数は複素数演算で、複素数ベクトル *pSrc* の大きさを要素単位で計算し、その結果を *pDst* に格納する。要素単位の大きさは、次の公式で定義する。

$$magn[n] = \sqrt{pSrc[n].re^2 + pSrc[n].im^2}$$

関数 ippsMagnitude は実数演算で、実数成分と虚数成分がそれぞれベクトル *pSrcRe* とベクトル *pSrcIm* で指定された複素数ベクトルの大きさを要素単位で計算し、その結果を *pDst* に格納する。要素単位の大きさは、次の式で定義する。

$$magn[n] = \sqrt{pSrcRe[n]^2 + pSrcIm[n]^2}$$

例 5-11 は、$\sin^2 x + \cos^2 x = 1$ を確認している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrEr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-11 ippsMagnitude 関数の使用例

```
void magn(void) {
        Ipp64f x[6], magn[4];
        int n;
        for (n = 0; n<6; ++n) x[n] = sin(IPP_2PI * n / 8);
        ippsMagnitude_64f(x, x+2, magn, 4);
        printf_64f("magn =", magn, 4, ippStsNoErr);
}
Output:
        magn =  1.000000 1.000000 1.000000 1.000000
Matlab* Analog:
        >> n = 0:9; x = sin(2*pi*n/8); z = [x(1:8)+j*x(3:10)]; abs(z(1:4))
```

MagSquared

複素数ベクトルの要素の大きさの 2 乗を計算する。

```
IppStatus ippsMagSquared_32sc32s_Sfs(const Ipp32sc* pSrc,
    Ipp32s* pDst, int len, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	「整数のスケーリング」を参照。

説明

関数 ippsMagSquared は、ipps.h ファイルで宣言される。この関数は、複素数ベクトル *pSrc* の要素単位の大きさを 2 乗し、その結果を *pDst* に格納する。要素単位の大きさの 2 乗は、次の式で定義する。

$$magn[n] = pSrc[n].re^2 + pSrc[n].im^2$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Phase

複素数入力ベクトルの要素の位相角度を 2 番目のベクトルに返す。

```
IppStatus ippsPhase_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPhase_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len,
    int scaleFactor);
IppStatus ippsPhase_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len,
    int scaleFactor);
IppStatus ippsPhase_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
    Ipp64f* pDst, int len);
IppStatus ippsPhase_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
    Ipp32f* pDst, int len);
IppStatus ippsPhase_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp32f* pDst, int len);
IppStatus ippsPhase_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
    Ipp16s* pDst, int len, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pSrcRe	実数成分を格納するソース・ベクトルへのポインタ。
pSrcIm	虚数成分を格納するソース・ベクトルへのポインタ。
pDst	要素の位相（角度）成分をラジアン単位で格納するベクトルへのポインタ。位相の値の範囲は、$(-\pi, \pi]$ である。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsPhase は、ipps.h ファイルで宣言される。この関数は、この関数は、複素数入力ベクトル *pSrc* の要素、または実数成分と虚数成分がそれぞれベクトル *pSrcRe* とベクトル *pSrcIm* で指定している複素数入力ベクトル要素の位相角度を計算し、結果をベクトル *pDst* に格納する。位相の値は、ラジアン単位で (-π, π] の範囲で返される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

PowerSpectr

複素数ベクトルのパワー・スペクトラムを計算する。

```
IppStatus ippsPowerSpectr_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int
    len);
IppStatus ippsPowerSpectr_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int
    len);
IppStatus ippsPowerSpectr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s*
    pDst,int len, int scaleFactor);
IppStatus ippsPowerSpectr_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst,
    int len);
IppStatus ippsPowerSpectr_64f(const Ipp64f* pSrcRe, const Ipp64f*
    pSrcIm, Ipp64f* pDst, int len);
IppStatus ippsPowerSpectr_32f(const Ipp32f* pSrcRe, const Ipp32f*
    pSrcIm, Ipp32f* pDst, int len);
IppStatus ippsPowerSpectr_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s*
    pSrcIm, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsPowerSpectr_16s32f(const Ipp16s* pSrcRe, const Ipp16s*
    pSrcIm, Ipp32f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pSrcRe	実数成分を格納するソース・ベクトルへのポインタ。

pSrcIm	虚数成分を格納するソース・ベクトルへのポインタ。
pDst	要素のスペクトラム成分を格納するベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsPowerSpectr は、ipps.h ファイルで宣言される。この関数は、この関数は、複素数入力ベクトル *pSrc* のパワー・スペクトラム、または実数成分と虚数成分がそれぞれベクトル *pSrcRe* とベクトル *pSrcIm* で指定している複素数入力ベクトルのパワー・スペクトラムを計算し、結果をベクトル *pDst* に格納する。パワー・スペクトラムの要素は、複素数入力ベクトルの要素の大きさの 2 乗となる。

$pDst[n] = (pSrc[n].re)^2 + (pSrc[n].im)^2$ または
$pDst[n] = (pSrcRe[n])^2 + (pSrcIm[n])^2$

大きさを計算するには、5-56 ページの関数 ippsMagnitude を使用する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Real

複素数ベクトルの実数部を 2 番目のベクトルに返す。

```
IppStatus ippsReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, int len);
IppStatus ippsReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe, int len);
IppStatus ippsReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe, int len);
```

引数

pSrc	複素数ソース・ベクトルへのポインタ。
pDstRe	実数部を持つデスティネーション・ベクトルへのポインタ。

len	ベクトル内の要素の数。

説明

関数 ippsReal は、ipps.h ファイルで宣言される。この関数は、複素数ベクトル *pSrc* の実数部をベクトル *pDstRe* に返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDstRe* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Imag

複素数ベクトルの虚数部を 2 番目のベクトルに返す。

```
IppStatus ippsImag_16sc(const Ipp16sc* pSrc, Ipp16s* pDstIm, int len);
IppStatus ippsImag_32fc(const Ipp32fc* pSrc, Ipp32f* pDstIm, int len);
IppStatus ippsImag_64fc(const Ipp64fc* pSrc, Ipp64f* pDstIm, int len);
```

引数

pSrc	複素数ソース・ベクトルへのポインタ。
pDstIm	虚数部を持つデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsImag は、ipps.h ファイルで宣言される。この関数は、複素数ベクトル *pSrc* の虚数部をベクトル *pDstIm* に返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDstIm* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

RealToCplx

2 つの実数ベクトルの実数部と虚数部から
構築した複素数ベクトルを返す。

```
IppStatus ippsRealToCplx_16s(const Ipp16s* pSrcRe, const Ipp16s*
   pSrcIm, Ipp16sc* pDst, int len);
IppStatus ippsRealToCplx_32f(const Ipp32f* pSrcRe, const Ipp32f*
   pSrcIm, Ipp32fc* pDst, int len);
IppStatus ippsRealToCplx_64f(const Ipp64f* pSrcRe, const Ipp64f*
   pSrcIm, Ipp64fc* pDst, int len);
```

引数

pSrcRe	複素数の要素の実数部を持つベクトルへのポインタ。
pSrcIm	複素数の要素の虚数部を持つベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsRealToCplx は、ipps.h ファイルで宣言される。この関数は、2 つの入力ベクトル *pSrcRe* と *pSrcIm* の実数部と虚数部から構築した複素数ベクトル *pDst* を返す。

pSrcRe が NULL の場合、ベクトルの実数成分をゼロに設定する。

pSrcIm が NULL の場合、ベクトルの虚数成分をゼロに設定する。

両方のポインタが同時に NULL であってはならない点に注意しなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタが NULL。ポインタ *pSrcRe* または *pSrcIm* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

CplxToReal

複素数ベクトルの実数部と虚数部を 2 つの
ベクトルにそれぞれ返す。

```
IppStatus ippsCplxToReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe,
    Ipp16s* pDstIm, int len);
IppStatus ippsCplxToReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe,
    Ipp32f* pDstIm, int len);
IppStatus ippsCplxToReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe,
    Ipp64f* pDstIm, int len);
```

引数

pSrc	複素数ベクトル *pSrc* へのポインタ。
pDstRe	実数部を持つ出力ベクトルへのポインタ。
pDstIm	虚数部を持つ出力ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsCplxToReal は、ipps.h ファイルで宣言される。この関数は、複素数ベクトル *pSrc* の実数部と虚数部を、2 つのベクトル *pDstRe* と *pDstIm* に返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。データ・ベクトルが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Threshold

要素の値を *level* で制限しながら、ベクトル
の要素に対するしきい値演算を実行する。

```
IppStatus ippsThreshold_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp16s level, IppCmpOp relOp);
```

```
IppStatus ippsThreshold_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    Ipp32f level, IppCmpOp relOp);
IppStatus ippsThreshold_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    Ipp64f level, IppCmpOp relOp);
IppStatus ippsThreshold_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
    Ipp16s level, IppCmpOp relOp);
IppStatus ippsThreshold_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level,
    IppCmpOp relOp);
IppStatus ippsThreshold_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level,
    IppCmpOp relOp);
IppStatus ippsThreshold_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level,
    IppCmpOp relOp);
IppStatus ippsThreshold_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level,
    IppCmpOp relOp);
IppStatus ippsThreshold_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level,
    IppCmpOp relOp);
IppStatus ippsThreshold_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level,
    IppCmpOp relOp);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
level	*pSrc* または *pSrcDst* の各要素を制限するために使用する値。この引数には、常に実数を指定する必要がある。複素数型に対しては、正の値で、大きさを表すものでなければならない。
relOp	この引数の値は、使用する関係演算子を指定するとともに、*level* が入力に対する上限か下限かを指定する。*relOp* には、次の値のいずれかを指定する。

ippCmpLess	「より小さい」演算子を指定し、*level* は下限になる。

ippCmpGreater「より大きい」演算子を指定し、*level* は上限になる。

説明

関数 ippsThreshold は、ipps.h ファイルで宣言される。この関数は、しきい値 *level* で各要素を制限しながら、ベクトル *pSrc* に対するしきい値演算を実行する。この関数演算は、関数 ippsThreshold_LT と ippsThreshold_GT に似ているが、比較演算のタイプを指定する *relOp* 引数が含まれている。

関数 ippsThreshold はインプレース演算で、しきい値 *level* で各要素を制限しながら、ベクトル *pSrcDst* に対するしきい値演算を実行する。

relOp 引数では、「より大きい」または「より小さい」のどちらの関係演算子を使用するのかを指定するとともに、*level* が入力に対する上限か下限かを決定する。*IppCmpLess* フラグを指定して呼び出した ippsThreshold に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

関数 ippsThreshold の複素数型に対して、*level* 引数は常に実数になる。*IppCmpLess* フラグを指定して呼び出した複素数 ippsThreshold に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} \dfrac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

アプリケーション・ノート

すべての複素数型に対して、*level* は正の値で、大きさを表すものでなければならない。入力の大きさは制限されるが、位相は変化しない。ゼロの値を持つ入力は、ゼロの位相を持つものとみなされる。

整数複素数型の関数 ippsThreshold には、特別な規則が適用される。一般に、複素数平面での結果点の座標は整数ではない。関数は、しきい値演算を実行しない方法で、座標の値を整数に切り捨てる。したがって、「より小さい」演算（*ippCmpLess* フラグを指定）の場合、座標は無限大に丸められる（正の座標の場合は +Inf、負の座標の場合は -Inf)。「より大きい」演算（*ippCmpGreater* フラグを指定）の場合、座標はゼロに丸められる。「複素数」関数 ippsThreshold_16sc_I の詳しい使用方法は、例 5-13 を参照のこと。

例 5-12 は、「実数」関数 ippsTreshold_16s_I の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsThreshNegLevelErr	エラー。複素数型に対する *level* が負。

例 5-12　実数型の ippsThreshold 関数の使用例

```
IppStatus threshold( void ) {
      Ipp16s x[4] = { -1, 0, 2, 3 };
      IppStatus st = ippsThreshold_16s_I( x, 4, 2, ippCmpLess );
      printf_16s("threshold result =", x, 4, st );
      return st;
}
Output:
       threshold result =  2 2 2 3
```

例 5-13　複素数型の ippsThreshold 関数の使用例

```
IppStatus cmplx_threshold(void) {
      Ipp16sc x[4] = {{2,3}, {3,3}, {4,3}, {4,2}};
      /// level is near to the point {2,3} = 3.6
      /// the point {2,3} is to be replaced
      /// the computed coordinates are {2.2188,3.3282}
      /// the point used is {3,4};
      /// notice that it is the point with the phase,
      /// nearest to the source
      IppStatus st = ippsThreshold_16sc_I(x, 4, 4, ippCmpLess);
      printf_16sc("complex threshold result =", x, 4, st);
      return st;
}
Output:
       complex threshold result = {3, 4} {3, 3} {4, 3} {4, 2}
```

Threshold_LT, Threshold_GT

要素の値を *level* で制限しながら、ベクトルの要素に対するしきい値演算を実行する。

```
IppStatus ippsThreshold_LT_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, Ipp16s level);
IppStatus ippsThreshold_LT_32s(const Ipp32s* pSrc, Ipp32s* pDst,
    int len, Ipp32s level);
IppStatus ippsThreshold_LT_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    int len, Ipp64f level);
IppStatus ippsThreshold_LT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len, Ipp16s level);
IppStatus ippsThreshold_LT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_LT_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_LT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, Ipp16s level);
IppStatus ippsThreshold_GT_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
    len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    len, Ipp16s level);
IppStatus ippsThreshold_GT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
```

```
IppStatus ippsThreshold_GT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
level	*pSrc* または *pSrcDst* の各要素を制限するために使用する値。この引数には、常に実数を指定する必要がある。複素数型に対しては、正の値で、大きさを表すものでなければならない。

説明

関数 ippsThreshold_LT と ippsThreshold_GT は、ipps.h ファイルで宣言される。これらは、各要素をしきい値 *level* で制限しながら、ベクトル *pSrc* に対するしきい値演算を実装する。これらの関数演算は、ippsThreshold 関数と似ているが、比較演算の固定タイプを使用するように設計されている。ippsThreshold_LT では、「より小さい」比較を行い、ippsThreshold_GT では、「より大きい」比較を行う。

この関数はインプレース演算で、各要素をしきい値 *level* で制限しながら、ベクトル *pSrcDst* に対するしきい値演算を実行する。

ippsThreshold_LT。ippsThresholdLT 関数は、「より小さい」演算を実行する。*level* は入力に対する下限となる。IppsThresholdLT に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

複素数型の ippsThreshold_LT 関数の場合、*level* 引数は常に実数になる。

複素数型の ippsThreshold_LT に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} \dfrac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_GT。関数 ippsThreshold_GT は、「より大きい」演算を実行する。また、*level* は、入力に対する上限になる。

ippsThreshold_GT に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} level, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

複素数型の ippsThreshold_GT 関数の場合、*level* 引数は常に実数になる。

複素数型の ippsThreshold_GT に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} \dfrac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

アプリケーション・ノート

すべての複素数型に対して、*level* は正の値で、大きさを表すものでなければならない。入力の大きさは制限されるが、位相は変化しない。ゼロの値を持つ入力は、ゼロの位相を持つものとみなされる。

複素数型整数のしきい値関数には、特別な規則が適用される。一般に、複素数平面での結果点の座標は整数ではない。関数は、しきい値演算が実行されない方法で、座標の値を整数に切り捨てる。したがって、「より小さい」演算（ippsThreshold_LT 関数）の場合、座標は無限大に丸められる（正の座標の場合は +Inf、負の座標の場合は -Inf)。「より大きい」演算（ippsThreshold_GT 関数）の場合、座標はゼロに丸められる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsThreshNegLevelErr	エラー。複素数型に対する *level* が負。

Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal

要素の値を *level* で制限しながら、ベクトルの要素に対するしきい値演算を実行する。

```
IppStatus ippsThreshold_LTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_LTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len, Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_LTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
    len, Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_LTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len, Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_LTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len, Ipp32f level, Ipp32fc value);
IppStatus ippsThreshold_LTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int len, Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_LTVal_16s_I(Ipp16s* pSrcDst, int len,
    Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_LTVal_32f_I(Ipp32f* pSrcDst, int len,
    Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_LTVal_64f_I(Ipp64f* pSrcDst, int len,
    Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_LTVal_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s
    level, Ipp16sc value);
IppStatus ippsThreshold_LTVal_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f
    level, Ipp32fc value);
IppStatus ippsThreshold_LTVal_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f
    level, Ipp64fc value);
IppStatus ippsThreshold_GTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_GTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len, Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_GTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
    len, Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_GTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len, Ipp16s level, Ipp16sc value);
IppStatus ippsThreshold_GTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len, Ipp32f level, Ipp32fc value);
```

```
IppStatus ippsThreshold_GTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int len, Ipp64f level, Ipp64fc value);
IppStatus ippsThreshold_GTVal_16s_I(Ipp16s* pSrcDst, int len,
    Ipp16s level, Ipp16s value);
IppStatus ippsThreshold_GTVal_32f_I(Ipp32f* pSrcDst, int len,
    Ipp32f level, Ipp32f value);
IppStatus ippsThreshold_GTVal_64f_I(Ipp64f* pSrcDst, int len,
    Ipp64f level, Ipp64f value);
IppStatus ippsThreshold_GTVal_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s
    level, Ipp16sc value);
IppStatus ippsThreshold_GTVal_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f
    level, Ipp32fc value);
IppStatus ippsThreshold_GTVal_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f
    level, Ipp64fc value);

IppStatus ippsThreshold_LTValGTVal_16s(const Ipp16s* pSrc,
    Ipp16s* pDst, int len, Ipp16s levelLT, Ipp16s valueLT, Ipp16s
    levelGT, Ipp16s valueGT);
IppStatus ippsThreshold_LTValGTVal_32f(const Ipp32f* pSrc,
    Ipp32f* pDst, int len, Ipp32f levelLT, Ipp32f valueLT, Ipp32f
    levelGT, Ipp32f valueGT);
IppStatus ippsThreshold_LTValGTVal_64f(const Ipp64f* pSrc,
    Ipp64f* pDst, int len, Ipp64f levelLT, Ipp64f valueLT, Ipp64f
    levelGT, Ipp64f valueGT);

IppStatus ippsThreshold_LTValGTVal_16s_I(Ipp16s* pSrcDst, int len,
    Ipp16s levelLT, Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);
IppStatus ippsThreshold_LTValGTVal_32f_I(Ipp32f* pSrcDst, int len,
    Ipp32f levelLT, Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);
IppStatus ippsThreshold_LTValGTVal_64f_I(Ipp64f* pSrcDst, int len,
    Ipp64f levelLT, Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
level	*pSrc* または *pSrcDst* の各要素を制限するために使用する値。この引数には、常に実数を指定する必要がある。複素数型に対しては、正の値で、大きさを表すものでなければならない。

levelLT	ippsThreshold_LTValGTVal 関数で *pSrc* または *pSrcDst* の各要素を制限するために使用する下限。
levelGT	ippsThreshold_LTValGTVal 関数で *pSrc* または *pSrcDst* の各要素を制限するために使用する上限。
value	*level* よりも大きいか小さいベクトル要素に割り当てる値。
valueLT	ippsThreshold_LTValGTVal 関数で *levelLT* より小さいベクトル要素に割り当てられる値。
valueGT	ippsThreshold_LTValGTVal 関数で *levelGT* より大きいベクトル要素に割り当てられる値。

説明

これらの関数は、ipps.h ファイルで宣言される。これらは、各要素をしきい値で制限しながら、ベクトル *pSrc* に対するしきい値演算を実行する。

この関数はインプレース演算で、各要素をしきい値で制限しながら、ベクトル *pSrcDst* に対するしきい値演算を実行する。

ippsThreshold_LTVal。ippsThreshold_LTVal 関数は「より小さい」演算を実行する。*level* は入力に対する下限となる。*level* より小さなベクトル要素には *value* が設定される。

ippsThreshold_LTVal に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} value, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

複素数型の ippsThreshold_LTVal 関数の場合、*level* 引数は常に実数となる。

複素数型の ippsThreshold_LTVal に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_GTVal。ippsThreshold_GTVal 関数は「より大きい」演算を実行する。*level* は入力に対する上限となる。*level* より大きなベクトル要素には *value* が設定される。

ippsThreshold_GtVal に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} value, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

複素数型の ippsThreshold_GTVal 関数の場合、*level* 引数は常に実数になる。

複素数型の ippsThreshold_GTVal に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} value, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

ippsThreshold_LTValGTVal。ippsThreshold_LTValGTVal 関数は、「より小さい」条件と「より大きい」条件の両方をチェックする。引数 *levelLT* は入力の下限、引数 *levelGT* は入力の上限を指定する。*levelLT* より小さいソース・ベクトル要素は *valueLT* に設定され、*levelGT* より大きいソース・ベクトル要素は *valueGT* に設定される。*levelLT* の値は、*levelGT* 以下でなければならない。

ippsThreshold_LTValGTVal に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} valueLT, & pSrc[n] < levelLT \\ pSrc[n], & levelLT \le pSrc[n] \le levelGT \\ valueGT, & pSrc[n] > levelGT \end{cases}$$

アプリケーション・ノート

すべての複素数型に対して、*level* は正の値で、大きさを表すものでなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*, *pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsThresholdErr	エラー。*levelLT* が *levelGT* より大きい。
ippStsThreshNegLevelErr	エラー。複素数型に対する *level* が負。

Threshold_LTInv

下限でベクトルの各要素の大きさを制限した後で、ベクトルの各要素の逆を計算する。

```
IppStatus ippsThreshold_LTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
    len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32f_I(Ipp32f* pSrcDst, int len, Ipp32f
    level);
IppStatus ippsThreshold_LTInv_64f_I(Ipp64f* pSrcDst, int len, Ipp64f
    level);
IppStatus ippsThreshold_LTInv_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f
    level);
IppStatus ippsThreshold_LTInv_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f
    level);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
level	*pSrc* または *pSrcDst* の各要素を制限するために使用する値。この引数には、常に正の実数を指定する必要がある。

説明

関数 ippsThreshold_LTInv は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の要素の逆を計算し、その結果を *pDst* に格納する。この計算は、最初に各要素の大きさをしきい値 *level* で制限した後に実行される。

関数 ippsThreshold_LTInv はインプレース演算で、ベクトル *pSrcDst* の要素の逆を計算し、その結果を *pSrcDst* に格納する。この計算は、最初に各要素の大きさをしきい値 *level* で制限した後に実行される。

しきい値演算は、ゼロ除算を防ぐために実行する。*level* は大きさを表すため、その値は常に正の実数でなければならない。ippsThreshold_LTInv に対する式は、次のとおりである。

$$pDst[n] = \begin{cases} \dfrac{1}{level}, & abs(pSrc[n]) = 0 \\ \dfrac{abs(pSrc[n])}{pSrc[n] \cdot level}, & 0 < abs(pSrc[n]) < level \\ \dfrac{1}{pSrc[n]}, & otherwise \end{cases}$$

関数でゼロ値のベクトル要素が検出され、*level* もゼロである場合は、出力値に Inf（無限大）を設定する。ただし、演算は中断されない。

$$pDst[n] = \begin{cases} Inf, & pSrc[n] = 0 \\ \dfrac{1}{pSrc[n]}, & otherwise \end{cases}$$

例 5-14 は、関数 ippsThreshold_LTInv_32f_I の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsThreshNegLevelErr	エラー。*level* が負。
ippStsInvZero	警告。*level* とベクトル要素がゼロである。演算の実行は中断されない。デスティネーション・ベクトルの要素の値は Inf になる。

例 5-14 ippsThreshold_LTInv 関数の使用例

```
IppStatus invThreshold(void) {
     Ipp32f x[4] = {-1, 0, 2, 3};
     IppStatus st = ippsThreshold_LTInv_32f_I(x, 4, 0);
     printf_32f("inv threshold =", x, 4, st);
     return st;
}
Output:
      -- warning 4, INF result. Zero value met by invThreshold with zero level
    inv threshold = -1.000000 1.#INF00 0.500000 0.333333
```

CartToPolar

複素数ベクトルの要素を極座標形式に
変換する。

```
IppStatus ippsCartToPolar_32fc(const Ipp32fc* pSrc, Ipp32f* pDstMagn,
    Ipp32f* pDstPhase, int len);
IppStatus ippsCartToPolar_64fc(const Ipp64fc* pSrc, Ipp64f* pDstMagn,
    Ipp64f* pDstPhase, int len);
IppStatus ippsCartToPolar_32f(const Ipp32f* pSrcRe, const Ipp32f*
    pSrcIm, Ipp32f* pDstMagn, Ipp32f* pDstPhase, int len);
IppStatus ippsCartToPolar_64f(const Ipp64f* pSrcRe, const Ipp64f*
    pSrcIm, Ipp64f* pDstMagn, Ipp64f* pDstPhase, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pSrcRe	直交座標 X、Y の実数成分を格納するソース・ベクトルへのポインタ。
pSrcIm	直交座標 X、Y の虚数成分を格納するソース・ベクトルへのポインタ。
pDstMagn	ベクトル *pSrc* の要素の大きさ（半径）成分を格納するベクトルへのポインタ。

pDstPhase	ベクトル *pSrc* の要素の位相（角度）成分をラジアン単位で格納するベクトルへのポインタ。位相の値の範囲は、(- π , π] である。
len	ベクトル内の要素の数。

説明

関数 ippsCartToPolar は、ipps.h ファイルで宣言される。この関数は、複素数入力ベクトル *pSrc*（またはベクトル *pSrcRe* と *pSrcIm* で指定された実数成分と虚数成分を持つ複素数入力ベクトル）の要素を極座標形式に変換し、各要素の大きさ（半径）成分をベクトル *pDstMagn* に、各要素の位相（角度）成分をベクトル *pDstPhase* に格納する。

例 5-15 は、点が単位半径の円内にあることを確認している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-15　ippsCartToPolar 関数の使用例

```
IppStatus cart2polar( void ) {
     Ipp64f cart[6], magn[4], phase[4];
     int n;
     for (n=0; n<6; ++n) cart[n] = sin(IPP_2PI * n / 8);
     IppStatus st = ippsCartToPolar_64f( cart, cart+2, magn, phase, 4 );
     printf_64f( "magn =", magn, 4, st );
     return st;
}

Output:
      magn =  1.000000 1.000000 1.000000 1.000000
```

PolarToCart

実数部 / 虚数部の組からなる入力ベクトルを
極座標形式から複素数形式に変換する。

```
IppStatus ippsPolarToCart_32fc(const Ipp32f* pSrcMagn,
    const Ipp32f* pSrcPhase, Ipp32fc* pDst, int len);
IppStatus ippsPolarToCart_64fc(const Ipp64f* pSrcMagn,
    const Ipp64f* pSrcPhase, Ipp64fc* pDst, int len);
```

```
IppStatus ippsPolarToCart_32f(const Ipp32f* pSrcMagn, const
    Ipp32f* pSrcPhase, Ipp32f* pDstRe, Ipp32f* pDstIm, int len);
IppStatus ippsPolarToCart_64f(const Ipp64f* pSrcMagn, const
    Ipp64f* pSrcPhase, Ipp64f* pDstRe, Ipp64f* pDstIm, int len);
```

引数

pSrcMagn	極座標形式で要素の大きさ（半径）成分を格納するソース・ベクトルへのポインタ。
pSrcPhase	極座標形式で要素の位相（角度）成分をラジアン単位で格納するベクトルへのポインタ。位相の値は、$(-\pi, \pi]$ の範囲内である。
pDst	直交座標（X+iY）で複素数値を格納する変換後のベクトルへのポインタ。
pDstRe	直交座標 X、Y の実数成分を格納する変換後のベクトルへのポインタ。
pDstIm	直交座標 X、Y の虚数成分を格納する変換後のベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsPolarToCart は、ipps.h ファイルで宣言される。関数 ippsPolarToCart は、入力ベクトル *pSrcMagn* と *pSrcPhase* に極座標形式で格納された大きさ / 位相の組を複素数ベクトルに変換し、その結果をベクトル *pDst* に格納するか、あるいは結果の実数成分をベクトル *pDstRe* に格納し、虚数成分をベクトル *pDstIm* に格納する。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

MaxOrder

ベクトルの最大次数を計算する。

```
IppStatus ippsMaxOrder_16s(const Ipp16s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32s(const Ipp32s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32f(const Ipp32f* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_64f(const Ipp64f* pSrc, int len, int* pOrder);
```

引数

pSrc	ソース・ベクトルへのポインタ。
len	ベクトル内の要素の数。
pOrder	結果の値へのポインタ。

説明

関数 ippsMaxOrder は、ipps.h ファイルで宣言される。この関数は、指数ベクトル *pSrc* の要素の最大バイナリ数を検索し、その結果を *pOrder* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pOrder* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsNanArg	警告。入力データ・ベクトルで NaN が検出された。

Preemphasize

単精度実数信号のプリエンファシスを
計算する（インプレース方式）。

```
IppStatus ippsPreemphasize_16s(Ipp16s* pSrcDst, int len, Ipp32f val);
```

```
IppStatus ippsPreemphasize_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
```

引数

pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
val	差信号プリエンファシス式で使用する乗数。

説明

インプレース関数 ippsPreemphasize は、ipps.h ファイルで宣言される。この関数は、実数信号 *pSrcDst* のプリエンファシスを計算する。この計算は、次の差信号プリエンファシス式に従って行う。

$$y(n) = x(n) - val \cdot x(n - 1),$$

y(n) はプリエンファシスされた出力、*x(n)* は入力、*val* は乗数を表す。

一般に、音声信号の場合は *val*=0.95 になる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Flip

ベクトル内の要素の順序を逆にする。

```
IppStatus ippsFlip_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsFlip_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsFlip_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsFlip_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsFlip_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsFlip_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsFlip_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsFlip_64f_I(Ipp64f* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsFlip` は、`ipps.h` ファイルで宣言される。この関数は、次の式に従って、ソース・ベクトル *pSrc* の要素を逆の順序でデスティネーション・ベクトル *pDst* に格納する。

pDst[*n*] = *pSrc*[*len*-*n*-1], *n*=0..*len*-1

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

FindNearestOne

指定された値に最も近いテーブルの要素を見つける。

```
IppStatus ippsFindNearestOne_16u(Ipp16u inpVal, Ipp16u* pOutVal,
    int* pOutIndex, const Ipp16u *pTable, int tblLen);
```

引数

inpVal	基準値。
pOutVal	出力値へのポインタ。
pOutIndx	出力インデックスへのポインタ。
pTable	検索するテーブルへのポインタ。
tbllen	テーブル内の要素の数。

説明

関数 ippsFindNearestOne は、ipps.h ファイルで宣言される。この関数は、指定された基準値 *inpVal* に最も近い要素をテーブル *pTable* 内で検索する。得られた要素とそのインデックスは、それぞれ *pOutVal* と *pOutIndex* に格納される。
テーブルの要素は、*pTable*[i] ≤*pTable*[*i+1*] の条件を満たしている必要がある。

この関数は、次のような距離の判定法を使用して、最も近い要素を決定する。
min(|*inpVal*-*pTable*[*i*]|)

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsSizeErr	エラー。*tbllen* がゼロ以下。

FindNearest

指定されたベクトルの要素に最も近い
テーブルの要素を見つける。

```
IppStatus ippsFindNearest_16u(const Ipp16u* pVals, Ipp16u* pOutVals,
    int* pOutIndexes, int len, const Ipp16u *pTable, int tblLen);
```

引数

pVals	基準値を含むベクトルへのポインタ。
pOutVals	出力ベクトルへのポインタ。
pOutIndexes	出力インデックスを格納する配列へのポインタ。
len	入力ベクトル内の要素の数。
pTable	検索するテーブルへのポインタ。
tbllen	テーブル内の要素の数。

説明

関数 ippsFindNearest は、ipps.h ファイルで宣言される。この関数は、入力ベクトル *pVals* の基準となる要素に最も近い要素をテーブル *pTable* 内で検索する。得られた要素とそのインデックスは、それぞれ *pOutVals* と *pOutIndexes* に格納される。

テーブルの要素は、*pTable*[*i*] ≤*p Table*[*i+1*] の条件を満たしている必要がある。この関数は、次のような距離の判定法を使用して、*pVal*[*k*] に最も近いテーブル要素を決定する。

min(|*pVals*[*k*]-*pTable*[i]|)

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsSizeErr	エラー。*tbllen* または *len* がゼロ以下。

ビタビ・デコーダ関数

この項では、V.34 レシーバ内でビタビ・デコーディング演算を実行する関数について説明する。エンコーディングは、9.6.3 項で説明する ITU-T 勧告 V.34 のアルゴリズムに従って実行される（[ITUV34] を参照）。

GetVarPointDV

受け取った点に最も近い点に関する情報を
配列に格納する。

```
IppStatus ippsGetVarPointDV_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
    Ipp16sc* pVariantPoint, const Ipp8u* pLabel, int state);
```

引数

pSrc	基準点へのポインタ（9:7 形式）。
pDst	基準点に最も近い左下の複素数点へのポインタ（9:7 形式）。
pVariantPoint	点に関する情報が格納される配列へのポインタ。
pLabel	ラベルを格納するテーブルへのポインタ。
state	たたみ込みコーダのステート数。

説明

関数 ippsGetVarPointDV は、ipps.h ファイルで宣言される。この関数は、基準点 *pSrc* に最も近い 2D 複素数点に関する情報を、指定された配列 *pVariantPoint* に格納する。この情報には、オフセット・テーブルから得られる対応するラベルと

計算されたエラーが含まれる。可能なステートの数は、16、32、64 である。配列内の点の数は、ステートの数によって異なる。*state*=16 の場合は、配列内で 4 個の点が参照される。*state*=32 または 64 の場合は、8 個の点が参照される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。

CalcStatesDV

ビタビ・デコーダのステートを計算する。

```
IppStatus ippsCalcStatesDV_16sc(const Ipp16u* pathError,
    const Ipp8u* pNextState, Ipp16u* pBranchError,
    const Ipp16s* pCurrentSubsetPoint, Ipp16s* pPathTable,
    int state, int presentIndex);
```

引数

pPathError	パス・エラー・メトリックのテーブルへのポインタ。
pNextState	次のステート・テーブルへのポインタ。
pBranchError	分岐エラー・テーブルへのポインタ。
pCurrentSubsetPoint	現在の 4D サブセットへのポインタ。
pPathTable	ビタビ・パス・テーブルへのポインタ。
state	たたみ込みエンコーダのステート数。
presentIndex	ビタビ・パス・テーブルの開始インデックス。

説明

関数 ippsCalcStatesDV は、ipps.h ファイルで宣言される。この関数は、ビタビ・デコーダの可能なステートを計算し、累算エラー *pPathError* のテーブルとアクティブなビタビ・パス・テーブル *pPathTable* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。

BuildSymblTableDV4D

可能な 4D シンボルに関する情報を配列に
格納する。

```
IppStatus ippsBuildSymblTableDV4D_16sc(const Ipp16sc* pVariantPoint,
    Ipp16sc* pCurrentSubsetPoint, int state, int bitInversion);
```

引数

pVariantPoint	可能な 2D 点の配列へのポインタ。
pCurrentSubsetPoint	可能な 4D シンボルに関する情報を格納する配列へのポインタ。
state	たたみ込みエンコーダのステート数。
bitInversion	ビット反転。

説明

関数 ippsBuildSymblTableDV4D は、ipps.h ファイルで宣言される。この関数は、可能な 4D シンボルに関する情報を配列 *pCurrentSubsetPoint* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。

UpdatePathMetricsDV

最小パス・メトリックを持つステートを
検索する。

```
IppStatus ippsUpdatePathMetricsDV_16u(Ipp16u* pBranchError, Ipp16u*
    pMinPathError, Ipp8u* pMinSost, Ipp16u* pPathError, int state);
```

引数

pBranchError	分岐エラー・テーブルへのポインタ。
pMinPathError	現在の最小パス・エラー・メトリックへのポインタ。

pMinSost	最小メトリックを持つステートの数へのポインタ。
pPathError	累算パス・エラーのテーブルへのポインタ。
state	たたみ込みエンコーダのステート数。

説明

関数 ippsBuildSymblTableDV4D は、ipps.h ファイルで宣言される。この関数は、最小パス・エラー・メトリックを持つステートを検索し、ステートの数を *pMinSost* に格納する。すべてのステートについて、パス・メトリックから最小メトリックが減算される。すべてのステートの分岐エラーは、ビタビ・デコーディングの次のステップの必要に応じて、無限に大きい値に設定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。

窓（Window）関数

本章では、信号処理で使用する窓関数をいくつか説明する。窓関数とは、後に続く分析の特性を改善するために信号を乗算する一種の算術関数である。窓関数は、FFT ベースのスペクトル分析で使用される。

窓関数の概要

インテル® IPP では、窓関数サンプルを生成するために、次の関数がある。

- Bartlett（バートレット）窓関数
- Blackman（ブラックマン）ファミリの窓関数
- Hamming（ハミング）窓関数
- Hann（ハニング）窓関数
- Kaiser（カイザー）窓関数

これらの関数は、窓関数サンプルを生成し、それらを既存の信号に掛ける。窓関数サンプル自体を入手するには、窓関数を呼び出す前に、ベクトル引数を単位ベクトルに初期化する。

異なる信号フレームに同じ窓関数を複数回掛けたい場合は、すべての要素を 1.0 にセットしてベクトルに対する窓関数の 1 つ（例えば ippsWinHann）を呼び出して、最初に窓関数を計算する。この後、新しい入力サンプルのセットが入手可能になる

たびに、ベクトル乗算関数の 1 つ（例えば ippsMul）を使用して窓関数を信号に掛ける。この方法だと、窓関数サンプルを繰り返し計算する手間を省ける。この例を、例 5-16 に示す。

例 5-16　信号の多数のフレームへの窓関数の適用と FFT の入手

```
void multiFrameWin( void ) {
      Ipp32f win[LEN], x[LEN], X[LEN];
      IppsFFTSpec_R_32f* ctx;
      ippsSet_32f( 1, win, LEN );
      ippsWinHann_32f_I( win, LEN );
      /// ... initialize FFT context
      while(1 ){
         /// ... get x signal
         ///
         ippsMul_32f_I( win, x, LEN );
         ippsFFTFwd_RToPack_32f( x, X, ctx, 0 );
      }
}
```

関連項目

窓関数の詳細は、[Jac89] 7.3 項「Windows in Spectrum Analysis」、[Jac89] 9.1 項「Window-Function Technique」、[Mit93] 16-2 項「Fourier Analysis of Finite-Time Signals」を参照のこと。
これらの参考資料の詳細については、本書の最後にある「参考文献」を参照のこと。

WinBartlett

ベクトルに Bartlett（バートレット）窓関数を掛ける。

```
IppStatus ippsWinBartlett_16s(const Ipp16s* pSrc, Ipp16s* pDst,
   int len);
IppStatus ippsWinBartlett_32f(const Ipp32f* pSrc, Ipp32f* pDst,
   int len);
```

```
IppStatus ippsWinBartlett_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    int len);
IppStatus ippsWinBartlett_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len);
IppStatus ippsWinBartlett_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len);
IppStatus ippsWinBartlett_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int len);
IppStatus ippsWinBartlett_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBartlett_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBartlett_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBartlett_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBartlett_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBartlett_64fc_I(Ipp64fc* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsWinBartlett` は、`ipps.h` ファイルで宣言される。この関数は、ベクトル *pSrc* に Bartlett（バートレット）窓関数を掛け、その結果を *pDst* に格納する。

関数 `ippsWinBartlett` はインプレース演算で、*pSrcDst* に Bartlett（バートレット）窓関数を掛け、その結果を *pSrcDst* に格納する。

複素数型は、ベクトルの実数部と虚数部のいずれにも同じ窓関数を掛ける。

Bartlett（バートレット）窓関数は、次のように定義される。

$$w_{bartlett}(n) = \begin{cases} \dfrac{2n}{len-1}, & 0 \le n \le \dfrac{len-1}{2} \\ 2 - \dfrac{2n}{len-1}, & \dfrac{len-1}{2} < n \le len-1 \end{cases}$$

例 5-17 は、関数 `ippsWinBartlett_32f_I` の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* が 3 より小さい。

例 5-17 ippsWinBartlett 関数の使用例

```
void bartlett(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBartlett_32f_I(x, 8);
      printf_32f("bartlett (half) =", x, 4, ippStsNoErr);
}
Output:
      bartlett (half) =  0.000000 0.285714 0.571429 0.857143
Matlab* Analog:
      >> b = bartlett(8); b(1:4)'
```

WinBlackman

ベクトルに Blackman（ブラックマン）窓関数を掛ける。

```
IppStatus ippsWinBlackmanQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst,
   int len, int alphaQ15);
IppStatus ippsWinBlackmanQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
   int len, int alphaQ15);
IppStatus ippsWinBlackman_16s(const Ipp16s* pSrc, Ipp16s* pDst,
   int len, float alpha);
IppStatus ippsWinBlackman_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
   int len, float alpha);
IppStatus ippsWinBlackman_32f(const Ipp32f* pSrc, Ipp32f* pDst,
   int len, float alpha);
IppStatus ippsWinBlackman_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
   int len, float alpha);
```

```
IppStatus ippsWinBlackman_64f(Ipp64f* pSrc, Ipp64f* pDst, int len,
    double alpha);
IppStatus ippsWinBlackman_64fc(Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    double alpha);
IppStatus ippsWinBlackmanQ15_16s_I(Ipp16s* pSrcDst, int len,
    int alphaQ15);
IppStatus ippsWinBlackmanQ15_16s_ISfs(Ipp16s* pSrcDst, int len,
    int alphaQ15, int scaleFactor);
IppStatus ippsWinBlackmanQ15_16sc_I(Ipp16sc* pSrcDst, int len,
    int alphaQ15);
IppStatus ippsWinBlackman_16s_I(Ipp16s* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_16sc_I(Ipp16sc* pSrcDst, int len,
    float alpha);
IppStatus ippsWinBlackman_32f_I(Ipp32f* pSrcDst, int len,
    float alpha);
IppStatus ippsWinBlackman_32fc_I(Ipp32fc* pSrcDst, int len,
    float alpha);
IppStatus ippsWinBlackman_64f_I(Ipp64f* pSrcDst, int len,
    double alpha);
IppStatus ippsWinBlackman_64fc_I(Ipp64fc* pSrcDst, int len,
    double alpha);
IppStatus ippsWinBlackmanStd_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len);
IppStatus ippsWinBlackmanStd_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len);
IppStatus ippsWinBlackmanStd_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len);
IppStatus ippsWinBlackmanStd_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len);
IppStatus ippsWinBlackmanStd_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    int len);
IppStatus ippsWinBlackmanStd_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int len);
IppStatus ippsWinBlackmanStd_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len);
```

```
IppStatus ippsWinBlackmanOpt_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len);
IppStatus ippsWinBlackmanOpt_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len);
IppStatus ippsWinBlackmanOpt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int len);
IppStatus ippsWinBlackmanOpt_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    int len);
IppStatus ippsWinBlackmanOpt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int len);
IppStatus ippsWinBlackmanOpt_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64fc_I(Ipp64fc* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
alpha	Blackman window 方程式に関連付けられた調節可能パラメータ。
alphaQ15	*alpha* のスケーリング・バージョン。*scaleFactor* の値は 15。
len	ベクトル内の要素の数。

説明

ippsWinBlackman ファミリに属する関数は、ipps.h ファイルで宣言される。これらの関数は、ベクトル *pSrc* に Blackman（ブラックマン）窓関数を掛け、その結果を *pDst* に格納する。

関数 ippsWinBlackman ファミリに属する関数はインプレース演算で、ベクトル *pSrcDst* に Blackman（ブラックマン）窓関数を掛け、その結果を *pSrcDst* に格納する。

複素数型は、ベクトルの実数部と虚数部のいずれにも同じ窓関数を掛ける。Blackman（ブラックマン）ファミリに属する窓関数は、次のように定義される。

ippsWinBlackman。関数 ippsWinBlackman を使用すると、アプリケーションで *alpha* を指定できる。Blackman（ブラックマン）窓関数は、次のように定義される。

$$w_{blackman}(n) = \frac{alpha+1}{2} - 0.5\cos\left(\frac{2\pi n}{len-1}\right) - \frac{alpha}{2}\cos\left(\frac{4\pi n}{len-1}\right)$$

ippsWinBlackmanQ15。関数 ippsWinBlackmanQ15 は、*alphaQ15*（スケーリング係数は 15）を指定して、ベクトルに Blackman（ブラックマン）窓関数を掛ける。

ippsWinBlackmanStd。標準的な Blackman（ブラックマン）窓関数は、関数 ippsWinBlackmanStd により提供される。この関数は、次に示す *alpha* の標準値を指定して、ベクトルに Blackman（ブラックマン）窓関数を掛けるだけである。

alpha = -0.16

ippsWinBlackmanOpt。関数 ippsWinBlackmanOpt は、次に示す *alpha* の最適値を指定してベクトルに Blackman（ブラックマン）窓関数を掛けると、30dB / オクターブのロールオフを持つ変形 Window を提供する。

$$alpha = -\frac{0.5}{1 + \cos\frac{2\pi}{len-1}}$$

len の最小値は 4 である。*len* が大きくなると、最適な *alpha* は、漸近値 *alpha* に漸近的に収束する。アプリケーションは *alpha* の漸近値として次の値を使用できる。

alpha = -0.25

例 5-18 は、関数 ippsWinBlackmanStd_32f_I の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* の値が関数 ippsWinBlackmanOpt に対して 4 未満であり、ファミリに属するその他すべての関数に対して 3 未満である。

例 5-18 ippsWinBlackmanStd 関数の使用例

```
void blackman(void) {
      Ipp32f x[8];
      ippsSet_32f(1, x, 8);
      ippsWinBlackmanStd_32f_I(x, 8);
      printf_32f("blackman (half) =", x, 4, ippStsNoErr);
}
Output:
      blackman(half) =  0.000000 0.090453 0.459183 0.920364
Matlab* Analog:
      >> b = blackman(8)'; b(1:4)
```

WinHamming

ベクトルに Hamming（ハミング）
窓関数を掛ける。

```
IppStatus ippsWinHamming_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
   len);
IppStatus ippsWinHamming_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
   len);
IppStatus ippsWinHamming_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
   len);
IppStatus ippsWinHamming_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
   len);
IppStatus ippsWinHamming_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
   len);
IppStatus ippsWinHamming_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
   len);
IppStatus ippsWinHamming_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHamming_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHamming_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHamming_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHamming_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHamming_64fc_I(Ipp64fc* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsWinHamming は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* に Hamming（ハミング）窓関数を掛け、その結果を *pDst* に格納する。

関数 ippsWinHamming はインプレース演算で、ベクトル *pSrcDst* に Hamming（ハミング）窓関数を掛け、その結果を *pSrcDst* に格納する。

複素数型は、ベクトルの実数部と虚数部のいずれにも同じ窓関数を掛ける。Hamming（ハミング）窓関数は、次のように定義される。

$$w_{hamming}(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{len - 1}\right)$$

例 5-19 は、関数 ippsWinHamming_32f_I の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* が 3 より小さい。

例 5-19　ippsWinHamming 関数の使用例

```
void hamming(void) {
        Ipp32f x[8];
        ippsSet_32f(1, x, 8);
        ippsWinHamming_32f_I(x, 8);
        printf_32f("hamming(half) =", x, 4, ippStsNoErr);
}
Output:
        hamming(half) =  0.080000 0.253195 0.642360 0.954446
Matlab* Analog:
        >> b = hamming(8); b(1:4)'
```

WinHann

ベクトルに Hann（ハニング）窓関数を掛ける。

```
IppStatus ippsWinHann_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHann_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    len);
IppStatus ippsWinHann_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHann_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
    len);
IppStatus ippsWinHann_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHann_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
    len);
IppStatus ippsWinHann_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHann_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHann_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHann_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHann_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHann_64fc_I(Ipp64fc* pSrcDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsWinHann` は、`ipps.h` ファイルで宣言される。この関数は、ベクトル *pSrc* に Hann（ハニング）窓関数を掛け、その結果を *pDst* に格納する。

関数 `ippsWinHann` はインプレース演算で、ベクトル *pSrcDst* に Hann 窓関数を掛け、その結果を *pSrcDst* に格納する。

複素数型は、ベクトルの実数部と虚数部のいずれにも同じ窓関数を掛ける。Hann 窓関数は、次のように定義される。

$$w_{hann}(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{len - 1}\right)$$

intel®

例 5-20 は、関数 ippsWinHann_32f_I の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* が 3 より小さい。

例 5-20　ippsWinHann 関数の使用例

```
void hann(void) {
        Ipp32f x[8];
        ippsSet_32f(1, x, 8);
        ippsWinHann_32f_I(x, 8);
        printf_32f("hann(half) =", x, 4, ippStsNoErr);
}
Output:
        hann(half) =  0.000000 0.188255 0.611260 0.950484
Matlab* Analog:
        >> N = 8; n = 0:N-1; 0.5*(1-cos(2*pi*n/(N-1)))
```

WinKaiser

ベクトルに Kaiser（カイザー）窓関数を掛ける。

```
IppStatus ippsWinKaiser_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    float alpha);
IppStatus ippsWinKaiser_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    float alpha);
IppStatus ippsWinKaiser_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    float alpha);
IppStatus ippsWinKaiser_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    len, float alpha);
IppStatus ippsWinKaiser_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
    len, float alpha);
```

```
IppStatus ippsWinKaiser_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
    len, float alpha);
IppStatus ippsWinKaiserQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
    len, int alphaQ15);
IppStatus ippsWinKaiserQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
    len, int alphaQ15);
IppStatus ippsWinKaiser_16s_I(Ipp16s* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32f_I(Ipp32f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_64f_I(Ipp64f* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_16sc_I(Ipp16sc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_32fc_I(Ipp32fc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiser_64fc_I(Ipp64fc* pSrcDst, int len, float alpha);
IppStatus ippsWinKaiserQ15_16s_I(Ipp16s* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinKaiserQ15_16sc_I(Ipp16sc* pSrcDst, int len, int alphaQ15);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション・ベクトルへのポインタ。
alpha	Kaiser（カイザー）窓方程式に関連付けられた調節可能パラメータ。
alphaQ15	*alpha* のスケーリング・バージョン。*ScaleFactor* の値は 15。
len	ベクトル内の要素の数。

説明

関数 `ippsWinKaiser` は、`ipps.h` ファイルで宣言される。この関数は、ベクトル *pSrc* に Kaiser（カイザー）窓関数を掛け、その結果を *pDst* に格納する。

関数 `ippsWinKaiser` はインプレース演算で、ベクトル *pSrcDst* に Kaiser（カイザー）窓関数を掛け、その結果を *pSrcDst* に格納する。

ippsWinKaiser。関数 `ippsWinKaiser` を使用すると、アプリケーションで *alpha* を指定できる。この関数は、複素数ベクトルの実数部と虚数部のいずれにも同じ窓関数を掛ける。Kaiser（カイザー）窓関数は、次のように定義される。

$$w_{kaiser}(n) = \frac{I_0\left(alpha\sqrt{\left(\frac{len-1}{2}\right)^2 - \left(n - \left(\frac{len-1}{2}\right)\right)^2}\right)}{I_0\left(alpha\left(\frac{len-1}{2}\right)\right)}$$

$I_0()$ は、変形されたゼロ次の第 1 種 Bessel 関数である。

ippsWinKaiserQ15。関数 ippsWinKaiserQ15 は、*alphaQ15*（スケーリング係数は 15）を指定して、ベクトルに Kaiser（カイザー）窓関数を掛ける。

例 5-21 は、関数 ippsWinKaiser_32f_I の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst*、*pSrc* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* が 1 より小さい。
ippStsHugeWinErr	エラー。Kaiser（カイザー）窓 が大きすぎる。

例 5-21　ippsWinKaiser 関数の使用例

```
void kaiser(void) {
     Ipp32f x[8];
     IppStatus st;
     ippsSet_32f(1, x, 8);
     st = ippsWinKaiser_32f_I( x, 8, 1.0f );
     printf_32f("kaiser(half) =", x, 4, ippStsNoErr);
}
Output:
   kaiser(half) =  0.135534 0.429046 0.755146 0.970290
Matlab* Analog:
    >> kaiser(8,7/2)'
```

統計関数

この項では、ベクトル測度値（最大、最小、平均、標準偏差）を計算するインテル® IPP 関数について説明する。

Sum

ベクトルの要素の合計を計算する。

```
IppStatus ippsSum_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum,
    IppHintAlgorithm hint);
IppStatus ippsSum_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pSum,
    IppHintAlgorithm hint);
IppStatus ippsSum_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSum_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pSum);
IppStatus ippsSum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pSum,
    int scaleFactor);
IppStatus ippsSum_32s_Sfs(const Ipp32s* pSrc, int len, Ipp32s* pSum,
    int scaleFactor);
IppStatus ippsSum_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pSum,
    int scaleFactor);
IppStatus ippsSum_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pSum,
    int scaleFactor);
IppStatus ippsSum_16sc32sc_Sfs(const Ipp16sc* pSrc, int len, Ipp32sc* pSum,
    int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pSum	出力結果へのポインタ。
len	ベクトル内の要素の数。
hint	特定のコードの使用を指定する。`hint` 引数の値については、「flag 引数と hint 引数」を参照のこと。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 `ippsSum` は、`ipps.h` ファイルで宣言される。この関数は、ベクトル *pSrc* の要素の合計を計算し、その結果を *pSum* に格納する。

intel®

pSrc の要素の合計は、次の式で定義される。

$$sum = \sum_{n=0}^{len-1} pSrc[n]$$

hint 引数は、計算を高速化する（精度は低下する）か、または精度を向上する（速度は低下する）、特定のコードの使用を指定する。

整数の合計を計算すると、出力結果がデータ範囲を超え、飽和する場合がある。正確な結果を得るには、スケール係数を使用する。スケーリングは、*scaleFactor* の値に従って実行される。

例 5-22 は、関数 ippsSum の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSum* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-22　ippsSum 関数の使用例

```
void sum(void) {
      Ipp16s x[4] = {-32768, 32767, 32767, 32767}, sm;
      ippsSum_16s_Sfs(x, 4, &sm, 1);
      printf_16s("sum =", &sm, 1, ippStsNoErr);
}
Output:
      sum = 32766
Matlab* Analog:
      >> x = [-32768, 32767, 32767, 32767]; sum(x)/2
```

Max

ベクトルの最大値を返す。

```
IppStatus ippsMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax);
IppStatus ippsMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax);
IppStatus ippsMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pMax	出力結果へのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsMax は、ipps.h ファイルで宣言される。この関数は、入力ベクトル *pSrc* の最大値を返し、その結果を *pMax* に格納する。

例 5-23 は、関数 ippsMax_32f の使用例を示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMax* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

MaxIndx

ベクトルの最大値と、最大要素のインデックスを返す。

```
IppStatus ippsMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax,
    int* pIndx);
IppStatus ippsMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax,
    int* pIndx);
IppStatus ippsMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax,
    int* pIndx);
IppStatus ippsMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax,
    int* pIndx);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pMax	出力結果へのポインタ。
len	ベクトル内の要素の数。
pIndx	最大要素のインデックス値へのポインタ。

intel®

説明

関数 ippsMaxIndx は、ipps.h ファイルで宣言される。この関数は、入力ベクトル *pSrc* の最大値を返し、その結果を *pMax* に格納する。*pIndx* が NULL ポインタでない場合、この関数は、最大要素のインデックスを返し、それを *pIndx* に格納する。同じ最大値の要素がいくつかある場合は、先頭のインデックスが返される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMax* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Min

ベクトルの最小値を返す。

```
IppStatus ippsMin_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin);
IppStatus ippsMin_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin);
IppStatus ippsMin_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pMin	出力結果へのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsMin は、ipps.h ファイルで宣言される。この関数は、入力ベクトル *pSrc* の最小値を返し、その結果を *pMin* に格納する。

例 5-23 は、関数 ippsMin_32f の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMin* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

MinIndx

ベクトルの最小値と、最小要素のインデックスを返す。

```
IppStatus ippsMinIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin,
    int* pIndx);
IppStatus ippsMinIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin,
    int* pIndx);
IppStatus ippsMinIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin,
    int* pIndx);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pMin	出力結果へのポインタ。
len	ベクトル内の要素の数。
pIndx	最小要素のインデックス値へのポインタ。

説明

関数 ippsMinIndx は、ipps.h ファイルで宣言される。この関数は、入力ベクトル *pSrc* の最小値を返し、その結果を *pMin* に格納する。*PIndx* が NULL ポインタでない場合、この関数は、最小要素のインデックスを返し、それを *pIndx* に格納する。同じ最小値の要素がいくつかある場合は、先頭のインデックスが返される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMin* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-23　ippsMin 関数と ippsMax 関数の使用例

```
void minmax(void) {
        Ipp32f *x = ippsMalloc_32f(1000), minmax[2];
        int i;
        for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
        ippsMin_32f(x, 1000, &minmax[0]);
        ippsMax_32f(x, 1000, &minmax[1]);
        printf_32f("min max =", minmax, 2, ippStsNoErr);
        ippsFree(x);
}
Output:
        min max = 0.000855 0.999695
Matlab* Analog:
        >> x = rand(1,1000); min(x), max(x)
```

MinMax

ベクトルの最大値および最小値を返す。

```
IppStatus ippsMinMax_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin, Ipp8u*
    pMax);
IppStatus ippsMinMax_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin,
    Ipp16u* pMax);
IppStatus ippsMinMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin,
    Ipp16s* pMax);
IppStatus ippsMinMax_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin,
    Ipp32u* pMax);
IppStatus ippsMinMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin,
    Ipp32s* pMax);
IppStatus ippsMinMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin,
    Ipp32f* pMax);
IppStatus ippsMinMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin,
    Ipp64f* pMax);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pMin	最小値へのポインタ。

pMax	最大値へのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsMinMax は、ipps.h ファイルで宣言される。この関数は、入力ベクトル *pSrc* の最小値と最大値を返し、その結果をそれぞれ *pMin* と *pMax* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMin* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

MinMaxIndx

ベクトルの最大値、最小値、および対応する要素のインデックスを返す。

```
IppStatus ippsMinMaxIndx_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin,
    int* pMinIndx, Ipp8u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin,
    int* pMinIndx, Ipp16u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin,
    int* pMinIndx, Ipp16s* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin,
    int* pMinIndx, Ipp32u* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin,
    int* pMinIndx, Ipp32s* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin,
    int* pMinIndx, Ipp32f* pMax, int* pMaxIndx);
IppStatus ippsMinMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin,
    int* pMinIndx, Ipp64f* pMax, int* pMaxIndx);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pMin	最小値へのポインタ。
pMax	最大値へのポインタ。
len	ベクトル内の要素の数。

pMinIndx	最小要素のインデックス値へのポインタ。
pMaxIndx	最大要素のインデックス値へのポインタ。

説明

関数 ippsMinMaxIndx は、ipps.h ファイルで宣言される。この関数は、入力ベクトル *pSrc* の最小値と最大値を返し、その結果をそれぞれ *pMin* と *pMax* に格納する。また、関数は最小要素と最大要素のインデックスを返し、それぞれ *pMinIndx* と *pMaxIndx* に格納する。同じ最小値または最大値の要素がいくつかある場合は、先頭のインデックスが返される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Mean

ベクトルの平均値を計算する。

```
IppStatus ippsMean_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean,
    IppHintAlgorithm hint);
IppStatus ippsMean_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pMean,
    IppHintAlgorithm hint);
IppStatus ippsMean_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean);
IppStatus ippsMean_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pMean);
IppStatus ippsMean_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean,
    int scaleFactor);
IppStatus ippsMean_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pMean,
    int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pMean	出力結果へのポインタ。
len	ベクトル内の要素の数。
hint	特定のコードの使用を指定する。*hint* 引数の値については、「flag 引数と hint 引数」を参照のこと。

scaleFactor 第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsMean は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* の平均を計算し、その結果を *pMean* に格納する。*pSrc* の平均は、次の式で定義される。

$$mean = \frac{1}{len}\sum_{n=0}^{len-1} pSrc[n]$$

hint 引数は、計算を高速化する（精度は低下する）か、または精度を向上する（速度は低下する）、特定のコードの使用を指定する。

例 5-24 は、関数 ippsMean_32f の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMean* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-24 ippsMean 関数の使用例

```
void mean(void) {
      Ipp32f *x = ippsMalloc_32f(1000), mean;
      int i;
      for(i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsMean_32f(x, 1000, &mean, ippAlgHintFast);
      printf_32f("mean =", &mean, 1, ippStsNoErr);
      ippsFree(x);
}
Output:
      mean = 0.492591
Matlab* Analog:
      >> x = rand(1,1000); mean(x)
```

StdDev

ベクトルの標準偏差値を計算する。

```
IppStatus ippsStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pStdDev,
    IppHintAlgorithm hint);
IppStatus ippsStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pStdDev);
IppStatus ippsStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len,
    Ipp32s* pStdDev, int scaleFactor);
IppStatus ippsStdDev_16s_Sfs(const Ipp16s* pSrc, int len,
    Ipp16s* pStdDev, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pStdDev	出力結果へのポインタ。
len	ベクトル内の要素の数。
hint	特定のコードの使用を指定する。hint 引数の値については、「flag 引数と hint 引数」を参照のこと。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsStdDev は、ipps.h ファイルで宣言される。この関数は、入力ベクトル *pSrc* の標準偏差を計算し、その結果を *pStdDev* に格納する。ベクトルの長さは 2 以上にする必要がある。*pSrc* の標準偏差は、次のバイアスなしの推定式で定義される。

$$stdev = \sqrt{\frac{\sum_{n=0}^{len-1}(pSrc[n] - mean(pSrc))^2}{len - 1}}$$

hint 引数は、計算を高速化する（精度は低下する）か、または精度を向上する（速度は低下する）、特定のコードの使用を指定する。

例 5-25 は、関数 ippsStdDev_32f の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pStdDev* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* が 1 以下。

例 5-25 ippsStdDev 関数の使用例

```
void stdev(void) {
      Ipp32f *x = ippsMalloc_32f(1000), stdev;
      int i;
      for (i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
      ippsStdDev_32f(x, 1000, &stdev, ippAlgHintFast);
      printf_32f("stdev =", &stdev, 1, ippStsNoErr);
      ippsFree(x);
}
Output:
         stdev = 0.286813
Matlab* Analog:
         >> x = rand(1,1000); std(x)
```

Norm

ベクトルのノルム C、L1、または L2 を
計算する。

```
IppStatus ippsNorm_Inf_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_32fc32f(const Ipp32fc* pSrc, int len,
   Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64fc64f(const Ipp64fc* pSrc, int len,
   Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32s_Sfs(const Ipp16s* pSrc, int len,
   Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNorm_L1_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L1_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_32fc64f(const Ipp32fc* pSrc, int len,
   Ipp64f* pNorm);
IppStatus ippsNorm_L1_64fc64f(const Ipp64fc* pSrc, int len,
   Ipp64f* pNorm);
```

```
IppStatus ippsNorm_L1_16s32s_Sfs(const Ipp16s* pSrc, int len,
    Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNorm_L2_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_32fc64f(const Ipp32fc* pSrc, int len,
    Ipp64f* pNorm);
IppStatus ippsNorm_L2_64fc64f(const Ipp64fc* pSrc, int len,
    Ipp64f* pNorm);
IppStatus ippsNorm_L2_16s32s_Sfs(const Ipp16s* pSrc, int len,
    Ipp32s* pNorm, int scaleFactor);
```

引数

pSrc	ソース・ベクトル *pSrc* へのポインタ。
pNorm	出力結果へのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsNorm は、ipps.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc* のノルム C、L1、または L2 を計算し、その結果を *pNorm* に格納する。

ippsNorm_Inf。関数 ippsNorm_Inf は、次の式で定義される C ノルムを計算する。

$$Norm_C = \max_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L1。関数 ippsNorm_L1 は、次の式で定義される L1 ノルムを計算する。

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc[n]|$$

ippsNorm_L2。関数 ippsNorm_L2 は、次の式で定義される L2 ノルムを計算する。

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc[n]|^2}$$

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値のスケーリングを実行する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pNorm* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

NormDiff

2 つのベクトルの差のノルム C、L1、
または L2 を計算する。

```
IppStatus ippsNormDiff_Inf_32f(const Ipp32f* pSrc1, const Ipp32f*
    pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64f(const Ipp64f* pSrc1, const Ipp64f*
    pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32f(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_32fc32f(const Ipp32fc* pSrc1, const Ipp32fc*
    pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_Inf_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc*
    pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_Inf_16s32s_Sfs(const Ipp16s* pSrc1, const
    Ipp16s* pSrc2, int len, Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNormDiff_L1_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_16s32f(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L1_32fc64f(const Ipp32fc* pSrc1, const Ipp32fc*
    pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc*
    pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L1_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32s* pNorm, int scaleFactor);
IppStatus ippsNormDiff_L2_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    int len, Ipp64f* pNorm);
```

```
IppStatus ippsNormDiff_L2_16s32f(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32f* pNorm);
IppStatus ippsNormDiff_L2_32fc64f(const Ipp32fc* pSrc1, const Ipp32fc*
    pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc*
    pSrc2, int len, Ipp64f* pNorm);
IppStatus ippsNormDiff_L2_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32s* pNorm, int scaleFactor);
```

引数

pSrc1, pSrc2	2 つのソース・ベクトルへのポインタ。*pSrc2* は NULL にできる。
pNorm	出力結果へのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsNorm は、ipps.h ファイルで宣言される。この関数は、ソース・ベクトルの差のノルム C、L1、または L2 を計算し、その結果を *pNorm* に格納する。

ippsNormDiff_Inf。関数 ippsNormDiff_Inf は、次の式で定義される C ノルムを計算する。

$$Norm_{Inf} = \max_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L1。関数 ippsNormDiff_L1 は、次の式で定義される L1 ノルムを計算する。

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

ippsNormDiff_L2。関数 ippsNormDiff_L2 は、次の式で定義される L2 ノルムを計算する。

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|^2}$$

Sfs サフィックスの付いた関数では、*scaleFactor* 値に従って結果の値のスケーリングを実行する。

例 5-26 は、関数 ippsNormDiff の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2* または *pNorm* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-26 ippsNorm 関数の使用例

```
int norm( void ) {
      Ipp16s x[LEN];
      Ipp32f Norm[3];
      IppStatus st;
      int i;
      for( i=0; i<LEN; ++i ) x[i] = (Ipp16s)rand();
      ippsNormDiff_Inf_16s32f( x, 0, LEN, Norm );
      ippsNormDiff_L1_16s32f( x, 0, LEN, Norm+1 );
      st = ippsNormDiff_L2_16s32f( x, 0, LEN, Norm+2 );
      printf_32f("Norm (oo,L1,L2) =", Norm, 3, st );
      return Norm[2] <= Norm[1] && Norm[1] <= LEN*Norm[0];
}
Output:
      Norm (oo,L1,L2) =  31993.000000 1526460.000000 180270.781250
Matlab* analog:
      >> x = 32767*rand(1,100);norm(x,inf),norm(x,1),norm(x,2)
```

DotProd

2 つのベクトルの内積を計算する。

```
IppStatus ippsDotProd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
   int len, Ipp32f* pDp);
```

```
IppStatus ippsDotProd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
   int len, Ipp32fc* pDp);
IppStatus ippsDotProd_32f32fc(const Ipp32f* pSrc1, const Ipp32fc*
   pSrc2, int len, Ipp32fc* pDp);
IppStatus ippsDotProd_32f64f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
   int len, Ipp64f* pDp);
IppStatus ippsDotProd_32fc64fc(const Ipp32fc* pSrc1, const Ipp32fc*
   pSrc2, int len, Ipp64fc* pDp);
IppStatus ippsDotProd_32f32fc64fc(const Ipp32f* pSrc1,
   const Ipp32fc* pSrc2, int len, Ipp64fc* pDp);
IppStatus ippsDotProd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
   int len, Ipp64f* pDp);
IppStatus ippsDotProd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
   int len, Ipp64fc* pDp);
IppStatus ippsDotProd_64f64fc(const Ipp64f* pSrc1, const Ipp64fc*
   pSrc2, int len, Ipp64fc* pDp);
IppStatus ippsDotProd_16s64s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
   int len, Ipp64s* pDp);
IppStatus ippsDotProd_16sc64sc(const Ipp16sc* pSrc1, const Ipp16sc*
   pSrc2, int len, Ipp64sc* pDp);
IppStatus ippsDotProd_16s16sc64sc(const Ipp16s* pSrc1, const Ipp16sc*
   pSrc2, int len, Ipp64sc* pDp);
IppStatus ippsDotProd_16s32f(const Ipp16s* pSrc1, const Ipp16s*
   pSrc2, int len, Ipp32f* pDp);
IppStatus ippsDotProd_16sc32fc(const Ipp16sc* pSrc1, const Ipp16sc*
   pSrc2, int len, Ipp32fc* pDp);
IppStatus ippsDotProd_16s16sc32fc(const Ipp16s* pSrc1, const Ipp16sc*
   pSrc2, int len, Ipp32fc* pDp);
IppStatus ippsDotProd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
   pSrc2, int len, Ipp16s* pDp, int scaleFactor);
IppStatus ippsDotProd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc*
   pSrc2, int len, Ipp16sc* pDp, int scaleFactor);
IppStatus ippsDotProd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s*
   pSrc2, int len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc*
   pSrc2, int len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
   pSrc2, int len, Ipp32s* pDp, int scaleFactor);
IppStatus ippsDotProd_16s16sc32sc_Sfs(const Ipp16s* pSrc1, const
   Ipp16sc* pSrc2, int len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16s32s32s_Sfs(const Ipp16s* pSrc1, const Ipp32s*
   pSrc2, int len, Ipp32s* pDp, int scaleFactor);
```

```
IppStatus ippsDotProd_16s16sc_Sfs(const Ipp16s* pSrc1, const Ipp16sc*
    pSrc2, int len, Ipp16sc* pDp, int scaleFactor);
IppStatus ippsDotProd_16sc32sc_Sfs(const Ipp16sc* pSrc1, const
    Ipp16sc* pSrc2, int len, Ipp32sc* pDp, int scaleFactor);
IppStatus ippsDotProd_32s32sc_Sfs(const Ipp32s* pSrc1, const Ipp32sc*
    pSrc2, int len, Ipp32sc* pDp, int scaleFactor);
```

引数

pSrc1	内積の値の計算に使用する最初のベクトルへのポインタ。
pSrc2	内積の値の計算に使用する 2 番目のベクトルへのポインタ。
pDp	出力結果へのポインタ。
len	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDotProd は、ipps.h ファイルで宣言される。この関数は、2 つのベクトル *pSrc1* と *pSrc2* の内積（スカラー値）を計算し、その結果を *pDp* に格納する。

計算は、次のように行われる。

$$dp = \sum_{n=0}^{len-1} pSrc1[n] \cdot pSrc2[n]$$

複素数データの内積を計算するには、オペランドの 1 つを共役化する関数 ippsConj を使用する。ベクトル *pSrc1* と *pSrc2* は、同じ長さでなければならない。

例 5-27 は、関数 ippsDotProd_64f を使用して、正弦関数と余弦関数の直交性を確認する方法を示している。2 つのベクトルの内積がゼロの場合、その 2 つのベクトルは互いに直交である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDp*、*pSrc1*、または *pSrc2* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 5-27 ippsDotProd を使用して、正弦と余弦の直交性を確認する方法

```
void dotprod(void) {
        Ipp64f x[10], dp;
        int n;
        for (n = 0; n<10; ++n) x[n] = sin(IPP_2PI * n / 8);
        ippsDotProd_64f(x, x+2, 8, &dp);
        printf_64f("dp =", &dp, 1, ippStsNoErr);
}
Output:
        dp = 0.000000
Matlab* Analog:
        >> n = 0:9; x = sin(2*pi*n/8); a = x(1:8); b = x(3:10); a*b'
```

MaxEvery, MinEvery

2 つのベクトルの要素の各ペアの最大値
または最小値を計算する。

```
IppStatus ippsMaxEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int
    len);
IppStatus ippsMaxEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int
    len);
IppStatus ippsMaxEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int
    len);
IppStatus ippsMinEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int
    len);
IppStatus ippsMinEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int
    len);
IppStatus ippsMinEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int
    len);
```

引数

pSrc	最初の入力ベクトルへのポインタ。
pSrcDst	結果を格納する 2 番目の入力ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsMaxEvery は、ipps.h ファイルで宣言される。この関数は、2 つの入力ベクトルの対応する要素の各組の最大値を計算し、その結果を *pSrcDst* に格納する。

関数 ippsMinEvery は、ipps.h ファイルで宣言される。この関数は、ippsMaxEvery と同じように最小値を計算する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

サンプリング関数

この項では、信号サンプルを操作する関数について説明する。サンプリング関数は、入力信号に対するサンプリング・レートを変更し、必要な長さの信号ベクトルを得るために使用される。サンプリング関数は、次の演算を実行する。

- ゼロの値を持つサンプルを信号の隣接サンプル間に挿入する（アップ・サンプリング）。
- 信号の隣接サンプル間からサンプルを削除する（ダウン・サンプリング）。

アップ・サンプリング関数とダウン・サンプリング関数は、第 6 章で説明するフィルタリング関数の一部で使用する。

SampleUp

整数の係数でサンプリング・レートを概念的に
上げて信号をアップ・サンプリングする。

```
IppStatus ippsSampleUp_16s (const Ipp16s* pSrc, int srcLen,
   Ipp16s* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_32f (const Ipp32f* pSrc, int srcLen,
   Ipp32f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_64f (const Ipp64f* pSrc, int srcLen,
   Ipp64f* pDst, int* pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleUp_16sc (const Ipp16sc* pSrc, int srcLen,
    Ipp16sc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_32fc (const Ipp32fc* pSrc, int srcLen,
    Ipp32fc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleUp_64fc (const Ipp64fc* pSrc, int srcLen,
    Ipp64fc* pDst, int* pDstLen, int factor, int* pPhase);
```

引数

pSrc	入力配列（アップ・サンプリングする信号）へのポインタ。
srclen	入力配列 *pSrc* 内のサンプルの数。
pDst	出力配列へのポインタ。
pDstlen	出力配列 *pDst* の長さの値へのポインタ。
factor	信号をアップ・サンプリングするときの係数。すなわち、*factor*–1 個のゼロが、入力配列 *pSrc* の各サンプルの後に挿入される。
pPhase	*pSrc* からの各サンプルを、*pDst* 内の *factor* サンプルの各出力ブロック内のどこに配置するのかを決定する**入力**位相値へのポインタ。*pPhase* の値は、[0; *factor*-1] の範囲になければならない。*pPhase* の**出力**値は、同じ *factor* と次の *pSrc* を使用する次のアップ・サンプリングで使用できる。

説明

関数 ippsSampleUp は、ipps.h ファイルで宣言される。この関数は、長さ *srclen* の入力配列 *pSrc* を係数 *factor*、位相 *pPhase* でアップ・サンプリングし、その結果を配列 *pDst* に格納する。*pDstlen* のアドレスで指定する長さの値は無視される。

アップ・サンプリングは、*factor*-1 個のゼロを *pSrc* の各サンプル間に挿入する。*pPhase* 引数は、入力配列からの各サンプルを、*factor* サンプルの各出力ブロックのどこに配置するのかを決定する。*pPhase* の値は、[0; *factor*-1] の範囲になければならない。

例えば、入力の位相がゼロの場合、出力配列のすべての *factor* サンプルは対応する入力配列サンプルで開始し、他の *factor*-1 サンプルはゼロになる。出力配列の長さは、*pDstlen* のアドレスで格納される。

pPhase の値は、入力配列サンプルの位相を表す。この値は、返された出力位相でもあり、処理する次のブロックにおける最初のサンプルの入力位相として使用できる。連続した出力信号を得るには、ブロック・モード処理の *pPhase* を使用する。

ippsSampleUp の機能は、次のように記述できる。

$pDst[factor * n + phase] = pSrc[n],\ 0 \le n < srcLen$

$pDst[factor * n + m] = 0,\ 0 \le n < srcLen, 0 \le m < factor,\ m \ne phase$

$pDstLen = factor * srcLen$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst*、*pSrc*、*pDstlen* または *pPhase* が NULL。
ippStsSizeErr	エラー。*srclen* がゼロ以下。
ippStsSampleFactorErr	エラー。*factor* がゼロ以下。
ippStsSamplePhaseErr	エラー。*pPhase* が負、または *factor* 以上。

SampleDown

整数の係数でサンプリング・レートを概念的に
下げて信号をダウン・サンプリングする。

```
IppStatus ippsSampleDown_16s(const Ipp16s* pSrc, int srcLen,
    Ipp16s* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32f(const Ipp32f* pSrc, int srcLen,
    Ipp32f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_64f(const Ipp64f* pSrc, int srcLen,
    Ipp64f* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_16sc(const Ipp16sc* pSrc, int srcLen,
    Ipp16sc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_32fc(const Ipp32fc* pSrc, int srcLen,
    Ipp32fc* pDst, int* pDstLen, int factor, int* pPhase);
IppStatus ippsSampleDown_64fc(const Ipp64fc* pSrc, int srcLen,
    Ipp64fc* Dst, int* pDstLen, int factor, int* pPhase);
```

引数

pSrc	ダウン・サンプリングするサンプルを格納する入力配列へのポインタ。
srclen	入力配列 *pSrc* 内のサンプルの数。

pDst	関数 ippsSampleDown の出力を格納する配列へのポインタ。
pDstlen	出力配列 *pDst* の長さへのポインタ。
factor	信号をダウン・サンプリングするときの係数。すなわち、*factor*-1 サンプルは、*pSrc* 内の *factor* サンプルのすべてのブロックから破棄される。
pPhase	*pSrc* の *factor* サンプルの各ブロック内にあるサンプルの内、どのサンプルを破棄せず、また *pDst* にコピーしないのかを決定する**入力**位相値へのポインタ。*pPhase* の値は、[0; *factor*-1] の範囲になければならない。*pPhase* の**出力**値は、同じ *factor* と次の *pSrc* を使用する次のダウン・サンプリング（サブサンプリング）で使用できる。

説明

関数 ippsSampleDown は、ipps.h ファイルで宣言される。この関数 ippsSampleDown は、長さ *srclen* の配列 *pSrc* を係数 *factor*、位相 *pPhase* でダウン・サンプリングし、その結果を配列 *pDst* に格納する。*pDstlen* のアドレスで指定する長さの値は無視される。

ダウン・サンプリングを実行すると、*pSrc* から *factor* - 1 のサンプルを破棄し、*factor* サンプルの各ブロックからの 1 つのサンプルが *pSrc* から *pDst* にコピーされる。*pPhase* 引数は、各ブロックのどのサンプルを破棄しないかを決めるとともに、そのサンプルを *factor* サンプルの各入力ブロックのどこに配置するのかを決める。*pPhase* の値は、[0; *factor*-1] の範囲になければならない。出力配列の長さは、*pDstlen* のアドレスで格納される。

pPhase の値は、入力配列サンプルの位相を表す。この値は、返された出力位相でもあり、処理する次のブロックにおける最初のサンプルの入力位相として使用できる。連続した出力信号を得るには、ブロック・モード処理の *pPhase* を使用する。

FIR マルチレート・フィルタを使用すると、フィルタリングと再サンプリングを組み合わせられる。例えば、サブサンプリング手順の前にアンチエイリアシング・フィルタリングの実行などができる。

ippsSampleDown の機能は、次のように記述できる。

$$pDstLen = (srcLen + factor - 1 - phase)/factor$$

$$pDst[n] = pSrc[factor * n + phase],\ 0 \le n < pDstLen$$

$$phase = (factor + phase - srcLen \% factor) \% factor$$

例 5-28 は、関数 ippsSampleDown の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst*、*pSrc*、*pDstlen* または *pPhase* が NULL。
ippStsSizeErr	エラー。*srclen* がゼロ以下。
IppStsSampleFactorErr	エラー。*factor* がゼロ以下。
ippStsSamplePhaseErr	エラー。*pPhase* が負、または *factor* 以上。

例 5-28 ippsSampleDown 関数の使用例

```
void sampling( void ) {
      Ipp16s x[8] = { 1,2,3,4,5,6,7,8 };
      Ipp16s y[8] = {9,10,11,12,13,14,15,16}, z[8];
      int dstLen1, dstLen2, phase = 2;
      IppStatus st = ippsSampleDown_16s(x, 8, z, &dstLen1, 3, &phase);
      st = ippsSampleDown_16s(y, 8, z+dstLen1, &dstLen2, 3, &phase);
      printf_16s("down-sumpling =", z, dstLen1+dstLen2, st);
}
Output:
      down-sampling = 3 6 9 12 15
```

フィルタリング関数

6

本章では、たたみ込み演算、相関演算、線形フィルタリング、非線形フィルタリングを実行するインテル® IPP 関数について説明する。これらの関数は、次の項にまとめられている。

たたみ込み関数と相関関数

フィルタリング関数

各項では、最初に関数の一覧を示す。これらの関数については本項後半で説明する。

たたみ込み関数と相関関数

たたみ込みとは、任意の入力信号に対する LTI（linear time-invariant）プロセッサからの出力信号を定義する際に使う演算である。

この項の後半で説明する相関関数は、ソース・ベクトルの自己相関、または 2 つのベクトルの相互相関を推定する。

このグループのすべての関数を 表 6-1 に示す。

表 6-1　インテル ® IPP たたみ込み関数と相関関数

関数の基本名	操作
Conv	2 つのシーケンスに対して有限線形のたたみ込みを実行する。
ConvCyclic	固定サイズの 2 つのシーケンスに対して循環たたみ込みを実行する。
AutoCorr	ベクトルの自己相関をノーマル、バイアス付き、バイアスなしで推定し、その結果を 2 番目のベクトルに格納する。
CrossCorr	2 つのベクトルの相互相関を推定する。
UpdateLinear	指定された積分の重みを使用して入力ベクトルを積分する。
UpdatePower	指定された積分の重みを使用して入力ベクトルの 2 乗を積分する。

Conv

2 つのシーケンスに対して有限線形の
たたみ込みを実行する。

```
IppStatus ippsConv_32f(const Ipp32f* pSrc1, int lenSrc1, const Ipp32f*
    pSrc2, int lenSrc2, Ipp32f* pDst);
IppStatus ippsConv_64f(const Ipp64f* pSrc1, int lenSrc1, const Ipp64f*
    pSrc2, int lenSrc2, Ipp64f* pDst);
IppStatus ippsConv_16s_Sfs(const Ipp16s* pSrc1, int lenSrc1, const
    Ipp16s* pSrc2, int lenSrc2, Ipp16s* pDst, int scaleFactor);
```

引数

pSrc1, pSrc2	たたき込みの対象となる 2 つのベクトルへのポインタ。
lenSrc1	ベクトル *pSrc1* 内の要素の数。
lenSrc2	ベクトル *pSrc2* 内の要素の数。
pDst	ベクトル *pDst* へのポインタ。このベクトルは、たたみ込みの結果を格納する。
scaleFactor	第 2 章の 「整数のスケーリング」を参照。

説明

関数 ippsConv は、ipps.h ファイルで宣言される。この関数は、2 つのシーケンスに対して有限線形のたたみ込みを実行する。サイズが *lenSrc1*- のベクトル *pSrc1* とサイズが *lenSrc2*- のベクトル *pSrc2* をたたみ込むと、サイズが（*lenSrc1* + *lenSrc2* - 1）のベクトル *pDst* が生成される。たたみ込みの実行結果は、次のように定義される。

$$pDst[n] = \sum_{k=0}^{n} pSrc1[k] \cdot pSrc2[n-k], \quad 0 \le n < lenSrc1 + lenSrc2 - 1$$

$i \ge lenSrc1$ の場合は *pSrc1[i]* = 0、$j \ge lenSrc2$ の場合は *pSrc2[j]* = 0 となる。

例 6-1 は、ippsConv_16s_Sfs 関数を使用して 2 つのベクトルのたたみ込みを実行するコードを示す。

intel®

例 6-1　ippsConv 関数による 2 つのベクトルのたたみ込み

```
IppStatus convolution(void) {
      Ipp16s x[5] = {-2,0,1,-1,3}, h[2] = {0,1}, y[6];
      IppStatus st = ippsConv_16s_Sfs(x, 5, h, 2, y, 0);
      printf_16s("conv =", y, 6, st);
      return st;
}
Output:
    conv = 0 -2 0 1 -1 3
Matlab* Analog:
    >> x = [-2,0,1,-1,3]; h = [0,1]; y = conv(x,h)
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsMemAllocErr	エラー。内部バッファ用のメモリが不足している。

ConvCyclic

固定サイズの 2 つのシーケンスに対して
循環たたみ込みを実行する。

```
IppStatus ippsConvCyclic8x8_32f(const Ipp32f* x, const Ipp32f* h,
   Ipp32f* y);
IppStatus ippsConvCyclic4x4_32f32fc(const Ipp32f* x, const Ipp32fc* h,
   Ipp32fc* y);
IppStatus ippsConvCyclic8x8_16s_Sfs(const Ipp16s* x, const Ipp16s* h,
   Ipp16s* y, int scaleFactor);
```

引数

x, *h*	たたき込みの対象となるベクトルへのポインタ。

y	たたき込みの結果を格納するベクトル *y* へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsConvCyclic は、ipps.h ファイルで宣言される。この関数は、固定サイズの 2 つのシーケンスに対して循環たたみ込みを実行する。

戻り値

ippStsNoErr	エラーなし。

AutoCorr

ベクトルの自己相関をノーマル、バイアス付き、バイアスなしで推定し、その結果を 2 番目のベクトルに格納する。

```
IppStatus ippsAutoCorr_32f(const Ipp32f* pSrc, int srcLen,
    Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_32f(const Ipp32f* pSrc, int srcLen,
    Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_32f(const Ipp32f* pSrc, int srcLen,
    Ipp32f* pDst, int dstLen);
IppStatus ippsAutoCorr_64f(const Ipp64f* pSrc, int srcLen,
    Ipp64f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_64f(const Ipp64f* pSrc, int srcLen,
    Ipp64f* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_64f(const Ipp64f* pSrc, int srcLen,
    Ipp64f* pDst, int dstLen );
IppStatus ippsAutoCorr_32fc(const Ipp32fc* pSrc, int srcLen,
    Ipp32fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_32fc(const Ipp32fc* pSrc, int srcLen,
    Ipp32fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormB_32fc(const Ipp32fc* pSrc, int srcLen,
    Ipp32fc* pDst, int dstLen);
IppStatus ippsAutoCorr_64fc(const Ipp64fc* pSrc, int srcLen,
    Ipp64fc* pDst, int dstLen);
IppStatus ippsAutoCorr_NormA_64fc(const Ipp64fc* pSrc, int srcLen,
    Ipp64fc* pDst, int dstLen);
```

```
IppStatus ippsAutoCorr_NormB_64fc(const Ipp64fc* pSrc, int srcLen,
    Ipp64fc* pDst, int dstLen);
IppStatus ippsAutoCorr_16s_Sfs(const Ipp16s* pSrc, int srcLen,
    Ipp16s* pDst, int dstLen, int scaleFactor );
IppStatus ippsAutoCorr_NormA_16s_Sfs( const Ipp16s* pSrc, int srcLen,
    Ipp16s* pDst, int dstLen, int scaleFactor );
IppStatus ippsAutoCorr_NormB_16s_Sfs(const Ipp16s* pSrc, int srcLen,
    Ipp16s* pDst, int dstLen, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
srcLen	ソース・ベクトル内の要素の数。
pDst	ソース・ベクトルの推定された自己相関の結果を格納する、デスティネーション・ベクトルへのポインタ。
dstLen	デスティネーション・ベクトル内の要素の数（自己相関の長さ）。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsAutoCorr は、ipps.h ファイルで宣言される。この関数は、長さ *srcLen* のソース・ベクトル *pSrc* の自己相関をノーマルで推定し、その結果を長さ *dstLen* のベクトル *pDst* に格納する。関数型 ippsAutoCorr_NormA と ippsAutoCorr_NormB は、ソース・ベクトルの自己相関をそれぞれバイアス付きとバイアスなしで計算する。結果のベクトル *pDst* は、次の式で定義される。

$$pDst[n] = \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \text{（ノーマル）}$$

$$pDst[n] = \frac{1}{srcLen} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen \text{（バイアス付き）}$$

$$pDst[n] = \frac{1}{srcLen - n} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \le n < dstLen$$

（バイアスなし）

ここで、

$$pSrc[i] = \begin{cases} pSrc[i], & 0 \le i < srcLen \\ 0, & otherwise \end{cases}$$

アプリケーション・ノート

自己相関の測定値は、正のラグに対してのみ計算する。これは、負のラグ値に対する自己相関が、等価の正のラグに対する自己相関の共役複素数のためである。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*srcLen* または *dstLen* がゼロ以下。

関連項目

ippsCrossCorr	2 つのベクトルの相互相関を推定する。

CrossCorr

2 つのベクトルの相互相関を推定する。

```
IppStatus ippsCrossCorr_32f(const Ipp32f* pSrc1, int len1,
    const Ipp32f* pSrc2, int len2, Ipp32f* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64f(const Ipp64f* pSrc1, int len1,
    const Ipp64f* pSrc2, int len2, Ipp64f* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_32fc(const Ipp32fc* pSrc1, int len1,
    const Ipp32fc* pSrc2, int len2, Ipp32fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_64fc(const Ipp64fc* pSrc1, int len1,
    const Ipp64fc* pSrc2, int len2, Ipp64fc* pDst, int dstLen, int lowLag);
IppStatus ippsCrossCorr_16s_Sfs(const Ipp16s* pSrc1, int len1,
    const Ipp16s* pSrc2, int len2, Ipp16s* pDst, int dstLen, int lowLag,
    int scaleFactor);
```

引数

pSrc1	先頭のソース・ベクトルへのポインタ。
len1	ベクトル *pSrc1* 内の要素の数。
pSrc2	2 番目のソース・ベクトルへのポインタ。

len2	ベクトル *pSrc2* 内の要素の数。
pDst	ベクトル *pSrc1* と *pSrc2* の相互相関の推定結果を格納するベクトルへのポインタ。
dstLen	ベクトル *pDst* 内の要素の数。これにより、相関の推定値を計算する際のラグの範囲が決定される。
lowLag	相関の推定値を計算する際のラグ範囲の下限。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsCrossCorr は、ipps.h ファイルで宣言される。この関数は、長さ *len1* のベクトル *pSrc1* と長さ *len2* のベクトル *pSrc2* の相互相関を推定し、その結果をベクトル *pDst* に格納する。

結果のベクトル *pDst* は、次の式で定義される。

$$pDst[n] = \sum_{i=0}^{len1-1} conj(pSrc1[i]) \cdot pSrc2[n+i+lowLag] \ ,$$

$0 \le n < dstLen$ および

$$pSrc2[j] = \begin{cases} pSrc2[j], & 0 \le j < len2 \\ 0, & otherwise \end{cases}$$

例 6-2 は、関数 ippsCrossCorr の使用例を示す。

例 6-2 ippsCrossCorr 関数の使用例

```
void crossCorr(void) {
      #undef LEN
      #define LEN 11
      Ipp32f win[LEN], y[LEN];
      IppStatus st;
      ippsSet_32f (1, win, LEN);
      ippsWinHamming_32f_I (win, LEN);
      st = ippsCrossCorr_32f (win, LEN, win, LEN, y, LEN, -(LEN-1));
      printf_32f("cross corr =", y,7,st);
}
Output:
         cross corr = 0.006400 0.026856 0.091831 0.242704 0.533230
         1.009000 1.672774
Matlab* analog:
         >> x = hamming(11)'; y = xcorr(x,x); y(1:7)
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1* または *pSrc2* が NULL。
ippStsSizeErr	エラー。*len1* または *len2* がゼロ以下。

UpdateLinear

指定された積分の重みを使用して入力ベクトルを積分する。

```
IppStatus ippsUpdateLinear_16s32s_I(const Ipp16s* pSrc, int len,
   Ipp32s* pSrcDst, int srcShiftRight, Ipp16s alpha, IppHintAlgorithm
   hint);
```

引数

pSrc	ソース・ベクトルへのポインタ。
len	ベクトルの要素の数。
pSrcDst	入力値と出力結果へのポインタ。
srcShiftRight	シフト値（負でない値)。

intel®

alpha	積分の重み。
hint	特別のコードの使用を指定する。*hint* 引数の値については、「flag 引数と hint 引数」を参照のこと。

説明

関数 ippsUpdateLinear は、ipps.h ファイルで宣言される。この回数は *len* 回反復され、合計

*alpha**pSrcDst + (1-*alpha*)*$pSrc[i]_{shift}$

を計算して *pSrcDst* に格納する。

ここで、*i* はそれまでの反復回数、*pSrcDst* は前回の反復の結果、$pSrc[i]_{shift}$ は、負でない値 *srcShiftRight* によって右にシフトされたソース・ベクトル要素である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

UpdatePower

指定された積分の重みを使用して入力ベクトルの 2 乗を積分する。

```
IppStatus ippsUpdatePower_16s32s_I(const Ipp16s* pSrc, int len,
    Ipp32s* pSrcDst, int srcShiftRight,Ipp16s alpha, IppHintAlgorithm
    hint);
```

引数

pSrc	ソース・ベクトルへのポインタ。
len	ベクトルの要素の数。
pSrcDst	入力と出力へのポインタ。
srcShiftRight	シフト値
alpha	積分の重み。
hint	特別のコードの使用を指定する。*hint* 引数の値については、「flag 引数と hint 引数」を参照のこと。

説明

関数 ippsUpdatePower は、ipps.h ファイルで宣言される。この回数は *len* 回反復され、合計

$alpha*pSrcDst + (1-alpha)*pSrc[i]_{shift}*pSrc[i]_{shift}$

を計算して *pSrcDst* に格納する。

ここで、*i* はそれまでの反復回数、*pSrcDst* は 前回の反復の結果、$pSrc[i]_{shift}$ は、負でない値 *srcShiftRight* によって右にシフトされたソース・ベクトル要素である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

フィルタリング関数

この項では､次のタイプのフィルタを実装するインテル® IPP 関数について説明する。

- 有限インパルス応答（FIR）フィルタ
- 最小平均 2 乗（LMS）フィルタを使用する適応 FIR フィルタ
- 無限インパルス応答（IIR）フィルタ
- メディアン・フィルタ

この関数のセットは、異なるタイプの FIR フィルタのフィルタ係数を生成する。

すべてのオーディオ・コーディング関数を 表 6-2 に示す。

表 6-2　インテル® IPP フィルタリング関数

関数の基本名	操作
FIR フィルタ関数	
FIRInitAlloc, FIRMRInitAlloc	メモリを割り当て、シングルレートまたはマルチレートの FIR フィルタ・ステート構造体を初期化する。
FIRFree	ippsFIRInitAlloc または ippsFIRMRInitAlloc 関数によって割り当てられたメモリを解放する。
ippsFIRInit, FIRMRInit	シングルレートまたはマルチレートの FIR フィルタ・ステート構造体を初期化する。

表 6-2　インテル® IPP フィルタリング関数 （続き）

関数の基本名	操作
ippsFIRGetStateSize, FIRMRGetStateSize	FIR フィルタ構造体の外部バッファのサイズを計算する。
ippsFIRGetTaps, FIRSetTaps	FIR フィルタ・ステートのタップ値を取得し設定する。
FIRGetDlyLine, FIRSetDlyLine	FIR フィルタ・ステートの遅延線の内容を取得し設定する。
FIROne	FIR フィルタを使用して単一のサンプルをフィルタリングする。
FIR	シングルレートまたはマルチレートの FIR フィルタを使用して、サンプルのブロックをフィルタリングする。
直接フィルタリング	
FIROne_Direct	FIR フィルタを使用して単一のサンプルを直接フィルタリングする。
FIR_Direct	シングルレートの FIR フィルタを使用してサンプルのブロックを直接フィルタリングする。
FIRMR_Direct	マルチレートの FIR フィルタを使用してサンプルのブロックを直接フィルタリングする。
FIR フィルタ係数生成関数	
ippsFIRGenLowpass	ローパスの FIR フィルタ係数を計算する。
ippsFIRGenHighpass	ハイパスの FIR フィルタ係数を計算する。
ippsFIRGenBandpass	バンドパスの FIR フィルタ係数を計算する。
ippsFIRGenBandstop	バンドストップ FIR フィルタ係数を計算する。
シングルレート FIR LMS フィルタ関数	
FIRLMSInitAlloc	メモリを割り当て、最小平均 2 乗（LMS）アルゴリズムを使用する適応 FIR フィルタを初期化する。
FIRLMSFree	最小平均 2 乗 (LMS) アルゴリズムを使用する適応 FIR フィルタをクローズする。
FIRLMSGetTaps	FIR LMS フィルタのタップを取得する。
FIRLMSGetDlyLine, FIRLMSSetDlyLine	FIR LMS フィルタの遅延線の内容を取得し設定する。
FIRLMS	FIR LMS フィルタを使用して配列をフィルタリングする。
直接フィルタリング	
FIRLMSOne_Direct	FIR LMS フィルタを使用して単一のサンプルをフィルタリングする。
マルチレート FIR LMS フィルタ関数	
FIRLMSMRInitAlloc	メモリを割り当て、最小平均 2 乗（LMS）アルゴリズムを使用する適応マルチレート FIR フィルタを初期化する。
FIRLMSMRFree	最小平均 2 乗アルゴリズムを使用する適応マルチレート FIR フィルタをクローズする。
FIRLMSMRSetMu	適応ステップを設定する。
FIRLMSMRUpdateTaps	適応エラーの値を使用してフィルタ係数を更新する。

表 6-2 インテル® IPP フィルタリング関数 （続き）

関数の基本名	操作
FIRLMSMRGetTaps, FIRLMSMRSetTaps	マルチレート FIR LMS フィルタのタップを取得し設定する。
FIRLMSMRGetTapsPointer	フィルタ係数へのポインタを返す。
FIRLMSMRGetDlyLine, FIRLMSMRSetDlyLine	マルチレート FIR LMS フィルタ・ステートの遅延線の内容を取得し設定する。
FIRLMSMRGetDlyVal	指定された位置から 1 つの遅延線の値を取得する。
FIRLMSMRPutVal	遅延線内に入力値を置く。
FIRLMSMROne	遅延線内に置かれたデータをフィルタリングする。
FIRLMSMROneVal	1 つの入力値をフィルタリングする。
IIR フィルタ関数	
IIRInitAlloc, IIRInitAlloc_BiQuad	メモリを割り当て、IIR フィルタを初期化する。
IIRFree	IIR フィルタ・ステートをクローズする。
ippsIIRInit, IIRInit_BiQuad	IIR フィルタ・ステートを初期化する。
ippsIIRGetStateSize, IIRGetStateSize_BiQuad	IIR フィルタ・ステート構造体の長さを返す。
ippsIIRSetTaps	IIR フィルタ・ステートのタップを設定する。
IIRGetDlyLine, IIRSetDlyLine	IIR フィルタ・ステートの遅延線の値を取得し設定する。
IIROne	IIR フィルタを使用して単一のサンプルをフィルタリングする。
説明	IIR フィルタを使用してサンプルのブロックをフィルタリングする。
直接フィルタリング	
ippsIIROne_Direct, IIROne_BiQuadDirect	IIR フィルタを使用して単一のサンプルを直接フィルタリングする。
ippsIIR_Direct, IIR_BiQuadDirect	IIR フィルタを使用してサンプルのブロックを直接フィルタリングする。
メディアン・フィルタ関数	
FilterMedian	入力配列の各要素のメディアン値を計算する。

FIR フィルタ関数

この項で説明する関数は、入力データの有限インパルス応答（FIR）フィルタリングを実行する。この項では、FIR フィルタの初期化、遅延線とフィルタ係数（タップ）の取得と設定、フィルタリングを実行する関数を説明する。FIR フィルタ関数を使用するには、一般に次の手順を実行する。

1. ippsFIRInitAlloc を呼び出してメモリを割り当て、シングルレート・フィルタのフィルタ・ステート構造体のタップと遅延線を初期化するか、ippsFIRMRInitAlloc を呼び出してメモリを割り当て、マルチレート・フィルタのフィルタ・ステート構造体のタップと遅延線を初期化する。あるいは、ippsFIRInit か ippsFIRMRInit を呼び出して、以前に作成した外部バッファに格納された対応するフィルタ・ステート構造体のタップと遅延線を初期化する。このバッファのサイズは、関数 ippsFIRGetStateSize または ippsFIRMRGetStateSize を呼び出して計算できる。

2. ippsFIROne を呼び出し、シングルレート・フィルタを使用して単一のサンプルをフィルタリングするか、ippsFIR を呼び出し、シングルレート・フィルタまたはマルチレート・フィルタを使用して、連続するサンプルのブロックをフィルタリングする。

3. 以前に初期化したフィルタ・ステートに新しいタップと遅延線の値を設定するには、関数 ippsFIRSetTaps および ippsFIRSetDlyLine を呼び出す。初期化したフィルタ・ステートのタップと遅延線の値を取得するには、関数 ippsFIRGetTaps および ippsFIRGetDlyLine を呼び出す。

4. ippsFIRFree を呼び出し、ippsFIRInitAlloc または ippsFIRMRInitAlloc で作成した FIR フィルタ・ステート構造体に関連付けられた動的メモリを解放する。

あるいは、フィルタ関数の直接バージョンも使用できる。直接バージョンのフィルタ関数は、フィルタのステート構造体を初期化せずにフィルタリングを実行する。すべての必要なパラメータは、関数内で直接設定される。

この関数のセットは、異なるフィルタのフィルタ係数を計算する。

FIRInitAlloc, FIRMRInitAlloc

メモリを割り当て、シングルレートまたはマルチレートの FIR フィルタ・ステートを初期化する。

```
IppStatus ippsFIRInitAlloc_32f(IppsFIRState_32f** pState,
   const Ipp32f* pTaps, int tapsLen, const Ipp32f* pDlyLine);
IppStatus ippsFIRMRInitAlloc_32f(IppsFIRState_32f** pState,
   const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRInitAlloc_64f(IppsFIRState_64f** pState,
   const Ipp64f* pTaps, int tapsLen, const Ipp64f* pDlyLine);
IppStatus ippsFIRMRInitAlloc_64f(IppsFIRState_64f** pState,
   const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp64f* pDlyLine);
IppStatus ippsFIRInitAlloc_32fc(IppsFIRState_32fc** pState,
   const Ipp32fc* pTaps, int tapsLen, const Ipp32fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc_32fc(IppsFIRState_32fc** pState,
   const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp32fc* pDlyLine);
IppStatus ippsFIRInitAlloc_64fc(IppsFIRState_64fc** pState,
   const Ipp64fc* pTaps, int tapsLen, const Ipp64fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc_64fc(IppsFIRState_64fc** pState,
   const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp64fc* pDlyLine);


IppStatus ippsFIRInitAlloc32s_16s(IppsFIRState32s_16s** pState,
   const Ipp32s* pTaps, int tapsLen, int tapsFactor,
   const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc32s_16s(IppsFIRState32s_16s** pState,
   const Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor,
   int upPhase, int downFactor, int downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32s_16s32f(IppsFIRState32s_16s** pState,
   const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc32s_16s32f(IppsFIRState32s_16s** pState,
   const Ipp32f *pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** pState,
   const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
   const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** pState,
   const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor,
   int upPhase, int downFactor, int downPhase, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc** pState,
   const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc**
   pState, const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp16sc* pDlyLine);


IppStatus ippsFIRInitAlloc32f_16s(IppsFIRState32f_16s** pState,
   const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc32f_16s(IppsFIRState32f_16s** pState,
   const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp16s* pDlyLine);
```

```
IppStatus ippsFIRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** pState,
    const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** pState,
    const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp16sc* pDlyLine);

IppStatus ippsFIRInitAlloc64f_16s(IppsFIRState64f_16s** pState,
    const Ipp64f* pTaps, int tapsLen, const Ipp16s* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_16s(IppsFIRState64f_16s** pState,
    const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp16s* pDlyLine);
IppStatus ippsFIRInitAlloc64f_32s(IppsFIRState64f_32s** pState,
    const Ipp64f* pTaps, int tapsLen, const Ipp32s* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_32s(IppsFIRState64f_32s** pState,
    const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp32s* pDlyLine);
IppStatus ippsFIRInitAlloc64f_32f(IppsFIRState64f_32f** pState,
    const Ipp64f* pTaps, int tapsLen, const Ipp32f* pDlyLine);
IppStatus ippsFIRMRInitAlloc64f_32f(IppsFIRState64f_32f** pState,
    const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp32f* pDlyLine);

IppStatus ippsFIRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** pState,
    const Ipp64fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** pState,
    const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp16sc* pDlyLine);
IppStatus ippsFIRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** pState,
    const Ipp64fc* pTaps, int tapsLen, const Ipp32sc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** pState,
    const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp32sc* pDlyLine);
IppStatus ippsFIRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** pState,
    const Ipp64fc* pTaps, int tapsLen, const Ipp32fc* pDlyLine);
IppStatus ippsFIRMRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** pState,
    const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp32fc* pDlyLine);
```

引数

pTaps	タップの値を格納した配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップの値を格納した配列内の要素の数。

tapsFactor	データ型が Ipp32s のタップに対するスケール係数 (整数バージョンのみ)。
downFactor	マルチレート信号をダウンサンプリングする際に関数 ippsFIRMRInit が使用する係数。
downPhase	マルチレート信号をダウンサンプリングする際に関数 ippsFIRMRInit が使用する位相。
upFactor	マルチレート信号をアップサンプリングする際に関数 ippsFIRMRInit が使用する係数。
upPhas	マルチレート信号をアップサンプリングする際に関数 ippsFIRMRInit が使用する位相。
pDlyLine	遅延線の値が格納された配列へのポインタ。配列内の要素の数は、シングルレート・フィルタの場合は *tapsLen*、マルチレート・フィルタの場合は (*tapsLen* + *upFactor* – 1) / *upFactor* として定義される。
pState	生成する FIR ステート構造体へのポインタ。

説明

関数 ippsFIRInitAlloc と ippsFIRMRInitAlloc は ipps.h ファイルで宣言される。これらの関数は、それぞれシングルレートまたはマルチレートの FIR フィルタ・ステートのメモリを割り当て、初期化する。初期化関数は、長さ *tapsLen* の配列 *pTaps* からステート構造体 *pState* にタップをコピーする。整数タップをスケーリングするには、*tapsFactor* 値を使用する。配列 *pDlyLine* は、遅延線の値を指定する。配列 *pDlyLine* へのポインタが NULL でない場合、配列の内容はステート構造体 *pState* にコピーされる。それ以外の場合、ステート構造体の遅延線の値はゼロに初期化される。

ステートが生成されない場合、初期化関数はエラー・ステータスを返す。

ippsFIRInitAlloc。関数 ippsFIRInitAlloc は、シングルレート・フィルタのステート構造体 *pState* のタップと遅延線を初期化する。長さ *tapsLen* の配列 *pTaps* は、タップを指定する。遅延線配列 *pDlyLine* が NULL でない場合、その長さは *tapsLen* になる。ただし、この遅延線の長さは、直接 FIR フィルタの場合における遅延線の長さとは異なる（直接フィルタでは、遅延線の長さは 2 倍になる)。

ippsFIRMRInitAlloc。関数 ippsFIRMRInitAlloc は、マルチレート・フィルタ (すなわち、ポリフェーズ・フィルタ構造体を使用してアップサンプリングやダウンサンプリングを内部的に実行するフィルタ）のステート構造体 *pState* のタップと遅延線を初期化する。この関数がステート構造体を初期化する方法は、シングルレート・フィルタの場合と同じである。ただしこの関数には、アップサンプリングやダウンサンプリングで必要なパラメータの情報も含まれる。

引数 *upFactor* は、フィルタリング対象の信号を内部的にアップサンプリングする際の係数である（5-118 ページの「ippsSampleUp」を参照)。すなわち、入力信号の各サンプル間に *upFactor*-1 個のゼロを挿入する。

引数 *upPhase* は、アップサンプリング後の入力信号のサイズ *upFactor* のブロック内にあるゼロ以外のサンプルの位置を決定するためのパラメータである。

引数 *downFactor* は、アップサンプリングした入力信号をフィルタリングして得た FIR 応答を内部でダウンサンプリングする際の係数である（5-120 ページの「ippsSampleDown」を参照）。つまり、アップサンプリング後のフィルタ応答のサイズ *downFactor* の各出力ブロックから、*downFactor*-1 個の出力サンプルを破棄する。

引数 *downPhase* は、アップサンプリング後のフィルタ応答のブロック内で、破棄されていないサンプルの位置を決定するためのパラメータである。

遅延線配列 *pDelay* が NULL ではない場合、そのサイズは次のように定義される。
(*tapsLen* + *upFactor* - 1) / *upFactor*

戻り値

ippStsNoErr	エラーなし。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsFIRLenErr	エラー。*tapsLen* がゼロ以下。
ippStsFIRMRFactorErr	エラー。*upFactor*（*downFactor*）がゼロ以下。
ippStsFIRMRPhaseErr	エラー。*upPhase*（*downPhase*）が負、または *upFactor*（*downFactor*）以上。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRFree

FIR フィルタ・ステートをクローズする。

```
IppStatus ippsFIRFree_32f(IppsFIRState_32f* pState);
IppStatus ippsFIRFree_64f(IppsFIRState_64f* pState);
IppStatus ippsFIRFree_32fc(IppsFIRState_32fc* pState);
IppStatus ippsFIRFree_64fc(IppsFIRState_64fc* pState);
IppStatus ippsFIRFree32s_16s(IppsFIRState32s_16s* pState);
```

```
IppStatus ippsFIRFree32f_16s(IppsFIRState32f_16s* pState);
IppStatus ippsFIRFree32sc_16sc(IppsFIRState32sc_16sc* pState);
IppStatus ippsFIRFree32fc_16sc(IppsFIRState32fc_16sc* pState);
IppStatus ippsFIRFree64f_16s(IppsFIRState64f_16s* pState);
IppStatus ippsFIRFree64f_32s(IppsFIRState64f_32s* pState);
IppStatus ippsFIRFree64f_32f(IppsFIRState64f_32f* pState);
IppStatus ippsFIRFree64fc_16sc(IppsFIRState64fc_16sc* pState);
IppStatus ippsFIRFree64fc_32sc(IppsFIRState64fc_32sc* pState);
IppStatus ippsFIRFree64fc_32fc(IppsFIRState64fc_32fc* pState);
```

引数

pState	クローズする FIR フィルタ・ステート構造体へのポインタ。

説明

関数 ippsFIRFree は、ipps.h ファイルで宣言される。この関数は、ippsFIRInitAlloc または ippsFIRMRInitAlloc により生成されたフィルタ・ステートに関連付けられているメモリをすべて解放して、FIR フィルタ・ステート構造体をクローズする。フィルタリングが完了したら、ippsFIRFree を呼び出す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRInit, FIRMRInit

シングルレートまたはマルチレートの
FIR フィルタ・ステートを初期化する。

```
IppStatus ippsFIRInit_32f(IppsFIRState_32f** pState, const Ipp32f*
    pTaps, int tapsLen, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit_32f(IppsFIRState_32f** pState, const Ipp32f* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
    const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit_64f(IppsFIRState_64f** pState, const Ipp64f* pTaps,
    int tapsLen, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit_64f(IppsFIRState_64f** pState, const Ipp64f* pTaps,
    int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
    const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit_32fc(IppsFIRState_32fc** pState, const Ipp32fc*
    pTaps, int tapsLen, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit_32fc(IppsFIRState_32fc** pState,
    const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRInit_64fc(IppsFIRState_64fc** pState, const Ipp64fc*
    pTaps, int tapsLen, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit_64fc(IppsFIRState_64fc** pState, const Ipp64fc*
    pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
    const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit32s_16s(IppsFIRState32s_16s** pState,
    const Ipp32s* pTaps, int tapsLen, int tapsFactor, const Ipp16s*
    pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32s_16s(IppsFIRState32s_16s** pState,
    const Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor,
    int upPhase, int downFactor, int downPhase, const Ipp16s* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRInit32s_16s32f(IppsFIRState32s_16s** pState,
    const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32s_16s32f(IppsFIRState32s_16s** pState,
    const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp16s* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRInit32sc_16sc(IppsFIRState32sc_16sc** pState,
    const Ipp32sc* pTaps, int tapsLen, int tapsFactor, const Ipp16sc*
    pDlyLine, Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32sc_16sc(IppsFIRState32sc_16sc** pState,
    const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int
    upPhase, int downFactor, int downPhase, const Ipp16sc* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRInit32sc_16sc32fc(IppsFIRState32sc_16sc** pState,
    const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32sc_16sc32fc(IppsFIRState32sc_16sc** pState,
    const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp16sc* pDlyLine,
    Ipp8u* pBuffer);
```

```
IppStatus ippsFIRInit32f_16s(IppsFIRState32f_16s** pState,
   const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine,
   Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32f_16s(IppsFIRState32f_16s** pState,
   const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp16s* pDlyLine,
   Ipp8u* pBuffer);
IppStatus ippsFIRInit32fc_16sc(IppsFIRState32fc_16sc** pState,
   const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine,
   Ipp8u* pBuffer);
IppStatus ippsFIRMRInit32fc_16sc(IppsFIRState32fc_16sc** pState,
   const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp16sc* pDlyLine,
   Ipp8u* pBuffer);

IppStatus ippsFIRInit64f_16s(IppsFIRState64f_16s** pState,
   const Ipp64f* pTaps, int tapsLen, const Ipp16s* pDlyLine,
   Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64f_16s(IppsFIRState64f_16s** pState,
   const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp16s* pDlyLine,
   Ipp8u* pBuffer);
IppStatus ippsFIRInit64f_32s(IppsFIRState64f_32s** pState,
   const Ipp64f* pTaps, int tapsLen, const Ipp32s* pDlyLine,
   Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64f_32s(IppsFIRState64f_32s** pState,
   const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp32s* pDlyLine,
   Ipp8u* pBuffer);
IppStatus ippsFIRInit64f_32f(IppsFIRState64f_32f** pState,
   const Ipp64f* pTaps, int tapsLen, const Ipp32f* pDlyLine,
   Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64f_32f(IppsFIRState64f_32f** pState,
   const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp32f* pDlyLine,
   Ipp8u* pBuffer);

IppStatus ippsFIRInit64fc_16sc(IppsFIRState64fc_16sc** pState,
   const Ipp64fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine,
   Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64fc_16sc(IppsFIRState64fc_16sc** pState,
   const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, const Ipp16sc* pDlyLine,
   Ipp8u* pBuffer);
```

```
IppStatus ippsFIRInit64fc_32sc(IppsFIRState64fc_32sc** pState,
    const Ipp64fc* pTaps, int tapsLen, const Ipp32sc* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64fc_32sc(IppsFIRState64fc_32sc** pState,
    const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp32sc* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRInit64fc_32fc(IppsFIRState64fc_32fc** pState,
    const Ipp64fc* pTaps, int tapsLen, const Ipp32fc* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsFIRMRInit64fc_32fc(IppsFIRState64fc_32fc** pState,
    const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
    int downFactor, int downPhase, const Ipp32fc* pDlyLine,
    Ipp8u* pBuffer);
```

引数

pTaps	タップの値を格納した配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップの値を格納した配列内の要素の数。
tapsFactor	データ型が Ipp32s のタップに対するスケール係数(整数バージョンのみ)。
downFactor	マルチレート信号をダウンサンプリングする際に関数 ippsFIRMRInit が使用する係数。
downPhase	マルチレート信号をダウンサンプリングする際に関数 ippsFIRMRInit が使用する位相。
upFactor	マルチレート信号をアップサンプリングする際に関数 ippsFIRMRInit が使用する係数。
upPhas	マルチレート信号をアップサンプリングする際に関数 ippsFIRMRInit が使用する位相。
pDlyLine	遅延線の値が格納された配列へのポインタ。配列内の要素の数は、シングルレート・フィルタの場合は *tapsLen*、マルチレート・フィルタの場合は(*tapsLen* + *upFactor* - 1) / *upFactor* として定義される。
pState	生成する FIR ステート構造体へのポインタ。
pBuffer	FIR ステート構造体の外部バッファへのポインタ。

説明

関数 `ippsFIRInit` と `ippsFIRMRInit` は、`ipps.h` ファイルで宣言される。これらの関数は、外部バッファにあるシングルレートまたはマルチレートの FIR フィルタ・ステート構造体を初期化する。バッファのサイズは、関数 `ippsFIRGetStateSize, FIRMRGetStateSize` を呼び出して、あらかじめ計算する必要がある。初期化関数は、長さ *tapsLen* の配列 *pTaps* からステート構造体 *pState* にタップをコピーする。整数タップをスケーリングするには、*tapsFactor* 値を使用する。配列 *pDlyLine* は、遅延線の値を指定する。配列 *pDlyLine* へのポインタが NULL ではない場合、配列の内容はステート構造体 *pState* にコピーされる。それ以外の場合、ステート構造体の遅延線の値はゼロに初期化される。

ippsFIRInit。関数 `ippsFIRInit` は、シングルレート・フィルタのステート構造体 *pState* のタップと遅延線を初期化する。長さ *tapsLen* の配列 *pTaps* は、タップを指定する。遅延線配列 *pDlyLine* が NULL ではない場合、その長さは *tapsLen* になる。ただし、この遅延線の長さは、直接 FIR フィルタの場合における遅延線の長さとは異なる（直接フィルタでは、遅延線の長さは 2 倍になる）。

ippsFIRMRInit。関数 `ippsFIRMRInit` は、マルチレート・フィルタ（すなわち、ポリフェーズ・フィルタ構造体を使用してアップサンプリングやダウンサンプリング信号を内部的に実行するフィルタ）のステート構造体 *pState* のタップと遅延線を初期化する。この関数がステート構造体を初期化する方法は、シングルレート・フィルタの場合と同じである。ただし、この関数にはアップサンプリングやダウンサンプリングで必要なパラメータの情報も含まれる。

引数 *upFactor* は、フィルタリング対象の信号を内部的にアップサンプリングする際の係数である（5-118 ページの「`ippsSampleUp`」を参照）。すなわち、入力信号の各サンプル間に *upFactor*-1 個のゼロを挿入する。

引数 *upPhase* は、アップサンプリング後の入力信号のサイズ *upFactor* のブロック内にあるゼロ以外のサンプルの位置を決定するためのパラメータである。

引数 *downFactor* は、アップサンプリングした入力信号をフィルタリングして得た FIR 応答を内部でダウンサンプリングする際の係数である（5-120 ページの「`ippsSampleDown`」を参照）。つまり、アップサンプリング後のフィルタ応答のサイズ *downFactor* の各出力ブロックから、*downFactor*-1 個の出力サンプルを破棄する。

引数 *downPhase* は、アップサンプリング後のフィル ^ 応答のブロック内で、破棄されていないサンプルの位置を決定するためのパラメータである。

遅延線配列 *pDlyLine* が NULL ではない場合、そのサイズは次のように定義される。
(*tapsLen* + *upFactor* – 1) / *upFactor*

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsFIRLenErr	エラー。*tapsLen* がゼロ以下。
ippStsFIRMRFactorErr	エラー。*upFactor*（*downFactor*）がゼロ以下。
ippStsFIRMRPhaseErr	エラー。*upPhase*(*downPhase*)が負、または *upFactor*（*downFactor*）以上。

FIRGetStateSize, FIRMRGetStateSize

FIR フィルタ・ステート構造体の長さを返す。

```
IppStatus ippsFIRGetStateSize_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_32f(int tapsLen, int upFactor,
   int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize_64f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_64f(int tapsLen, int upFactor,
   int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize_32fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_32fc(int tapsLen, int upFactor,
   int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize_64fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize_64fc(int tapsLen, int upFactor,
   int downFactor, int* pBufferSize);

IppStatus ippsFIRGetStateSize32s_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32s_16s(int tapsLen, int upFactor,
   int downFactor, int* pBufferSize));
IppStatus ippsFIRGetStateSize32s_16s32f(int tapsLen, int*
   pBufferSize));
IppStatus ippsFIRMRGetStateSize32s_16s32f(int tapsLen, int upFactor,
   int downFactor, int* pBufferSize));
IppStatus ippsFIRGetStateSize32sc_16sc(int tapsLen, int* pBufferSize));
IppStatus ippsFIRMRGetStateSize32sc_16sc(int tapsLen, int upFactor,
   int downFactor, int* pBufferSize));
IppStatus ippsFIRGetStateSize32sc_16sc32fc(int tapsLen, int*
   pBufferSize));
```

```
IppStatus ippsFIRMRGetStateSize32sc_16sc32fc(int tapsLen, int upFactor,
    int downFactor, int* pBufferSize));

IppStatus ippsFIRGetStateSize32f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32f_16s(int tapsLen, int upFactor,
    int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize32fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize32fc_16sc(int tapsLen, int upFactor,
    int downFactor, int* pBufferSize);

IppStatus ippsFIRGetStateSize64f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_16s(int tapsLen, int upFactor,
    int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_32s(int tapsLen, int upFactor,
    int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64f_32f(int tapsLen, int upFactor,
    int downFactor, int* pBufferSize);

IppStatus ippsFIRGetStateSize64fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_16sc(int tapsLen, int upFactor,
    int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_32sc(int tapsLen, int upFactor,
    int downFactor, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRMRGetStateSize64fc_32fc(int tapsLen, int upFactor,
    int downFactor, int* pBufferSize);
```

引数

tapsLen	タップの値を格納した配列内の要素の数。
downFactor	マルチレート信号をダウンサンプリングする際に関数 ippsFIRMRInit が使用する係数。
upFactor	マルチレート信号をアップサンプリングする際に関数 ippsFIRMRInit が使用する係数。
pBufferSize	計算されたバッファ・サイズ値へのポインタ。

説明

関数 ippsFIRGetStateSize と ippsFIRMRGetStateSize は、ipps.h ファイルで宣言される。これらの関数は、シングルレートまたはマルチレート FIR フィルタ・ステートの外部バッファ・サイズを計算し、その結果を *pBufferSize* に格納する。

シングルレート FIR フィルタ・ステートのバッファ・サイズを計算する場合、タップの数 *tapsLen* のみを指定する必要がある。マルチレート FIR フィルタ・ステートのバッファ・サイズを計算する場合、タップの数 *tapsLen* とアップサンプリング / ダウンサンプリング・パラメータ *upFactor* と *downFactor* を指定する必要がある。引数 *upFactor* は、フィルタリング対象の信号を内部的にアップサンプリングする際の係数である（5-118 ページの「ippsSampleUp」を参照）。すなわち、入力信号の各サンプル間に *upFactor*-1 個のゼロを挿入する。

引数 *downFactor* は、アップサンプリングした入力信号をフィルタリングして得た FIR 応答を内部でダウンサンプリングする際の係数である（5-120 ページの「ippsSampleDown」を参照）。つまり、アップサンプリング後のフィルタ応答のサイズ *downFactor* の各出力ブロックから、*downFactor*-1 個の出力サンプルを破棄する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pBufferSize* が NULL。
ippStsFIRLenErr	エラー。*tapsLen* がゼロ以下。
ippStsFIRMRFactorErr	エラー。*upFactor* (*downFactor*) がゼロ以下。

FIRGetTaps, FIRSetTaps

FIR フィルタ・ステートのタップを取得し設定する。

```
IppStatus ippsFIRGetTaps_32f(const IppsFIRState_32f* pState,
    Ipp32f* pTaps);
IppStatus ippsFIRGetTaps_64f(const IppsFIRState_64f* pState,
    Ipp64f* pTaps);
IppStatus ippsFIRGetTaps32f_16s(const IppsFIRState32f_16s* pState,
    Ipp32f* pTaps);
IppStatus ippsFIRGetTaps64f_16s(const IppsFIRState64f_16s* pState,
    Ipp64f* pTaps);
```

```
IppStatus ippsFIRGetTaps64f_32s(const IppsFIRState64f_32s* pState,
    Ipp64f* pTaps);
IppStatus ippsFIRGetTaps64f_32f(const IppsFIRState64f_32f* pState,
    Ipp64f* pTaps);
IppStatus ippsFIRGetTaps_32fc(const IppsFIRState_32fc* pState,
    Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps_64fc(const IppsFIRState_64fc* pState,
    Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps32fc_16sc(const IppsFIRState32fc_16sc* pState,
    Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps64fc_16sc(const IppsFIRState64fc_16sc* pState,
    Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps64fc_32sc(const IppsFIRState64fc_32sc* pState,
    Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps64fc_32fc(const IppsFIRState64fc_32fc* pState,
    Ipp64fc* pTaps);
IppStatus ippsFIRGetTaps32s_16s32f(const IppsFIRState32s_16s* pState,
    Ipp32f* pTaps);
IppStatus ippsFIRGetTaps32sc_16sc32fc(const IppsFIRState32sc_16sc*
    pState, Ipp32fc* pTaps);
IppStatus ippsFIRGetTaps32s_16s(const IppsFIRState32s_16s* pState,
    Ipp32s* pTaps, int* tapsFactor);
IppStatus ippsFIRGetTaps32sc_16sc(const IppsFIRState32sc_16sc* pState,
    Ipp32sc* pTaps, int* tapsFactor);

IppStatus ippsFIRSetTaps_32f(const Ipp32f* pTaps,
    IppsFIRState_32f* pState);
IppStatus ippsFIRSetTaps_64f(const Ipp64f* pTaps,
    IppsFIRState_64f* pState);
IppStatus ippsFIRSetTaps32f_16s(const Ipp32f* pTaps,
    IppsFIRState32f_16s* pState);
IppStatus ippsFIRSetTaps64f_16s(const Ipp64f* pTaps,
    IppsFIRState64f_16s* pState);
IppStatus ippsFIRSetTaps64f_32s(const Ipp64f* pTaps,
    IppsFIRState64f_32s* pState);
IppStatus ippsFIRSetTaps64f_32f(const Ipp64f* pTaps,
    IppsFIRState64f_32f* pState);
IppStatus ippsFIRSetTaps_32fc(const Ipp32fc* pTaps,
    IppsFIRState_32fc* pState);
IppStatus ippsFIRSetTaps_64fc(const Ipp64fc* pTaps,
    IppsFIRState_64fc* pState);
```

```
IppStatus ippsFIRSetTaps32fc_16sc(const Ipp32fc* pTaps,
    IppsFIRState32fc_16sc* pState);
IppStatus ippsFIRSetTaps64fc_16sc(const Ipp64fc* pTaps,
    IppsFIRState64fc_16sc* pState);
IppStatus ippsFIRSetTaps64fc_32sc(const Ipp64fc* pTaps,
    IppsFIRState64fc_32sc* pState);
IppStatus ippsFIRSetTaps64fc_32fc(const Ipp64fc* pTaps,
    IppsFIRState64fc_32fc* pState);
IppStatus ippsFIRSetTaps32s_16s32f(const Ipp32f* pTaps,
    IppsFIRState32s_16s* pState);
IppStatus ippsFIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps,
    IppsFIRState32sc_16sc* pState);
IppStatus ippsFIRSetTaps32s_16s(const Ipp32s* pTaps,
    IppsFIRState32s_16s* pState, int tapsFactor);
IppStatus ippsFIRSetTaps32sc_16sc(const Ipp32sc* pTaps,
    IppsFIRState32sc_16sc* pState, int tapsFactor);
```

引数

pState	FIR フィルタ・ステート構造体へのポインタ。
pTaps	タップの値を格納した配列へのポインタ。
tapsFactor	データ型が Ipp32s のタップに対するスケール係数 (整数バージョンのみ)。

説明

関数 FIRGetTaps と ippsFIRSetTaps は、ipps.h ファイルで宣言される。

ippsFIRGetTaps。関数 ippsFIRGetTaps は、初期化された FIR ステート構造体 *pState* から長さ *tapsLen* の配列 *pTaps* にタップ値をコピーする。整数タップをスケーリングするには、*tapsFactor* 値を使用する。

ippsFIRSetTaps。関数 ippsFIRSetTaps は、以前に初期化された FIR フィルタ・ステート構造体 *pState* に新しいタップ値を設定する。新しいタップ値は、配列 *pTaps* で指定しなければならない。配列 *pTaps* の長さは、初期化されたフィルタ・ステートの *tapsLen* パラメータ値と同じでなければならない。

整数タップをスケーリングするには、*tapsFactor* 値を使用する。

関数 ippsFIRGetTaps または ippsFIRSetTaps は、初期化関数を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRGetDlyLine, FIRSetDlyLine

FIR フィルタ・ステートの遅延線の内容を
取得し設定する。

```
IppStatus ippsFIRGetDlyLine_32f(const IppsFIRState_32f* pState,
    Ipp32f* pDlyLine);
IppStatus ippsFIRGetDlyLine_64f(const IppsFIRState_64f* pState,
    Ipp64f* pDlyLine);
IppStatus ippsFIRGetDlyLine32s_16s(const IppsFIRState32s_16s* pState,
    Ipp16s* pDlyLine);
IppStatus ippsFIRGetDlyLine32f_16s(const IppsFIRState32f_16s* pState,
    Ipp16s* pDlyLine);
IppStatus ippsFIRGetDlyLine64f_16s(const IppsFIRState64f_16s* pState,
    Ipp16s* pDlyLine);
IppStatus ippsFIRGetDlyLine64f_32s(const IppsFIRState64f_32s* pState,
    Ipp32s* pDlyLine);
IppStatus ippsFIRGetDlyLine64f_32f(const IppsFIRState64f_32f* pState,
    Ipp32f* pDlyLine);

IppStatus ippsFIRGetDlyLine_32fc(const IppsFIRState_32fc* pState,
    Ipp32fc* pDlyLine);
IppStatus ippsFIRGetDlyLine_64fc(const IppsFIRState_64fc* pState,
    Ipp64fc* pDlyLine);
IppStatus ippsFIRGetDlyLine32sc_16sc(const IppsFIRState32sc_16sc* pState,
    Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine32fc_16sc(const IppsFIRState32fc_16sc* pState,
    Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_16sc(const IppsFIRState64fc_16sc*
    pState, Ipp16sc* pDlyLine);
IppStatus ippsFIRGetDlyLine64fc_32sc(const IppsFIRState64fc_32sc* pState,
    Ipp32sc* pDlyLine);
```

```
IppStatus ippsFIRGetDlyLine64fc_32fc(const IppsFIRState64fc_32fc* pState,
    Ipp32fc* pDlyLine);

IppStatus ippsFIRSetDlyLine_32f(IppsFIRState_32f* pState,
    const Ipp32f* pDlyLine);
IppStatus ippsFIRSetDlyLine_64f(IppsFIRState_64f* pState,
    const Ipp64f* pDlyLine);
IppStatus ippsFIRSetDlyLine32s_16s(IppsFIRState32s_16s* pState,
    const Ipp16s* pDlyLine);
IppStatus ippsFIRSetDlyLine32f_16s(IppsFIRState32f_16s* pState,
    const Ipp16s* pDlyLine);
IppStatus ippsFIRSetDlyLine64f_16s(IppsFIRState64f_16s* pState,
    const Ipp16s* pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32s(IppsFIRState64f_32s* pState,
    const Ipp32s* pDlyLine);
IppStatus ippsFIRSetDlyLine64f_32f(IppsFIRState64f_32f* pState,
    const Ipp32f* pDlyLine);

IppStatus ippsFIRSetDlyLine_32fc(IppsFIRState_32fc* pState,
    const Ipp32fc* pDlyLine);
IppStatus ippsFIRSetDlyLine_64fc(IppsFIRState_64fc* pState,
    const Ipp64fc* pDlyLine);
IppStatus ippsFIRSetDlyLine32sc_16sc(IppsFIRState32sc_16sc* pState,
    const Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine32fc_16sc(IppsFIRState32fc_16sc* pState,
    const Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_16sc(IppsFIRState64fc_16sc* pState,
    const Ipp16sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32sc(IppsFIRState64fc_32sc* pState,
    const Ipp32sc* pDlyLine);
IppStatus ippsFIRSetDlyLine64fc_32fc(IppsFIRState64fc_32fc* pState,
    const Ipp32fc* pDlyLine);
```

引数

pState	FIR フィルタ・ステート構造体へのポインタ。
pDlyLine	遅延線の値を格納した配列へのポインタ。

説明

関数 `ippsFIRGetDlyLine` と `ippsFIRSetDlyLine` は、`ipps.h` ファイルで宣言される。この関数は、FIR フィルタ・ステートの遅延線の値を取得し設定する。

ippsFIRGetDlyLine。関数 ippsFIRGetDlyLine は、ステート構造体 *pState* から遅延線の値をコピーし、それらを *pDlyLine* に格納する。デスティネーション配列 *pDlyLine* 内のサンプルは、ソース・ベクトル内のサンプルとは逆の順序で格納される。

ippsFIRSetDlyLine。関数 ippsFIRSetDlyLine は、*pDlyLine* から遅延線の値をコピーし、それらをステート構造体 *pState* に格納する。ソース配列 *pDlyLine* 内のサンプルは、ソース・ベクトル内のサンプルとは逆の順序で格納されている必要がある。

ippsFIRGetDlyLine または ippsFIRSetDlyLine は、関数 ippsFIRInitAlloc または ippsFIRMRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsContextMatchErr	エラー。ステート識別子が正しくない。

FIROne

FIR フィルタを使用して単一のサンプルをフィルタリングする。

```
IppStatus ippsFIROne_32f(Ipp32f src, Ipp32f* pDstVal,
    IppsFIRState_32f* pState);
IppStatus ippsFIROne_64f(Ipp64f src, Ipp64f* pDstVal,
    IppsFIRState_64f* pState);
IppStatus ippsFIROne64f_32f(Ipp32f src, Ipp32f* pDstVal,
    IppsFIRState64f_32f* pState);
IppStatus ippsFIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal,
    IppsFIRState_32fc* pState);
IppStatus ippsFIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal,
    IppsFIRState_64fc* pState);
IppStatus ippsFIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal,
    IppsFIRState64fc_32fc* pState);

IppStatus ippsFIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
    IppsFIRState32s_16s* pState, int scaleFactor);
```

```
IppStatus ippsFIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
    IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
    IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
    IppsFIRState64f_32s* pState, int scaleFactor);

IppStatus ippsFIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
    IppsFIRState64fc_32sc* pState, int scaleFactor);
```

引数

pState	FIR フィルタ・ステート構造体へのポインタ。
src	関数 ippsFIROne でフィルタリングされる入力サンプル。
pDstVal	関数 ippsFIROne でフィルタリングされる出力サンプルへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsFIROne は、ipps.h ファイルで宣言される。この関数は、シングルレート・フィルタを使用して単一のサンプル *src* をフィルタリングし、その結果を *pDstVal* に格納する。フィルタのパラメータは、*pState* で指定される。整数サンプルの出力は、*scaleFactor* に従ってスケーリングするが、飽和される場合もある。次に示す FIR フィルタの定義では、フィルタリングされるサンプルは $x(n)$ で示され、タップは $h(i)$ で示される。

戻り値 $y(n)$ は、次のシングルレート・フィルタの式で定義される。

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

ippsFIROne は、ippsFIRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。また、タップの数 *tapsLen*、*pTaps* 内のタップの値、*pDlyLine* 内の遅延線の値はあらかじめ指定しておく必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsContextMatchErr	エラー。ステート識別子が正しくない。

FIR

シングルレート・フィルタまたはマルチレート・フィルタを使用してサンプルのブロックをフィルタリングする。

```
IppStatus ippsFIR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
   int numIters, IppsFIRState_32f* pState);
IppStatus ippsFIR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
   int numIters, IppsFIRState_64f* pState);
IppStatus ippsFIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
   int numIters, IppsFIRState_32fc* pState);
IppStatus ippsFIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
   int numIters, IppsFIRState_64fc* pState);
IppStatus ippsFIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst,
   int numIters, IppsFIRState64f_32f* pState);
IppStatus ippsFIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
   int numIters, IppsFIRState64fc_32fc* pState);

IppStatus ippsFIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
   int numIters, IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
   int numIters, IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
   int numIters, IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst,
   int numIters, IppsFIRState64f_32s* pState, int scaleFactor);
IppStatus ippsFIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
   int numIters, IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
   int numIters, IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
   int numIters, IppsFIRState64fc_16sc* pState, int scaleFactor);
```


intel®

```
IppStatus ippsFIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
    int numIters, IppsFIRState64fc_32sc* pState, int scaleFactor);


IppStatus ippsFIR_32f_I(Ipp32f* pSrcDst, int numIters,
    IppsFIRState_32f* pState);
IppStatus ippsFIR_64f_I(Ipp64f* pSrcDst, int numIters,
    IppsFIRState_64f* pState);
IppStatus ippsFIR64f_32f_I(Ipp32f* pSrcDst, int numIters,
    IppsFIRState64f_32f* pState);
IppStatus ippsFIR_32fc_I(Ipp32fc* pSrcDst, int numIters,
    IppsFIRState_32fc* pState);
IppStatus ippsFIR_64fc_I(Ipp64fc* pSrcDst, int numIters,
    IppsFIRState_64fc* pState);
IppStatus ippsFIR64fc_32fc_I(Ipp32fc* pSrcDst, int numIters,
    IppsFIRState64fc_32fc* pState);


IppStatus ippsFIR32s_16s_ISfs(Ipp16s* pSrcDst, int numIters,
    IppsFIRState32s_16s* pState, int scaleFactor);
IppStatus ippsFIR32f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
    IppsFIRState32f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
    IppsFIRState64f_16s* pState, int scaleFactor);
IppStatus ippsFIR64f_32s_ISfs(Ipp32s* pSrcDst, int numIters,
    IppsFIRState64f_32s* pState, int scaleFactor);
IppStatus ippsFIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    IppsFIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsFIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    IppsFIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    IppsFIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsFIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
    IppsFIRState64fc_32sc* pState, int scaleFactor);
```

引数

pState	FIR フィルタ・ステート構造体へのポインタ。
pSrc	関数 `ippsFIR` でフィルタリングする入力配列へのポインタ。
pDst	関数 `ippsFIR` でフィルタリングされた出力配列へのポインタ。
pSrcDst	関数 `ippsIIR` でフィルタリングする入出力配列（インプレース演算用）へのポインタ。

numIters	関数 ippsFIR でフィルタリングするサンプルの数に関連付けられているパラメータ。シングルレート・フィルタの場合、入力配列内の *numIters* 個のサンプルがフィルタリングされ、その結果、*numIters* 個のサンプルを出力配列に格納する。マルチレート・フィルタの場合、入力配列内の(*numIters* * *downFactor*)個のサンプルがフィルタリングされ、その結果、(*numIters* **upFactor*)個のサンプルを出力配列に格納する。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsFIR は、ipps.h ファイルで宣言される。この関数は、シングルレート・フィルタまたはマルチレート・フィルタを使用して、入力配列 *pSrc* または *pSrcDst* をフィルタリングし、その結果をそれぞれ *pDst* または *pState* に格納する。フィルタのパラメータは、*pState* で指定される。

シングルレート・フィルタの場合、配列 *pSrc* または *pSrcDst* 内の *numIters* 個のサンプルがフィルタリングされ、その結果、*numIters* 個のサンプルを配列 *pDst* または *pSrcDst* に格納する。これは、*ippsFIROne* を numIters 回だけ連続して呼び出した場合と同じ結果になる。

次に示す FIR フィルタの定義では、フィルタリングされるサンプルは $x(n)$、タップは $h(i)$、戻り値は $y(n)$ で示される。

戻り値 $y(n)$ は、次のシングルレート・フィルタの式で定義される。

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \le n < numIters$$

関数を計算が実行すると、ステート内に格納している遅延線の値が更新される。シングルレート・フィルタの場合、*numIters* パラメータでソース配列とデスティネーション配列のサイズが決まる。

マルチレート・フィルタの場合、ippsFIR が(*numIters**downFactor*)個の入力サンプルをフィルタリングし、その結果、(*numIters**upFactor*)個のサンプルを出力配列内に格納する。マルチレート・フィルタリングは、アップサンプリング、シングルレート FIR フィルタによるフィルタリング、ダウンサンプリングの 3 つの演算のシーケンスであると見なせる。アルゴリズムは、これらの 3 つの手順を含む単一の演算として実装される。したがって、この関数では、アップサンプリングの結果を格納するサイズ(*upFactor**srcLen*)の内部バッファは生成されない。

IppsFIR は、ippsFIRInitAlloc または ippsFIRMRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。また、タップの数 *tapsLen*、*pTaps* 内のタップの値、*pDlyLine* 内の遅延線の値はあらかじめ指定しておく必要がある。

例 6-3 は、関数 ippsFIR_32f によるシングルレート・フィルタリングを示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsSizeErr	エラー。*numIters* がゼロ以下。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

例 6-3　ippsFIR 関数によるシングルレート・フィルタリング

```
IppStatus fir(void) {
#undef NUMITERS
#define NUMITERS 150
     int n;
     IppStatus status;
     IppsIIRState_32f *ictx;
     IppsFIRState_32f *fctx;
     Ipp32f *x = ippsMalloc_32f(NUMITERS),
            *y = ippsMalloc_32f(NUMITERS),
            *z = ippsMalloc_32f(NUMITERS);
     const float taps[] = {
         0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
         0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
     };
     for (n =0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *0.2);
     ippsIIRInitAlloc_32f( &ictx, taps, 10, NULL );
     ippsFIRInitAlloc_32f( &fctx, taps, 11, NULL );
     status = ippsIIR_32f( x, y, NUMITERS, ictx);
     printf_32f("IIR 32f output + 120 =", y+120, 5, status);
     ippsIIRFree_32f(ictx);
     status = ippsFIR_32f( x, z, NUMITERS, fctx );
     printf_32f("FIR 32f output + 120 =", z+120, 5, status);
     ippsFIRFree_32f(fctx);
     ippsFree(z);
     ippsFree(y);
     ippsFree(x);
     return status;
}
Output:
    IIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
    FIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
Matlab* Analog:
    >> F = 0.2; N = 150; n = 0:N-1; x = sin(2*pi*n*F);
    y = filter(fir1(10,0.15),1,x); y(121:125)
```

FIROne_Direct

FIR フィルタを使用して単一のサンプルを直接フィルタリングする。

```
IppStatus ippsFIROne_Direct_32f(Ipp32f src, Ipp32f* pDstVal,
    const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64f(Ipp64f src, Ipp64f* pDstVal,
    const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal,
    const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
    int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64fc(Ipp64fc src, Ipp64fc* pDstVal,
    const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
    int* pDlyLineIndex);
IppStatus ippsFIROne64f_Direct_32f(Ipp32f src, Ipp32f* pDstVal,
    const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
    int* pDlyLineIndex);
IppStatus ippsFIROne64fc_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal,
    const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
    int* pDlyLineIndex);


IppStatus ippsFIROne_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
    const Ipp16s* pTapsQ15, int tapsLen, Ipp16s* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
    const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
    const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
    const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
    const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIROne64fc_Direct_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
   const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32s_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
   const Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32sc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
   const Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne_Direct_32f_I(Ipp32f* pSrcDstVal,
   const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64f_I(Ipp64f* pSrcDstVal,
   const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_32fc_I(Ipp32fc* pSrcDstVal,
   const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIROne_Direct_64fc_I(Ipp64fc* pSrcDstVal,
   const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIROne64f_Direct_32f_I(Ipp32f* pSrcDstVal,
   const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIROne64fc_Direct_32fc_I(Ipp32fc* pSrcDstVal,
   const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
   int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
   const Ipp16s* pTaps, int tapsLen, Ipp16s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32f_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
   const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
   const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64f_Direct_32s_ISfs(Ipp32s* pSrcDstVal,
   const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIROne32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal,
    const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal,
    const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDstVal,
    const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32s_Direct_16s_ISfs(Ipp16s* pSrcDstVal,
    const Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIROne32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal,
    const Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
```

引数

src	関数でフィルタリングされる入力サンプル。
pDstVal	関数でフィルタリングされる出力サンプルへのポインタ。
pSrcDstVal	インプレース演算用の入力と出力サンプルへのポインタ。
pTaps	タップの値を格納した配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
pTapsQ15	Q0.15 で表現されたタップ値を格納する配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップの値を格納した配列内の要素の数。
tapsFactor	データ型が Ipp32s のタップに対するスケール係数 (整数バージョンのみ)。
pDlyLine	遅延線の値が格納された配列へのポインタ。配列内の要素の数は、*2*tapsLen* となる。ただし、この遅延線の長さは、ステート構造体を使用する FIR フィルタの場合の遅延線の長さとは異なる。
pDlyLineIndex	現在の遅延線のインデックスへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 `ippsFIROne_Direct` は、`ipps.h` ファイルで宣言される。この関数は、シングルレート・フィルタを使用して単一のサンプル *src* または *pSrcDstVal* を直接フィルタリングし、その結果を *pDstVal* または *pSrcDstVal* に格納する。
フィルタ係数（タップ）の値は、サイズ *tapsLen* の配列 *pTaps* で指定される。整数タップをスケーリングするには、*tapsFactor* 値を使用する。
tapsLen 個の入力サンプルは、サイズ *2*tapsLen* の配列 *pDlyLine* に 2 回コピーされる。直接 FIR フィルタでは、2 倍の長さの遅延線を使用して、サンプルのコピー操作の回数を減らし、フィルタ・パフォーマンスを向上させる。現在の遅延線のインデックスは、*pDlyLineIndex* で指定される。
整数サンプルの出力は、*scaleFactor* に従ってスケーリングするが、飽和される場合もある。次に示す FIR フィルタの定義では、フィルタリングされるサンプルは $x(n)$ で示され、タップは $h(i)$ で示される。

戻り値 $y(n)$ は、次のシングルレート・フィルタの式で定義される。

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。任意の指定されたポインタが NULL。
`ippStsFIRLenErr`	エラー。*tapsLen* がゼロ以下。

FIR_Direct

シングルレートの FIR フィルタを使用してサンプルのブロックを直接フィルタリングする。

```
IppStatus ippsFIR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int numIters, const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine,
    int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    int numIters, const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine,
    int* pDlyLineIndex);
```


intel®

```
IppStatus ippsFIR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int numIters, const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
    int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
    int* pDlyLineIndex);
IppStatus ippsFIR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int numIters, const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
    int* pDlyLineIndex);
IppStatus ippsFIR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
    int* pDlyLineIndex);

IppStatus ippsFIR_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int numIters, const Ipp16s* pTapsQ15, int tapsLen, Ipp16s* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int numIters, const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int numIters, const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64f_Direct_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst,
    int numIters, const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int numIters, const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
    int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32s_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor,
    Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor,
    Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
   const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64f_I(Ipp64f* pSrcDst, int numIters,
   const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
   const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters,
   const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
   const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine,
   int* pDlyLineIndex);
IppStatus ippsFIR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
   const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine,
   int* pDlyLineIndex);

IppStatus ippsFIR_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
   const Ipp16s* pTapsQ15, int tapsLen, Ipp16s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
   const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
   const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters,
   const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
   const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
   const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
IppStatus ippsFIR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
   const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIR32s_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
   const Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine,
   int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
    const Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine,
    int* pDlyLineIndex, int scaleFactor);
```

引数

pSrc	フィルタリングされる入力配列へのポインタ。
pDst	出力配列へのポインタ。
pSrcDst	インプレース演算用の入力と出力配列へのポインタ。
numIters	フィルタリングされる入力配列内のサンプルの数。
pTaps	タップの値を格納した配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
pTapsQ15	Q0.15 で表現されたタップ値を格納する配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップの値を格納した配列内の要素の数。
tapsFactor	データ型が Ipp32s のタップに対するスケール係数 (整数バージョンのみ)。
pDlyLine	遅延線の値が格納された配列へのポインタ。配列内の要素の数は、*2*tapsLen* となる。ただし、この遅延線の長さは、ステート構造体を使用する FIR フィルタの場合の遅延線の長さとは異なる。
pDlyLineIndex	現在の遅延線のインデックスへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsFIR_Direct は、ipps.h ファイルで宣言される。この関数は、シングルレート・フィルタを使用して、*numIters* 個のサンプルを格納した入力配列 *pSrc* または *pSrcDst* をフィルタリングし、得られた *numIters* 個のサンプルを *pDst* または *pSrcDst* に格納する。これは、ippsFIROne_Direct を *numIters* 回連続して呼び出した場合と同じ結果になる。

フィルタ係数（タップ）の値は、サイズ *tapsLen* の配列 *pTaps* で指定される。整数タップをスケーリングするには、*tapsFactor* 値を使用する。

tapsLen 個の入力サンプルは、サイズ *2*tapsLen* の配列 *pDlyLine* にコピーされる。直接 FIR フィルタでは、2 倍の長さの遅延線を使用して、サンプルのコピー操作の回数を減らし、フィルタ・パフォーマンスを向上させる。現在の遅延線のインデックスは、*pDlyLineIndex* で指定される。

整数サンプルの出力は、*scaleFactor* に従ってスケーリングするが、飽和される場合もある。

次に示す FIR フィルタの定義では、フィルタリングされるサンプルは *x(n)*、タップは *h(i)*、戻り値は *y(n)* で示される。

戻り値 *y(n)* は、次のシングルレート・フィルタの式で定義される。

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i), \quad 0 \le n < numIters$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsFIRLenErr	エラー。*tapsLen* がゼロ以下。
ippStsSizeErr	エラー。*len* がゼロ以下。

FIRMR_Direct

マルチレートの FIR フィルタを使用して サンプルのブロックを直接フィルタリングする。

```
IppStatus ippsFIRMR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int numIters, const Ipp32f* pTaps, int tapsLen, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    int numIters, const Ipp64f* pTaps, int tapsLen, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp64f* pDlyLine);
IppStatus ippsFIRMR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int numIters, const Ipp32fc* pTaps, int tapsLen, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp32fc* pDlyLine);
IppStatus ippsFIRMR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    int numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp64fc* pDlyLine);
IppStatus ippsFIRMR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int numIters, const Ipp64f* pTaps, int tapsLen, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    int numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor,
    int upPhase, int downFactor, int downPhase, Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMR32f_Direct_16s_Sfs(const Ipp16s* pSrc,
   Ipp16s* pDst, int numIters, const Ipp32f* pTaps, int tapsLen,
   int upFactor, int upPhase, int downFactor, int downPhase,
   Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_16s_Sfs(const Ipp16s* pSrc,
   Ipp16s* pDst, int numIters, const Ipp64f* pTaps, int tapsLen,
   int upFactor, int upPhase, int downFactor, int downPhase,
   Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_32s_Sfs(const Ipp32s* pSrc,
   Ipp32s* pDst, int numIters, const Ipp64f* pTaps, int tapsLen,
   int upFactor, int upPhase, int downFactor, int downPhase,
   Ipp32s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc,
   Ipp16sc* pDst, int numIters, const Ipp32fc* pTaps, int tapsLen,
   int upFactor, int upPhase, int downFactor, int downPhase,
   Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc,
   Ipp16sc* pDst, int numIters, const Ipp64fc* pTaps, int tapsLen,
   int upFactor, int upPhase, int downFactor, int downPhase,
   Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc,
   Ipp32sc* pDst, int numIters, const Ipp64fc* pTaps, int tapsLen,
   int upFactor, int upPhase, int downFactor, int downPhase,
   Ipp32sc* pDlyLine, int scaleFactor);


IppStatus ippsFIRMR32s_Direct_16s_Sfs(const Ipp16s* pSrc,
   Ipp16s* pDst, int numIters, const Ipp32s* pTaps, int tapsLen,
   int tapsFactor, int upFactor, int upPhase, int downFactor,
   int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc,
   Ipp16sc* pDst, int numIters, const Ipp32sc* pTaps, int tapsLen,
   int tapsFactor, int upFactor, int upPhase, int downFactor,
   int downPhase, Ipp16sc* pDlyLine, int scaleFactor)


IppStatus ippsFIRMR_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
   const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR_Direct_64f_I(Ipp64f* pSrcDst, int numIters,
   const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp64f* pDlyLine);)
IppStatus ippsFIRMR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
   const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters,
   const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp64fc* pDlyLine);
IppStatus ippsFIRMR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters,
   const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp32f* pDlyLine);
IppStatus ippsFIRMR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters,
   const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp32fc* pDlyLine);

IppStatus ippsFIRMR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
   const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters,
   const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters,
   const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp32s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
   const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
   const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
   const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
   int downFactor, int downPhase, Ipp32sc* pDlyLine, int scaleFactor);

IppStatus ippsFIRMR32s_Direct_16s_ISfs(Ipp16s* pSrcDst,
   int numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor,
   int upFactor, int upPhase, int downFactor, int downPhase,
   Ipp16s* pDlyLine, int scaleFactor);
IppStatus ippsFIRMR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst,
   int numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int
   upFactor, int upPhase, int downFactor, int downPhase,
   Ipp16sc* pDlyLine, int scaleFactor);
```

引数

pSrc	フィルタリングされる入力配列へのポインタ。
pDst	出力配列へのポインタ。
pSrcDst	インプレース演算用の入力と出力配列へのポインタ。

numIters	関数でフィルタリングするサンプルの数に関連付けられているパラメータ。入力配列内の *numIters* * *downFactor* 個のサンプルがフィルタリングされ、その結果、*numIters* * *upFactor* 個のサンプルを出力配列に格納する。
pTaps	タップの値を格納した配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップの値を格納した配列内の要素の数。
tapsFactor	データ型が Ipp32s のタップに対するスケール係数 (整数バージョンのみ)。
pDlyLine	遅延線の値が格納された配列へのポインタ。配列内の要素の数は、(*tapsLen*+*upFactor*-1) /*upFactor* で指定される。
upFactor	マルチレート信号のアップサンプリングに使用される係数。
downFactor	マルチレート信号のダウンサンプリングに使用される係数。
upPhase	マルチレート信号のアップサンプリングに使用される位相。
downPhase	マルチレート信号のダウンサンプリングに使用される位相。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsFIRMR_Direct は、ipps.h ファイルで宣言される。この関数は、マルチレート・フィルタを使用して、入力配列 *pSrc* または *pSrcDst* をフィルタリングし、得られたサンプルを *pDst* または *pSrcDst* に格納する。フィルタ係数 (タップ) の値は、サイズ *tapsLen* の配列 *pTaps* で指定される。整数タップをスケーリングするには、*tapsFactor* 値を使用する。配列 *pDlyLine* は、遅延線の値を指定する。入力配列には (*numIters*downFactor*) 個のサンプルが含まれ、出力配列は得られた (*numIters*upFactor*) 個のサンプルを格納する。
マルチレート・フィルタリングは、アップサンプリング、シングルレート FIR フィルタによるフィルタリング、ダウンサンプリングの 3 つの演算のシーケンスであると見なせる。アルゴリズムは、これらの 3 つの手順を含む単一の演算として実装される。

引数 *upFactor* は、フィルタリング対象の信号を内部的にアップサンプリングする際の係数である (5-118 ページの「ippsSampleUp」を参照)。すなわち、入力信号の各サンプル間に *upFactor*-1 個のゼロを挿入する。

引数 *upPhase* は、アップサンプリング後の入力信号のサイズ *upFactor* のブロック内にあるゼロ以外のサンプルの位置を決定するためのパラメータである。

引数 *downFactor* は、アップサンプリングした入力信号をフィルタリングして得た FIR 応答を内部でダウンサンプリングする際の係数である（5-120 ページの「ippsSampleDown」を参照）。つまり、アップサンプリング後のフィルタ応答のサイズ *downFactor* の各出力ブロックから、*downFactor*-1 個の出力サンプルを破棄する。

引数 *downPhase* は、アップサンプリング後のフィルタ応答のブロック内で、破棄されていないサンプルの位置を決定するためのパラメータである。遅延線配列 *pDlyLine* のサイズは、（*tapsLen*+*upFactor*-1）/*upFactor* となる。
整数サンプルの出力は、*scaleFactor* に従ってスケーリングするが、飽和される場合もある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsFIRLenErr	エラー。*tapsLen* がゼロ以下。
ippStsSizeErr	エラー。*numIters* がゼロ以下。
ippStsFIRMRFactorErr	エラー。*upFactor* (*downFactor*) がゼロ以下。
ippStsFIRMRPhaseErr	エラー。*upPhase*（*downPhase*）が負、または *upFactor*（*downFactor*）以上。

FIR フィルタ係数生成関数

この項では、理想的な無限フィルタ係数に窓関数を適用して、さまざまな FIR フィルタの係数（タップ値）を計算する関数について説明する。

FIRGenLowpass

ローパス FIR フィルタ係数を計算する。

```
IppStatus ippsFIRGenLowpass_64f(Ipp64f rFreq, Ipp64f* taps,
   int tapsLen, IppWinType winType, IppBool doNormal);
```

引数

rFreq	正規化された遮断周波数。周波数は、(0, 0,5) の範囲内でなければならない。
pTaps	計算されたタップ値が格納される配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップ値が格納される配列の要素の数。値は 5 以上でなければならない。
winType	計算で使用する窓関数のタイプを指定する値。*winType* には、次の値のいずれかを指定する。
	`ippWinBartlett` Bartlett（バーレット）窓関数。
	`ippWinBlackman` Blackman（ブラックマン）窓関数。
	`ippWinHamming` Hamming（ハミング）窓関数。
	`ippWinHann` Hann（ハニング）窓関数。
doNormal	正規化されたフィルタ係数のシーケンスを計算するか、または正規化されていないフィルタ係数のシーケンスを計算するかを指定する値。*doNormal* には、次の値のいずれかを指定する。
	`ippTrue` 関数は正規化された係数のシーケンスを計算する。
	`ippFalse` 関数は正規化されていない係数のシーケンスを計算する。

説明

関数 `ippsFIRGenLowpass` は、`ipps.h` ファイルで宣言される。この関数は、理想的な無限フィルタ係数に窓関数を適用して、遮断周波数 *rFreq* を持つローパス FIR フィルタの *tapsLen* 個の係数を計算する。パラメータ *winType* には、窓関数のタイプを指定する。この関数で使用する窓関数の詳細は、「窓（Window）関数」を参照のこと。計算した係数は、配列 *pTaps* に格納される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pTaps* が NULL。
`ippStsSizeErr`	エラー。*tapsLen* が 5 以下、または *rFreq* が範囲外。

FIRGenHighpass

ハイパスの FIR フィルタ係数を計算する。

```
IppStatus ippsFIRGenHighpass_64f(Ipp64f rFreq, Ipp64f* taps,
   int tapsLen, IppWinType winType, IppBool doNormal);
```

引数

rFreq	正規化された遮断周波数。 周波数は、(0, 0,5) の範囲内でなければならない。
pTaps	計算されたタップ値が格納される配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップ値が格納される配列の要素の数。値は 5 以上でなければならない。
winType	計算で使用する窓関数のタイプを指定する値。*winType* には、次の値のいずれかを指定する。

ippWinBartlett	Bartlett（バーレット）窓関数。
ippWinBlackman	Blackman（ブラックマン）窓関数。
ippWinHamming	Hamming（ハミング）窓関数。
ippWinHann	Hann（ハニング）窓関数。

doNormal	正規化されたフィルタ係数のシーケンスを計算するか、または正規化されていないフィルタ係数のシーケンスを計算するかを指定する値。*doNormal* には、次の値のいずれかを指定する。

ippTrue	関数は、正規化された係数のシーケンスを計算する。
ippFalse	関数は、正規化されていない係数のシーケンスを計算する。

説明

関数 ippsFIRGenHighpass は、ipps.h ファイルで宣言される。この関数は、理想的な無限フィルタ係数に窓関数を適用して、遮断周波数 *rFreq* を持つハイパス FIR フィルタの係数を *tapsLen* 個計算する。パラメータ *winType* には、窓関数のタイプを指定する。この関数で使用する窓関数の詳細は、「窓（Window）関数」を参照のこと。計算した係数は、配列 *pTaps* に格納される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pTaps* が NULL。
ippStsSizeErr	エラー。*tapsLen* が 5 以下、または *rFreq* が範囲外。

FIRGenBandpass

バンドパスの FIR フィルタ係数を計算する。

```
IppStatus ippsFIRGenBandpass_64f(Ipp64f rLowFreq, Ipp64f rHighFreq,
    Ipp64f* pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

引数

rLowFreq 正規化された低域遮断周波数。周波数は *rHighFreq* より小さく、(0, 0,5) の範囲内でなければならない。

rHighFreq 正規化された高域遮断周波数。周波数は *rLowFreq* より大きく、(0, 0,5) の範囲内でなければならない。

pTaps 計算されたタップ値が格納される配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。

tapsLen タップ値が格納される配列の要素の数。値は 5 以上でなければならない。

winType 計算で使用する窓関数のタイプを指定する値。*winType* には、次の値のいずれかを指定する。

ippWinBartlett	Bartlett(バーレット)窓関数。
ippWinBlackman	Blackman(ブラックマン)窓関数。
ippWinHamming	Hamming(ハミング)窓関数。
ippWinHann	Hann(ハニング)窓関数。

doNormal 正規化されたフィルタ係数のシーケンスを計算するか、または正規化されていないフィルタ係数のシーケンスを計算するかを指定する値。*doNormal* には、次の値のいずれかを指定する。

ippTrue	関数は、正規化された係数のシーケンスを計算する。

ippFalse	関数は、正規化されていない係数のシーケンスを計算する。

説明

関数 ippsFIRGenBandpass は、ipps.h ファイルで宣言される。この関数は、理想的な無限フィルタ係数に窓関数を適用して､遮断周波数 *rLowFreq* と *rHighFreq* を持つバンドパス FIR フィルタの *tapsLen* 個の係数を計算する。パラメータ *winType* には、窓関数のタイプを指定する。この関数で使用する窓関数の詳細は、「窓（Window）関数」を参照のこと。計算した係数は、配列 *pTaps* に格納される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pTaps* が NULL。
ippStsSizeErr	エラー。*tapsLen* が 5 以下、*rLowFreq* が *rHighFreq* 以上､または *rLowFreq* と *rHighFreq* のいずれかが範囲外。

FIRGenBandstop

バンドストップ FIR フィルタ係数を計算する。

```
IppStatus ippsFIRGenBandstop_64f(Ipp64f rLowFreq, Ipp64f rHighFreq,
   Ipp64f* pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

引数

rLowFreq	正規化された高域遮断周波数。周波数は *rHighFreq* より小さく、(0, 0,5) の範囲内でなければならない。
rHighFreq	正規化された高域遮断周波数。周波数は *rLowFreq* より大きく、(0, 0,5) の範囲内でなければならない。
pTaps	計算されたタップ値が格納される配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップ値が格納される配列の要素の数。値は 5 以上でなければならない。
winType	計算で使用する窓関数のタイプを指定する値。*winType* には、次の値のいずれかを指定する。

ippWinBartlett	Bartlett（バーレット）窓関数。

ippWinBlackman	Blackman（ブラックマン）窓関数。
ippWinHamming	Hamming（ハミング）窓関数。
ippWinHann	Hann（ハニング）窓関数。

doNormal 正規化されたフィルタ係数のシーケンスを計算するか、または正規化されていないフィルタ係数のシーケンスを計算するかを指定する値。*doNormal* には、次の値のいずれかを指定する。

ippTrue	関数は、正規化された係数のシーケンスを計算する。
ippFalse	関数は、正規化されていない係数のシーケンスを計算する。

説明

関数 ippsFIRGenBandstop は、ipps.h ファイルで宣言される。この関数は、理想的な無限フィルタ係数に窓関数を適用して、遮断周波数 *rLowFreq* と *rHighFreq* を持つバンドストップ FIR フィルタの *tapsLen* 個の係数を計算する。パラメータ *winType* には、窓関数のタイプを指定する。この関数で使用する窓関数の詳細は、「窓（Window）関数」を参照のこと。計算した係数は、配列 *pTaps* に格納される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pTaps* が NULL。
ippStsSizeErr	エラー。*tapsLen* が 5 以下、*rLowFreq* が *rHighFreq* 以上、または *rLowFreq* と *rHighFreq* のいずれかが範囲外。

シングルレート FIR LMS フィルタ関数

この項では、次のタスクを実行する関数について説明する。

- シングルレート FIR LMS フィルタの初期化
- 遅延線の値の取得と設定
- フィルタ係数（タップ）値の取得
- フィルタリングの実行
- 関数ステートに割り当てられた動的メモリの解放

シングルレート FIR LMS 適応フィルタ関数を使用するには、一般に次の手順を実行する。

1. `ippsFIRLMSInitAlloc` を呼び出してメモリを割り当て、シングルレート FIR LMS フィルタを初期化する。
2. `ippsFIRLMSOne_Direct` を呼び出し、1 つの入力サンプルに適応させて FIR フィルタ・タップを 1 回反復したり、`ippsFIRLMS` を呼び出し、連続する入力サンプルのブロックにタップを適応させる。
3. `ippsFIRLMSGetTaps` を呼び出し、フィルタ係数（タップ）を取得する。`ippsFIRLMSGetDlyLine` と `ippsFIRLMSSetDlyLine` を呼び出し、遅延線の値を取得し設定する。
4. `ippsFIRLMSFree` を呼び出し、FIR LMS フィルタに関連付けられている動的メモリを解放する。

FIRLMSInitAlloc

メモリを割り当て、最小平均 2 乗（LMS）アルゴリズムを使用する適応 FIR フィルタを初期化する。

```
IppStatus ippsFIRLMSInitAlloc_32f(IppsFIRLMSState_32f** pState,
    const Ipp32f* pTaps, int tapsLen, const Ipp32f* pDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSInitAlloc32f_16s(IppsFIRLMSState32f_16s** pState,
    const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine, int dlyLineIndex);
```

引数

pTaps	タップの値を格納した配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップの値を格納した配列内の要素の数。
pDlyLine	遅延線の値を格納した配列へのポインタ。配列内の要素の数は、*2*tapsLen* となる。
dlyLineIndex	遅延線の現在のインデックス。
pState	生成するステート構造体へのポインタのアドレス。

説明

関数 ippsFIRLMSInitAlloc は、ipps.h ファイルで宣言される。この関数は、メモリを割り当て、シングルレート FIR LMS フィルタ・ステートを初期化する。関数 ippsFIRLMSInitAlloc は、長さ *tapsLen* の配列 *pTaps* からステート構造体 *pState* にタップをコピーする。長さが 2**tapsLen* の配列 *pDlyLine* は、遅延線の値を指定する。遅延線 *pDlyLine* の現在のインデックスは、*dlyLineIndex* で定義される。配列 *pDlyLine* または *pTaps* へのポインタが NULL の場合、ステート構造体の対応する値はゼロに初期化される。

戻り I

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSFree

最小平均 2 乗（LMS）アルゴリズムを使用する
適応 FIR フィルタをクローズする。

```
IppStatus ippsFIRLMSFree_32f(IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMSFree32f_16s(IppsFIRLMSState32f_16s* pState);
```

引数

pState	クローズする FIR LMS フィルタ・ステート構造体へのポインタ。

説明

関数 ippsFIRLMSFree は、ipps.h ファイルで宣言される。この関数は、ippsFIRLMSInitAlloc によって作成されたフィルタ・ステートに関連するすべてのメモリを解放して、FIR LMS フィルタ・ステートをクローズする。フィルタリングが完了したら、ippsFIRLMSFree を呼び出す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。

`IppStsContextMatchErr` エラー。ステート識別子が正しくない。

FIRLMSGetTaps

FIR LMS フィルタのタップを取得する。

```
IppStatus ippsFIRLMSGetTaps_32f(const IppsFIRLMSState_32f* pState,
    Ipp32f* pOutTaps);
IppStatus ippsFIRLMSGetTaps32f_16s(const IppsFIRLMSState32f_16s*
    pState, Ipp32f* pOutTaps);
```

引数

pState	FIR LMS フィルタ・ステート構造体へのポインタ。
pOutTaps	タップのコピーが格納された配列へのポインタ。

説明

関数 `ippsFIRLMSGetTaps` は、`ipps.h` ファイルで宣言される。この関数は、ステート構造体 *pState* から長さ *tapsLen* の配列 *pOutTaps* にタップをコピーする。ステート構造体に新しいタップを設定するには、関数 `ippsFIRLMSInitAlloc` を使用して新しいステートを生成する。

関数 `ippsFIRLMSGetTaps` は、`IppsFIRLMSInitAlloc` を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	データ配列へのポインタが NULL。
`IppStsContextMatchErr`	エラー。ステート識別子が正しくない。

FIRLMSGetDlyLine, FIRLMSSetDlyLine

FIR LMS フィルタの遅延線の内容を取得し
設定する。

```
IppStatus ippsFIRLMSGetDlyLine_32f(const IppsFIRLMSState_32f* pState,
    Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSGetDlyLine32f_16s(const IppsFIRLMSState32f_16s*
    pState, Ipp16s* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine_32f(IppsFIRLMSState_32f* pState,
    const Ipp32f* pDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSSetDlyLine32f_16s(IppsFIRLMSState32f_16s* pState,
    const Ipp16s* pDlyLine, int dlyLineIndex);
```

引数

pState	FIR LMS フィルタ・ステート構造体へのポインタ。
pDlyLine	遅延線の値が格納された長さ *tapsLen* の配列へのポインタ。
pDlyLineIndex	関数 ippsFIRLMSGetDlyLine で *pState* からコピーされる現在の遅延線のインデックスを格納する配列へのポインタ。
dlyLineIndex	関数 ippsFIRLMSSetDlyLine により *pState* に格納される遅延線の最初のインデックス。

説明

関数 ippsFIRLMSGetDlyLine と ippsFIRLMSSetDlyLine は、ipps.h で宣言される。これらの関数は、FIR LMS フィルタ・ステートの遅延線の値を取得し設定する。

IppsFIRLMSGetDlyLine。関数 ippsFIRLMSGetDlyLine は、ステート構造体 *pState* から遅延線の値と現在の遅延線インデックスをコピーし、それらをそれぞれ *pDlyLine* と *pDlyLineIndex* に格納する。

ippsFIRLMSSetDlyLine。関数 ippsFIRLMSSetDlyLine は *pDlyLine* から遅延線の値を、*dlyLineIndex* から現在の遅延線インデックスをコピーし、それらをステート構造体 *pState* に格納する。

ippsFIRLMSGetDlyLine または ippsFIRLMSSetDlyLine は、関数 ippsFIRLMSInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMS

FIR LMS フィルタを使用して配列を
フィルタリングする。

```
IppStatus ippsFIRLMS_32f(const Ipp32f* pSrc, const Ipp32f* pRef,
   Ipp32f* pDst, int len, float mu, IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMS32f_16s(const Ipp16s* pSrc, const Ipp16s* pRef,
   Ipp16s* pDst, int len, float mu, IppsFIRLMSState32f_16s* pState);
```

引数

pState	FIR LMS フィルタ・ステート構造体へのポインタ。
pSrc	フィルタリングする入力サンプルへのポインタ。
pRef	参照信号へのポインタ。
pDst	出力信号へのポインタ。
len	配列内の要素の数。
mu	適応ステップ。

説明

関数 ippsFIRLMS は、ipps.h ファイルで宣言される。この関数は、適応 FIR LMS フィルタを使用して入力配列 *pSrc* をフィルタリングする。

関数により実行される *len* 回の反復は、それぞれが 2 つの主要な手順で構成する。1 つ目の手順は、ippsLMS が入力信号 *pSrc* の現在のサンプルをフィルタリングし、その結果を *pDst* に格納する。2 番目の手順は、参照信号 *pRef*、計算された結果の信号 *pDst*、適応ステップ *mu* を使用して、関数が現在のタップを更新する。

フィルタリングの手順は、FIR フィルタリング演算として記述できる。

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

ここで、フィルタリングされる入力サンプルは *x(n)*、タップは *h(i)*、戻り値は *y(n)* を示す。

この関数は、フィルタ・ステート構造体 *pState* に格納されたフィルタ係数を更新する。更新後のフィルタ係数は、次の式で定義される。

$$h_{n+1}(i) = h_n(i) + 2 \cdot mu \cdot errVal \cdot x(n-i)$$

ここで、$h_{n+1}(i)$ は新しいタップ、$h_n(i)$ は初期タップ、*mu* は適応ステップ、*errVal* は適応エラーの値を示す。適応エラーの値 *errVal* は、この関数の内部で、出力信号とリファレンス信号の差として計算される。

ippsFIRLMS は、関数 ippsFIRLMSInitAlloc を呼び出し、*pState* 構造体を初期化してから使用する必要がある。

例 6-4 は、関数 ippsFIRLMS_32f を使用して信号サンプルをフィルタリングする例を示している。

例 6-4 ippsFIRLMS 関数によるフィルタリング

```
IppStatus firlms(void) {
     IppStatus st;
     Ipp32f taps = 0, x[LEN], y[LEN], mu = 0.03f;
     IppsFIRLMSState_32f* ctx;
     int i;
     /// no taps and no delay line from outside
     ippsFIRLMSInitAlloc_32f( &ctx, 0, 1, 0, 0 );
     /// make a const signal of amplitude 1 and noise it
     for(i=0; i<LEN; ++i) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
     st = ippsFIRLMS_32f( x, x+1, y, LEN-1, mu, ctx);
     /// get FIR LMS tap, it must be near to 1
     ippsFIRLMSGetTaps_32f(ctx, &taps);
     ippsFIRLMSFree_32f(ctx);
     printf_32f("FIR LMS tap fitted =", &taps, 1, st);
     return st;
}
Output:
            FIR LMS tap adapted = 0.986842
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSOne_Direct

FIR LMS フィルタを使用して単一のサンプルをフィルタリングする。

```
IppStatus ippsFIRLMSOne_Direct_32f(Ipp32f src, Ipp32f refVal,
   Ipp32f* pDstVal, Ipp32f* pTapsInv, int tapsLen, float mu,
   Ipp32f* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSOne_Direct32f_16s(Ipp16s src, Ipp16s refVal,
   Ipp16s* pDstVal, Ipp32f* pTapsInv, int tapsLen, float mu,
   Ipp16s* pDlyLine, int* pDlyLineIndex);
IppStatus ippsFIRLMSOne_DirectQ15_16s(Ipp16s src, Ipp16s refVal,
   Ipp16s* pDstVal, Ipp32s* pTapsInv, int tapsLen, int muQ15,
   Ipp16s* pDlyLine, int* pDlyLineIndex);
```

引数

src	フィルタリングする入力サンプル。
pDstVal	出力サンプルへのポインタ。
refVal	参照する信号サンプル。
pTapsInv	適応される FIR フィルタ・タップを格納した配列へのポインタ。タップの値は、逆順で配列に格納される。
tapsLen	タップの値を格納した配列内の要素の数。
pDlyLine	遅延線の値を格納した配列へのポインタ。
pDlyLineIndex	遅延線の現在のインデックスへのポインタ。
mu	適応ステップ。
muQ15	整数バージョンの適応ステップ。

説明

関数 ippsFIRLMSOne_Direct は、ipps.h ファイルで宣言される。この関数は、FIR フィルタ・タップの適応を 1 回反復する。長さが *tapsLen* の配列 *pTapsInv* には、FIR フィルタ・タップが逆順で格納される。長さが 2**tapsLen* の配列 *pDlyLine* は、遅延線の値を指定する。*pDlyLineIndex* 配列は、遅延線の現在のインデックスを指定する。出力信号は、*pDstVal* に格納する。

注 : 適応エラーの値は、次のように計算する。*err*[*n*] = *refVal*[*n*] - **pDstVal*。

関数 ippsFIRLMSOne_Direct は、ステップ値 *mu* を使用して、FIR フィルタ・タップの適応を 1 回反復する。タップの数値は浮動小数点になる。

処理速度を上げるには、タップをゼロに設定するか、計算された値に近づける。

関数 ippsLMSOne_Direct は、入力サンプルの数に等しい反復回数のサイクル内で呼び出される。ユーザはどのデータを、フィルタリングされた出力信号またはエラーの適応の結果として格納するかを決定できる。

関数 ippsFIRLMSOne_DirectQ15 は、ステップ値 *muQ15* を使用して、FIR フィルタ・タップの適応を 1 回反復する。タップは整数値となる。適応ステップ *muQ15* は、次のように計算する。

muQ15 = (int)(*mu* * (1<<15) +0.5f)

例 6-5 は、関数 ippsFIRLMSOne_Direct を使用して、FIR フィルタ・タップを適応する例を示している。適応を実行すると、タップは 1.0 に近づく。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsSizeErr	エラー。*tapsLen* がゼロ以下。

例 6-5　ippsLMSOne_Direct 関数の使用例

```
IppStatus firlmsone(void) {
#define LEN 200
        IppStatus st = ippStsNoErr;
        Ipp32f taps = 0, dly[2] = {0};
        Ipp32f x[LEN], y[LEN], mu = 0.05f;
        int i, indx = 0;
        /// make a const signal of amplitude 1 and noise it
        for( i=0; i<LEN; ++i ) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
        for( i=0; i<LEN-1 && ippStsNoErr==st; ++i )
       st=ippsFIRLMSOne_Direct_32f( x[i], x[i+1], y+1+i, &taps, 1, mu,
        dly, &indx );
        printf_32f("FIRLMSOne tap adapted =", &taps, 1, st );
        return st;
}
Output:
     FIRLMSOne tap adapted = 0.993872
```

マルチレート FIR LMS フィルタ関数

この項では、次のタスクを実行する関数について説明する。

- マルチレート FIR LMS フィルタの初期化
- 遅延線の値の取得と設定
- フィルタ係数（タップ）値の取得と設定
- 適応ステップ値の設定
- フィルタリングの実行
- フィルタ演算の結果を使用したフィルタ係数の更新
- 関数ステートに割り当てられた動的メモリの解放

マルチレート FIR LMS 適応フィルタ関数を使用するには、一般に次の手順を実行する。

1. `ippsFIRLMSMRInitAlloc` を呼び出し、マルチレート FIR LMS フィルタする。
2. `ippsFIRLMSMRPutVal` を必要な回数だけ呼び出し、遅延線内に入力値を置く。

3. ippsFIRLMSMROne を呼び出し、遅延線内のサンプルをフィルタリングする。
4. ippsFIRLMSMRUpdateTaps を呼び出し、フィルタ後の信号とリファレンス信号の比較に基づく適応エラーの値を使用してタップを更新する。
5. ippsFIRLMSMRGetTaps と ippsFIRLMSMRSetTaps を呼び出し、フィルタ係数（タップ）を取得し設定する。ippsFIRLMSMRGetDlyLine と ippsFIRLMSMRSetDlyLine を呼び出し、遅延線の値を取得し設定する。
6. ippsFIRLMSMRFree を呼び出し、FIR LMS フィルタに関連付けられている動的メモリを解放する。

FIRLMSMRInitAlloc

メモリを割り当て、最小平均 2 乗（LMS）アルゴリズムを使用する適応マルチレート FIR フィルタを初期化する。

```
IppStatus ippsFIRLMSMRInitAlloc32s_16s(IppsFIRLMSMRState32s_16s**
    pState, const Ipp32s* pTaps, int tapsLen, const Ipp16s* pDlyLine,
    int dlyLineIndex, int dlyStep, int updateDly, int mu);
IppStatus ippsFIRLMSMRInitAlloc32sc_16sc(IppsFIRLMSMRState32sc_16sc**
    pState, const Ipp32sc* pTaps, int tapsLen, const Ipp16sc* pDlyLine,
    int dlyLineIndex, int dlyStep, int updateDly, int mu);
```

引数

pTaps	タップの値を格納した配列へのポインタ。配列内の要素の数は、*tapsLen* で指定される。
tapsLen	タップの値を格納した配列内の要素の数。
pDlyLine	遅延線の値を格納した配列へのポインタ。配列内の要素の数は、*tapsLen*dlyStep+updateDly* となる。
dlyLineIndex	遅延線の現在のインデックス。
dlyStep	遅延線の値に適用されるマルチレート・ダウンサンプリング係数。
updateDly	適応遅延の値（サンプル数単位)。
mu	適応ステップ。
pState	生成するフィルタ・ステート構造体へのポインタのアドレス。

説明

関数 ippsFIRLMSMRInitAlloc は、ipps.h ファイルで宣言される。この関数は、メモリを割り当て、マルチレート FIR LMS フィルタ・ステートを初期化する。この関数は、サイズ *tapsLen* の配列 *pTaps* からステート構造体 *pState* にフィルタ係数をコピーする。配列 *pDlyLine* は、遅延線の値を指定する。ステート構造体は、遅延線に対するダウンサンプリング係数 *dlyStep*、適応遅延値 *updateDly*、適応ステップ値 *mu* によって初期化される。この関数は、出力パラメータ *pState* にポインタを返し、操作ステータス値も返す。

この関数は、フィルタリング手順の実行中にタップ値のコピーを使用する。関数に渡される初期値は、前回のフィルタリング手順から取得される。ポインタ *pTaps* が NULL の場合、フィルタ係数の値はゼロに設定される。

フィルタリング手順には、入力サンプルのコピーが使用される。遅延線内の値は、フィルタリングされる入力データと同じ形式でデータを表現する。ポインタ *pDlyLine* が NULL の場合、フィルタの遅延線の値はゼロに設定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pState* が NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMRFree

最小平均 2 乗アルゴリズムを使用する
適応マルチレート FIR フィルタを
クローズする。

```
IppStatus ippsFIRLMSMRFree32s_16s(IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRFree32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState);
```

引数

pState	クローズされるマルチレート FIR LMS フィルタ・ステート構造体へのポインタ。

説明

関数 ippsFIRLMSMRFree は、ipps.h ファイルで宣言される。この関数は、ippsFIRLMSMRInitAlloc によって作成されたフィルタ・ステートに関連するすべてのメモリを解放して、マルチレート FIR LMS フィルタ・ステートをクローズする。ippsFIRLMSMRFree は、フィルタリングの終了後に呼び出す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pState* が NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMRSetMu

適応ステップを設定する。

```
IppStatus ippsFIRLMSMRSetMu32s_16s(IppsFIRLMSMRState32s_16s* pState,
    const int mu);
IppStatus ippsFIRLMSMRSetMu32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
    const int mu);
```

引数

mu	新しい適応ステップ。
pState	フィルタ・ステート構造体へのポインタ。

説明

関数 ippsFIRLMSMRSetMu は、ipps.h ファイルで宣言される。この関数は、*pState* に格納された適応ステップを新しい値 *mu* で更新する。

関数 ippsFIRLMSMRSetMu は、関数 ippsFIRLMSMRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pState* が NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMRUpdateTaps

適応エラーの値を使用してフィルタ係数を
更新する。

```
IppStatus ippsFIRLMSMRUpdateTaps32s_16s(Ipp32s errVal,
    IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRUpdateTaps32sc_16sc(Ipp32sc errVal,
    IppsFIRLMSMRState32sc_16sc* pState);
```

引数

pState	フィルタ・ステート構造体へのポインタ。
errVal	適応エラーの値。

説明

関数 `ippsFIRLMSMRUpdateTaps` は、`ipps.h` ファイルで宣言される。この関数は、フィルタ・ステート構造体 *pState* に格納されたフィルタ係数を更新する。この関数を呼び出す前に、フィルタ演算が実行され、適応エラーの値 *errVal* が計算されている必要がある。適応エラーの値は、この関数の外部で、出力信号とリファレンス信号の差として計算される。更新後のフィルタ係数は、次の式で定義される。

$$h_{n+1}(i) = h_n(i) + mu \cdot errVal \cdot x(n - (i \cdot dlyStep) - updateDly)$$

ここで、$h_{n+1}(i)$ は新しいタップ、$h_n(i)$ は初期タップ、*mu* は適応ステップ、*errVal* は適応エラーの値、*updateDly* は適応遅延を示す。
関数 `ippsFIRLMSMRUpdateTaps` は、関数 `ippsFIRLMSMRInitAlloc` を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pState* が NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMRGetTaps, FIRLMSMRSetTaps

マルチレート FIR LMS フィルタのタップを
取得し設定する。

```
IppStatus ippsFIRLMSMRGetTaps32s_16s(IppsFIRLMSMRState32s_16s* pState,
    Ipp32s* pOutTaps);
IppStatus ippsFIRLMSMRGetTaps32sc_16sc(IppsFIRLMSMRState32sc_16sc*
    pState, Ipp32sc* pOutTaps);
IppStatus ippsFIRLMSMRSetTaps32s_16s(IppsFIRLMSMRState32s_16s* pState,
    const Ipp32s* pInTaps);
IppStatus ippsFIRLMSMRSetTaps32sc_16sc(IppsFIRLMSMRState32sc_16sc*
    pState, const Ipp32sc* pInTaps);
```

引数

pState	フィルタ・ステート構造体へのポインタ。
pOutTaps	タップのコピーが格納された配列へのポインタ。
pInTaps	新しいタップの値のコピーが格納された配列へのポインタ。

説明

関数 ippsFIRLMSMRGetTaps と ippsFIRLMSMRSetTaps は、ipps.h ファイルで宣言される。これらの関数はフィルタ係数を取得し設定する。

ippsFIRLMSMRGetTaps。ippsFIRLMSMRGetTaps 関数は、フィルタ・ステート構造体 *pState* に格納されたフィルタ係数の値を、*pOutTaps* ポインタで指定される配列にコピーする。

ippsFIRLMSMRSetTaps。ippsFIRLMSMRSetTaps 関数は、フィルタ・ステート構造体 *pState* に格納されたフィルタ係数を、*pInTaps* ポインタで指定される配列に格納された新しい値に設定する。このポインタが NULL の場合、フィルタ係数の値はゼロに設定される。

ippsFIRLMSMRGetTaps または ippsFIRLMSMRSetTaps は、関数 ippsFIRLMSMRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。

intel®

ippStsNullPtrErr	エラー。ポインタ *pState* または *pOutTaps* が NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMRGetTapsPointer

フィルタ係数へのポインタを返す。

```
IppStatus ippsFIRLMSMRGetTapsPointer32s_16s(IppsFIRLMSMRState32s_16s*
    pState, Ipp32s** pTaps);
IppStatus ippsFIRLMSMRGetTapsPointer32sc_16sc(IppsFIRLMSMRState32sc_16sc*
    pState, Ipp32sc** pTaps);
```

引数

pState	フィルタ・ステート構造体へのポインタ。
pTaps	タップ値へのポインタを格納する変数へのポインタ。

説明

関数 ippsFIRLMSMRGetTapsPointer は、ipps.h ファイルで宣言される。この関数は、フィルタ・ステート構造体 *pState* に格納されたフィルタ係数へのポインタを、*pTaps* によって指定される変数に書き込む。

注： タップ値へのポインタを直接取得する操作は、ippsFIRLMSMRGetTaps 関数を使用してタップ値をコピーする操作より高速であるが、エラーが発生する場合がある。

関数 ippsFIRLMSMRGetTapsPointer は、関数 ippsFIRLMSMRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pState* または *pTaps* が NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMRGetDlyLine, FIRLMSMRSetDlyLine

マルチレート FIR LMS フィルタ・ステート
の遅延線の内容を取得し設定する。

```
IppStatus ippsFIRLMSMRGetDlyLine32s_16s(IppsFIRLMSMRState32s_16s*
    pState, Ipp16s* pOutDlyLine, int* pOutDlyLineIndex);
IppStatus ippsFIRLMSMRGetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc*
    pState, Ipp16sc* pOutDlyLine, int* pOutDlyLineIndex);
IppStatus ippsFIRLMSMRSetDlyLine32s_16s(IppsFIRLMSMRState32s_16s*
    pState, const Ipp16s* pInDlyLine, int dlyLineIndex);
IppStatus ippsFIRLMSMRSetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc*
    pState, const Ipp16sc* pInDlyLine, int dlyLineIndex);
```

引数

pState	フィルタ・ステート構造体へのポインタ。
pOutDlyLine	遅延線の値のコピーを格納した配列へのポインタ。配列内の要素の数は、*tapsLen*dlyStep+updateDly* で指定される。
pInDlyLine	新しい遅延線の値を格納する配列へのポインタ。配列内の要素の数は、*tapsLen*dlyStep+updateDly* で指定される。
pOutDlyLineIndex	*pState* からコピーされる現在の遅延線のインデックスを格納する配列へのポインタ。
dlyLineIndex	*pState* に格納される遅延線の最初のインデックス。

説明

関数 ippsFIRLMSMRGetDlyLine と ippsFIRLMSMSetDlyLine は、ipps.h ファイルで宣言関数される。これらの関数は、マルチレート FIR LMS フィルタ・ステートの遅延線の値を取得し設定する。

ippsFIRLMSMRGetDlyLine。ippsFIRLMSMRGetDlyLine 関数は、ステート構造体 *pState* から遅延線の値と現在の遅延線のインデックスをコピーし、それぞれ *pOutDlyLine* と *pOutDlyLineIndex* に格納する。

ippsFIRLMSMRSetDlyLine。ippsFIRLMSMRSetDlyLine 関数は、*pInDlyLine* に格納された新しい値と *dlyLineIndex* 内の対応する遅延線のインデックスをコピーし、ステート構造体 *pState* に格納する。ポインタ *pInDlyLine* が NULL の場合、遅延線の値はゼロに設定される。

ippsFIRLMSMRGetDlyLine または ippsFIRLMSMRSetDlyLine は、関数 ippsFIRLMSMRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタ *pState*、*pOutDlyLine*、*pOutDlyLineIndex* が NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMRGetDlyVal

指定された位置から 1 つの遅延線の値を取得する。

```
IppStatus ippsFIRLMSMRGetDlyVal32s_16s(IppsFIRLMSMRState32s_16s*
    pState, Ipp16s* pOutVal, int index);
IppStatus ippsFIRLMSMRGetDlyVal32sc_16sc(IppsFIRLMSMRState32sc_16sc*
    pState, Ipp16sc* pOutVal, int index);
```

引数

pState	フィルタ・ステート構造体へのポインタ。
pOutVal	コピーされた遅延線の値へのポインタ。
index	必要な遅延線の値のインデックス。

説明

関数 ippsFIRLMSMRGetDlyVal は、ipps.h ファイルで宣言される。この関数は、フィルタ・ステート構造体 *pState* から *pOurVal* に 1 つの値をコピーする。遅延線内のこのサンプルの位置は、*index* によって指定される（これは、このサンプルが遅延線内に置かれてから演算が *index* 回反復されたことを意味している）。

関数 ippsFIRLMSMRGetDlyVal は、関数 ippsFIRLMSMRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pState* または *pOutVal* が NULL。

IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMRPutVal

遅延線内に入力値を置く。

```
IppStatus ippsFIRLMSMRPutVal32s_16s(Ipp16s val,
   IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRPutVal32sc_16sc(Ipp16sc val,
   IppsFIRLMSMRState32sc_16sc* pState);
```

引数

pState	フィルタ・ステート構造体へのポインタ。
val	入力サンプルの値。

説明

関数 ippsFIRLMSMRPutVal は、ipps.h ファイルで宣言される。この関数は、入力サンプルの値 *val* を遅延線内に置き、ステート構造体 *pState* を持つフィルタによるフィルタリング手順の準備をする。

ippsFIRLMSMRPutVal は、関数 ippsFIRLMSMRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つの指定されたポインタが NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMROne

遅延線内に置かれたデータをフィルタリングする。

```
IppStatus ippsFIRLMSMROne32s_16s(Ipp32s* pDstVal,
   IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMROne32sc_16sc(Ipp32sc* pDstVal,
   IppsFIRLMSMRState32sc_16sc* pState);
```

引数

pState	フィルタ・ステート構造体へのポインタ。
pDstVal	出力信号の値へのポインタ。

説明

関数 FIRLSMMROne は、ipps.h ファイルで宣言される。この関数は、フィルタ・ステート構造体 *pState* に格納されたフィルタ係数を使用して、遅延線内に置かれたサンプルをフィルタリングする。得られた値は *pDstVal* に置かれる。ダウンサンプリング係数 *dlyStep* は、フィルタリングされるサンプルの数を指定する。フィルタ係数は更新されない。

フィルタリング手順は、次の FIR フィルタ演算として記述できる（ここで、フィルタリングされる入力サンプルは $x(n)$、タップは $h(i)$、戻り値は $y(n)$ で示される）。

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n - (i \cdot dlyStep))$$

この関数は、遅延線内に格納された値（入力サンプルのコピー）を操作する。
関数 ippsFIRLMSMROne は、ippsFIRLMSMRInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pState* または *pDstVal* が NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

FIRLMSMROneVal

1 つの入力値をフィルタリングする。

```
IppStatus ippsFIRLMSMROneVal32s_16s(Ipp16s val, Ipp32s* pDstVal,
    IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMROneVal32sc_16sc(Ipp16sc val, Ipp32sc* pDstVal,
    IppsFIRLMSMRState32sc_16sc* pState);
```

引数

pState	フィルタ・ステート構造体へのポインタ。
pDstVal	出力信号の値へのポインタ。
val	入力信号シグナルの値。

説明

関数 `ippsFIRLMSMROneVal` は、`ipps.h` ファイルで宣言される。この関数は、1 つの入力サンプル *val* を遅延線内に置き、フィルタ・ステート構造体 *pState* で指定されたフィルタ係数を使用してフィルタリングする。得られた値は、*pDstVal* に格納される。

関数 `ippsFIRLMSMROneVal` は、`ippsFIRLMSMRInitAlloc` を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

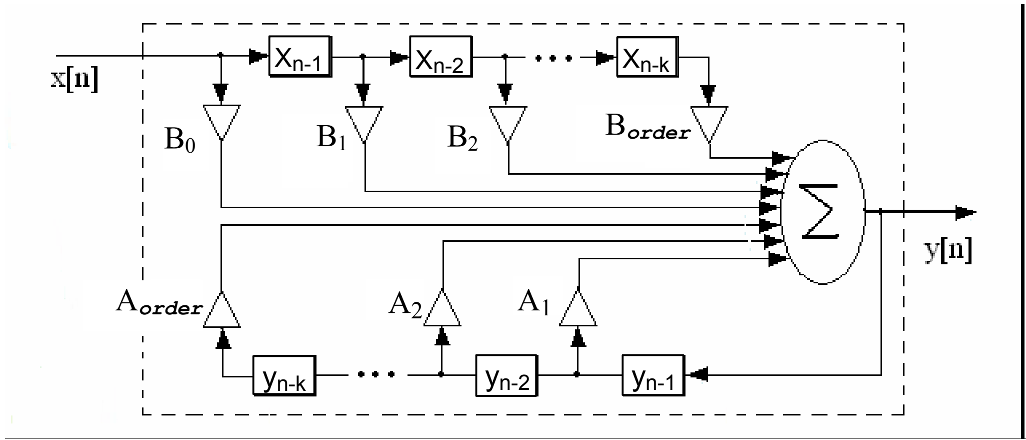
`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pState* または *pDstVal* が NULL。
`IppStsContextMatchErr`	エラー。ステート識別子が正しくない。

IIR フィルタ関数

この項では、無限インパルス応答（IIR）フィルタを初期化し、フィルタリングを実行する関数を説明する。インテル® IPP は、任意順序フィルタと BiQuad フィルタの 2 つのタイプのフィルタをサポートする。

図 6-1 は、任意順序 IIR フィルタの構造体を示す。

図 6-1　任意順序フィルタの構造体



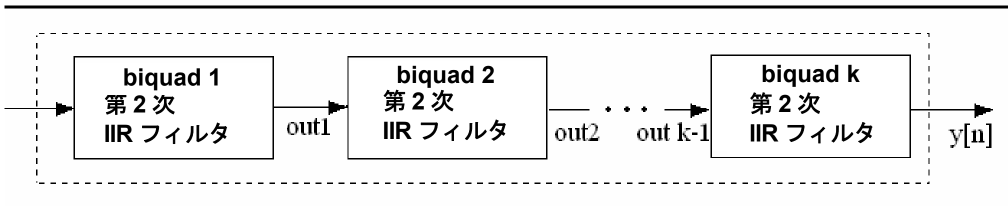
ここで、`x[n]` は入力信号のサンプル、`y[n]` は出力信号のサンプル、*order* はフィルタの順序、B_0、B_1、…、B_{order}、A_1、…、A_{order} は一定の係数 (タップ) を示す。

出力信号は、次の式で計算される。

$$y[n] = \sum_{k=0}^{order} B_k \cdot x(n-k) + \sum_{k=1}^{order} A_k \cdot x(n-k)$$

BiQuad IIR フィルタは、2 次フィルタのカスケードである。図 6-2 は、**k** 個の BiQuad フィルタの構造体を示す。

図 6-2　BiQuad IIR フィルタの構造体



IIR フィルタを初期化して使用するには、一般に次の手順を実行する。

1. `ippsIIRInitAlloc` を呼び出してメモリを割り当て、フィルタを任意順序 IIR フィルタとして初期化するか、`ippsIIRInitAlloc_BiQuad` を呼び出してメモリを割り当て、フィルタを BiQuad フィルタとして初期化する。
   あるいは、`ippsIIRInit` を呼び出して、外部バッファに格納されたフィルタを任意順序 IIR フィルタとして初期化するか、または `ippsIIRInitBiQuad` を呼び出して、外部バッファに格納されたフィルタを BiQuad フィルタとして初期化する。このバッファのサイズは、関数 `ippsIIRGetStateSize` または `ippsIIRMRGetStateSize_BiQuad` を呼び出して計算できる。

2. `ippsIIROne` を繰り返し呼び出し、IIR フィルタを使用して単一のサンプルをフィルタリングするか、`ippsIIR` を呼び出し、連続するサンプルを一度にフィルタリングする。

3. `ippsIIRGetDlyLine` と `ippsIIRSetDlyLine` を呼び出し、IIR ステート構造体の遅延線の値を取得し設定する。

4. `ippsIIRSetTaps` を呼び出して、以前に初期化したフィルタ・ステート構造体に新しいタップ値を設定する。

5. フィルタリングがすべて完了したら、`ippsIIRFree` を呼び出し、`ippsIIRInitAlloc` または `ippsIIRInitAlloc_BiQuad` で作成したフィルタ・ステート構造体に関連付けられた動的メモリを解放する。

あるいは、フィルタ関数の直接バージョンも使用できる。直接バージョンのフィルタ関数は、フィルタのステート構造体を初期化せずにフィルタリングを実行する。すべての必要なパラメータは、関数内で直接設定される。

IIRInitAlloc, IIRInitAlloc_BiQuad

メモリを割り当て、無限インパルス応答
フィルタ・ステートを初期化する。

```
IppStatus ippsIIRInitAlloc_32f(IppsIIRState_32f** pState,
   const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc_32fc(IppsIIRState_32fc** pState,
   const Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc_64f(IppsIIRState_64f** pState,
   const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc_64fc(IppsIIRState_64fc** pState,
   const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc32s_16s(IppsIIRState32s_16s** pState,
   const Ipp32s* pTaps, int order, int tapsFactor, const Ipp32s*
   pDlyLine);
IppStatus ippsIIRInitAlloc32s_16s32f(IppsIIRState32s_16s** pState,
   const Ipp32f* pTaps, int order, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_16sc(IppsIIRState32sc_16sc** pState,
   const Ipp32sc* pTaps, int order, int tapsFactor,
   const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_16sc32fc(IppsIIRState32sc_16sc** pState,
   const Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine);

IppStatus ippsIIRInitAlloc32f_16s(IppsIIRState32f_16s** pState,
   const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc32fc_16sc(IppsIIRState32fc_16sc** pState,
   const Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine);

IppStatus ippsIIRInitAlloc64f_16s(IppsIIRState64f_16s** pState,
   const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32s(IppsIIRState64f_32s** pState,
   const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_32f(IppsIIRState64f_32f** pState,
   const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_16sc(IppsIIRState64fc_16sc** pState,
   const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_32sc(IppsIIRState64fc_32sc** pState,
   const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_32fc(IppsIIRState64fc_32fc** pState,
   const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc_BiQuad_32f(IppsIIRState_32f** pState,
   const Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_32fc(IppsIIRState_32fc** pState,
   const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_64f(IppsIIRState_64f** pState,
   const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc_BiQuad_64fc(IppsIIRState_64fc** pState,
   const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc32s_BiQuad_16s(IppsIIRState32s_16s** pState,
   const Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine);
IppStatus ippsIIRInitAlloc32s_BiQuad_16s32f(IppsIIRState32s_16s**
   pState, const Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc(IppsIIRState32sc_16sc**
    pState, const Ipp32sc* pTaps, int numBq, int tapsFactor,
    const Ipp32sc* pDlyLine);
IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc**
    pState, const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine);

IppStatus ippsIIRInitAlloc32f_BiQuad_16s(IppsIIRState32f_16s** pState,
    const Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);
IppStatus ippsIIRInitAlloc32fc_BiQuad_16sc(IppsIIRState32fc_16sc**
    pState, const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);

IppStatus ippsIIRInitAlloc64f_BiQuad_16s(IppsIIRState64f_16s** pState,
    const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_BiQuad_32s(IppsIIRState64f_32s** pState,
    const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64f_BiQuad_32f(IppsIIRState64f_32f** pState,
    const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_16sc(IppsIIRState64fc_16sc**
    pState, const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_32sc(IppsIIRState64fc_32sc**
    pState, const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
IppStatus ippsIIRInitAlloc64fc_BiQuad_32fc(IppsIIRState64fc_32fc**
    pState, const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

引数

pTaps	タップが格納された配列へのポインタ。配列内の要素の数は、任意順序のフィルタの場合は 2*(*order*+1)、BQ フィルタの場合は 6**numBq* になる。
tapsFactor	データ型が Ipp32s のタップに対するスケール係数 (整数バージョンのみ)。
numBq	BiQuad 列の数。この引数は、BQ フィルタで使用される。
order	IIR フィルタの次数。この引数は、任意順序のフィルタで使用される。
pDlyLine	遅延線の値が格納された配列へのポインタ。配列内の要素の数は、任意順序のフィルタの場合は *order*、BQ フィルタの場合は 2**numBq* になる。
pState	生成する IIR ステート構造体へのポインタ。

説明

関数 ippsIIRInitAlloc と ippsIIRInitAlloc_BiQuad は、ipps.h ファイルで宣言される。この関数は、メモリを割り当て、それぞれ任意順序または BiQuad（BQ）の IIR フィルタ・ステートを初期化する。初期化関数は、配列 *pTaps* からステート構造体 *pState* にタップをコピーする。整数タップをスケーリングするには、*tapsFactor* 値を使用する。配列 *pDlyLine* は、遅延線の値を指定する。配列 *pDlyLine* へのポインタが NULL でない場合、配列の内容はコンテキスト構造体にコピーする。それ以外のときは、ステート構造体の遅延の値はゼロに設定される。

ステートが生成されない場合、初期化関数はエラー・ステータスを返す。

浮動小数点タップを使用して呼び出される、32s_32f サフィックスが付いた初期化関数は、タップを自動的に整数データ型へ変換する。

どちらの場合も、データはスケーリングを使用して整数型に変換するため、より良い精度が得られる。例 6-7 は、浮動小数点のタップを整数データ型に変換する例を示している。

ippsIIRInitAlloc。関数 ippsIIRInitAlloc は、任意順序の IIR フィルタのステート構造体に対し、そのタップと遅延線を初期化する。長さが *order* の配列 *pDlyLine* は、遅延線の値を指定する。フィルタの次数は *order* の値で定義される。ゼロ次のフィルタの場合、この値はゼロである。長さが 2*（*order* + 1）の配列 *pTaps* は、配列内に配列されるタップを次のように指定する。

$B_0, B_1, ..., B_{order}, A_0, A_1, ..., A_{order}$

$A_0 \neq 0$

ippsIIRInitAlloc_BiQuad。関数 ippsIIRInitAlloc_BiQuad は、BiQuadIIR フィルタ（すなわち、BiQuad 列により定義されるフィルタ）のステート構造体のタップと遅延線を初期化する。長さが 2**numBq* の配列 *pDlyLine* は、遅延線の値を指定する。BiQuad 列の数は、*numBq* の値で定義される。長さが 6*numBq の配列 *pTaps* は、配列内に配列されるタップを次のように指定する。

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; \quad B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; ...A_{numBq\text{-}1,2}$

$A_{n,0} \neq 0, B_{n,0} \neq 0$

戻り値

ippStsNoErr	エラーなし。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。
ippStsNullPtrErr	エラー。データ配列へのポインタが NULL。

ippStsIIROrderErr	エラー。*order* がゼロ以下、または *numBq* が 1 以下。
ippStsDivByZeroErr	エラー。A_0、$A_{n,0}$、または $B_{n,0}$ がゼロ。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

IIRFree

IIR フィルタ・ステートをクローズする。

```
IppStatus ippsIIRFree_32f(IppsIIRState_32f* pState);
IppStatus ippsIIRFree_64f(IppsIIRState_64f* pState);
IppStatus ippsIIRFree_32fc(IppsIIRState_32fc* pState);
IppStatus ippsIIRFree_64fc(IppsIIRState_64fc* pState);
IppStatus ippsIIRFree32s_16s(IppsIIRState32s_16s* pState);
IppStatus ippsIIRFree32sc_16sc(IppsIIRState32sc_16sc* pState);
IppStatus ippsIIRFree32f_16s(IppsIIRState32f_16s* pState);
IppStatus ippsIIRFree32fc_16sc(IppsIIRState32fc_16sc* pState);
IppStatus ippsIIRFree64f_16s(IppsIIRState64f_16s* pState);
IppStatus ippsIIRFree64f_32s(IppsIIRState64f_32s* pState);
IppStatus ippsIIRFree64f_32f(IppsIIRState64f_32f* pState);
IppStatus ippsIIRFree64fc_16sc(IppsIIRState64fc_16sc* pState);
IppStatus ippsIIRFree64fc_32sc(IppsIIRState64fc_32sc* pState);
IppStatus ippsIIRFree64fc_32fc(IppsIIRState64fc_32fc* pState);
```

引数

pState	クローズする IIR フィルタ・ステート構造体へのポインタ。

説明

関数 ippsIIRFree は、ipps.h ファイルで宣言される。この関数は、ippsIIRInitAlloc または ippsIIRInitAlloc_BiQuad により生成されたフィルタ・ステートに関連付けられているメモリをすべて解放して、IIR フィルタ・ステートをクローズする。フィルタリングが完了したら、ippsIIRFree を呼び出す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pState* が NULL。

IppStsContextMatchErr エラー。ステート識別子が正しくない。

IIRInit, IIRInit_BiQuad

IIR フィルタ・ステートを初期化する。

```
IppStatus ippsIIRInit_32f(IppsIIRState_32f** pState, const Ipp32f*
    pTaps, int order, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_32fc(IppsIIRState_32fc** pState, const Ipp32fc*
    pTaps, int order, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_64f(IppsIIRState_64f** pState, const Ipp64f*
    pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_64fc(IppsIIRState_64fc** pState, const Ipp64fc*
    pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32s_16s(IppsIIRState32s_16s** pState, const
    Ipp32s* pTaps, int order, int tapsFactor, const Ipp32s* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsIIRInit32s_16s32f(IppsIIRState32s_16s** pState, const
    Ipp32f* pTaps, int order, const Ipp32s* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32sc_16sc(IppsIIRState32sc_16sc** pState, const
    Ipp32sc* pTaps, int order, int tapsFactor, const Ipp32sc* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsIIRInit32sc_16sc32fc(IppsIIRState32sc_16sc** pState,
    const Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine, Ipp8u*
    pBuffer);

IppStatus ippsIIRInit32f_16s(IppsIIRState32f_16s** pState, const
    Ipp32f* pTaps, int order, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32fc_16sc(IppsIIRState32fc_16sc** pState, const
    Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_16s(IppsIIRState64f_16s** pState, const
    Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_32s(IppsIIRState64f_32s** pState, const
    Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_32f(IppsIIRState64f_32f** pState, const
    Ipp64f* pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_16sc(IppsIIRState64fc_16sc** pState, const
    Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsIIRInit64fc_32sc(IppsIIRState64fc_32sc** pState, const
    Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_32fc(IppsIIRState64fc_32fc** pState, const
    Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_BiQuad_32f(IppsIIRState_32f** pState, const
    Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_BiQuad_32fc(IppsIIRState_32fc** pState, const
    Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_BiQuad_64f(IppsIIRState_64f** pState, const
    Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit_BiQuad_64fc(IppsIIRState_64fc** pState, const
    Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32s_BiQuad_16s(IppsIIRState32s_16s** pState,
    const Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsIIRInit32sc_BiQuad_16sc(IppsIIRState32sc_16sc** pState,
    const Ipp32sc* pTaps, int numBq, int tapsFactor, const Ipp32sc*
    pDlyLine, Ipp8u* pBuffer);
IppStatus ippsIIRInit32s_BiQuad_16s32f(IppsIIRState32s_16s** pState,
    const Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine, Ipp8u*
    pBuffer);
IppStatus ippsIIRInit32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc**
    pState, const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine,
    Ipp8u* pBuffer);

IppStatus ippsIIRInit32f_BiQuad_16s(IppsIIRState32f_16s** pState,
    const Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u*
    pBuffer);
IppStatus ippsIIRInit32fc_BiQuad_16sc(IppsIIRState32fc_16sc** pState,
    const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u*
    pBuffer);

IppStatus ippsIIRInit64f_BiQuad_16s(IppsIIRState64f_16s** pState,
    const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_BiQuad_32s(IppsIIRState64f_32s** pState,
    const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsIIRInit64f_BiQuad_32f(IppsIIRState64f_32f** pState,
    const Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u*
    pBuffer);
```

```
IppStatus ippsIIRInit64fc_BiQuad_16sc(IppsIIRState64fc_16sc** pState,
    const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine,
    Ipp8u* pBuffer);
IppStatus ippsIIRInit64fc_BiQuad_32sc(IppsIIRState64fc_32sc** pState,
    const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u*
    pBuffer);
IppStatus ippsIIRInit64fc_BiQuad_32fc(IppsIIRState64fc_32fc** pState,
    const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u*
    pBuffer);
```

引数

pTaps	タップが格納された配列へのポインタ。配列内の要素の数は、任意順序のフィルタの場合は 2*(*order*+1)、BQ フィルタの場合は 6**numBq* になる。
tapsFactor	データ型が Ipp32s のタップに対するスケール係数(整数バージョンのみ)。
numBq	BiQuad 列の数。この引数は、BQ フィルタで使用される。
order	IIR フィルタの次数。この引数は、任意順序のフィルタで使用される。
pDlyLine	遅延線の値が格納された配列へのポインタ。配列内の要素の数は、任意順序のフィルタの場合は *order*、BQ フィルタの場合は 2**numBq* になる。
pState	生成する IIR ステート構造体へのポインタ。
pBuffer	IIR ステート構造体の外部バッファへのポインタ。

説明

関数 ippsIIRInit と ippsIIRInit_BiQuad は、ipps.h ファイルで宣言される。これら関数は、外部バッファに格納された任意順序または BiQuad（BQ）の IIR フィルタ・ステートをそれぞれ初期化する。バッファのサイズは、関数 IIRGetStateSize と IIRGetStateSize_BiQuad を呼び出して、あらかじめ計算する必要がある。初期化関数は、配列 *pTaps* からステート構造体 *pState* にタップをコピーする。整数タップをスケーリングするには、*tapsFactor* 値を使用する。配列 *pDlyLine* は、遅延線の値を指定する。配列 *pDlyLine* へのポインタが NULL でない場合、配列の内容はコンテキスト構造体にコピーする。それ以外のときは、ステート構造体の遅延の値はゼロに設定される。

ステートが生成されない場合、初期化関数はエラー・ステータスを返す。

浮動小数点タップを使用して呼び出される、32s_32f サフィックスが付いた初期化関数は、タップを自動的に整数データ型へ変換する。

どちらの場合も、データはスケーリングを使用して整数型に変換するため、より良い精度が得られる。例 6-7 は、浮動小数点のタップを整数データ型に変換する例を示している。

ippsIIRInit。関数 ippsIIRInit は、任意順序の IIR フィルタのステート構造体に対し、そのタップと遅延線を初期化する。長さが *order* の配列 *pDlyLine* は、遅延線の値を指定する。フィルタの次数は *order* の値で定義される。ゼロ次のフィルタの場合、この値はゼロである。長さが 2*(*order* + 1) の配列 *pTaps* は、配列内に配列されるタップを次のように指定する。

$B_0, B_1, ..., B_{order}, A_0, A_1, ..., A_{order}$

$A_0 \neq 0$

ippsIIRInit_BiQuad. 関数 ippsIIRInit_BiQuad は、BiQuadIIR フィルタ（すなわち、BiQuad 列により定義されるフィルタ）のステート構造体のタップと遅延線を初期化する。長さが 2**numBq* の配列 *pDlyLine* は、遅延線の値を指定する。BiQuad 列の数は、*numBq* の値で定義される。長さが 6*numBq の配列 *pTaps* は、配列内に配列されるタップを次のように指定する。

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2};\ \ B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; ...A_{numBq\text{-}1,2}$

$A_{n,0} \neq 0, B_{n,0} \neq 0$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsIIROrderErr	エラー。*order* がゼロ以下 または *numBq* が 1 以下。

IIRGetStateSize, IIRGetStateSize_BiQuad

IIR フィルタ・ステート構造体の長さを返す。

```
IppStatus ippsIIRGetStateSize_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64fc(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize32s_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32s_16s32f(int order, int* pBufferSize);
```

```
IppStatus ippsIIRGetStateSize32sc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32sc_16sc32fc(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize32f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32fc_16sc(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize64f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32fc(int order, int* pBufferSize);

IppStatus ippsIIRGetStateSize_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_32fc(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64f(int numBq, int* pBufferSize);
IppStatus ippsIIRGetStateSize_BiQuad_64fc(int numBq, int* pBufferSize);

IppStatus ippsIIRGetStateSize32s_BiQuad_16s(int numBq, int*
   pBufferSize);
IppStatus ippsIIRGetStateSize32s_BiQuad_16s32f(int numBq, int*
   pBufferSize);
IppStatus ippsIIRGetStateSize32sc_BiQuad_16sc(int numBq, int*
   pBufferSize);
IppStatus ippsIIRGetStateSize32sc_BiQuad_16sc32fc(int numBq, int*
   pBufferSize);

IppStatus ippsIIRGetStateSize32f_BiQuad_16s(int numBq, int*
   pBufferSize);
IppStatus ippsIIRGetStateSize32fc_BiQuad_16sc(int numBq, int*
   pBufferSize);

IppStatus ippsIIRGetStateSize64f_BiQuad_16s(int numBq, int*
   pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_32s(int numBq, int*
   pBufferSize);
IppStatus ippsIIRGetStateSize64f_BiQuad_32f(int numBq, int*
   pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_16sc(int numBq, int*
   pBufferSize);
```

```
IppStatus ippsIIRGetStateSize64fc_BiQuad_32sc(int numBq, int*
    pBufferSize);
IppStatus ippsIIRGetStateSize64fc_BiQuad_32fc(int numBq, int*
    pBufferSize);
```

引数

order	IIR フィルタの次数。この引数は、任意順序フィルタに使用される。
numBq	BiQuad 列の数。この引数は、BQ フィルタに使用される。
pBuffer	IIR ステート構造体の外部バッファへのポインタ。

説明

関数 `ippsIIRGetStateSize` と `ippsIIRGetStateSize_BiQuad` は、`ipps.h` ファイルで宣言される。これらの関数は、任意順序または BiQuad IIR フィルタ・ステートの外部バッファ・サイズを計算し、その結果を *pBufferSize* に格納する。

任意順序 IIR フィルタのバッファ・サイズを計算する場合、引数 *order* を指定する。

BiQuad IIR フィルタのバッファ・サイズを計算する場合、引数 *numBq* を指定する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pStateSize* が NULL。
ippStsIIROrderErr	エラー。*order* がゼロ以下、または *numBq* が 1 以下。

IIRSetTaps

IIR フィルタ・ステートのタップを設定する。

```
IppStatus ippsIIRSetTaps_32f(const Ipp32f* pTaps, IppsIIRState_32f*
    pState);
IppStatus ippsIIRSetTaps_32fc(const Ipp32fc* pTaps, IppsIIRState_32fc*
    pState);
IppStatus ippsIIRSetTaps_64f(const Ipp64f* pTaps, IppsIIRState_64f*
    pState);
IppStatus ippsIIRSetTaps_64fc(const Ipp64fc* pTaps, IppsIIRState_64fc*
    pState);
```

```
IppStatus ippsIIRSetTaps32s_16s(const Ipp32s* pTaps,
   IppsIIRState32s_16s* pState, int tapsFactor);
IppStatus ippsIIRSetTaps32s_16s32f(const Ipp32f* pTaps,
   IppsIIRState32s_16s* pState);
IppStatus ippsIIRSetTaps32sc_16sc(const Ipp32sc* pTaps,
   IppsIIRState32sc_16sc* pState, int tapsFactor);
IppStatus ippsIIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps,
   IppsIIRState32sc_16sc* pState);
IppStatus ippsIIRSetTaps32f_16s(const Ipp32f* pTaps,
   IppsIIRState32f_16s* pState);
IppStatus ippsIIRSetTaps32fc_16sc(const Ipp32fc* pTaps,
   IppsIIRState32fc_16sc* pState);
IppStatus ippsIIRSetTaps64f_16s(const Ipp64f* pTaps,
   IppsIIRState64f_16s* pState);
IppStatus ippsIIRSetTaps64f_32s(const Ipp64f* pTaps,
   IppsIIRState64f_32s* pState);
IppStatus ippsIIRSetTaps64f_32f(const Ipp64f* pTaps,
   IppsIIRState64f_32f* pState);
IppStatus ippsIIRSetTaps64fc_16sc(const Ipp64fc* pTaps,
   IppsIIRState64fc_16sc* pState);
IppStatus ippsIIRSetTaps64fc_32sc(const Ipp64fc* pTaps,
   IppsIIRState64fc_32sc* pState);
IppStatus ippsIIRSetTaps64fc_32fc(const Ipp64fc* pTaps,
   IppsIIRState64fc_32fc* pState);
```

引数

pTaps	タップの値を格納した配列へのポインタ。
pState	IIR フィルタ・ステート構造体へのポインタ。
tapsFactor	データ型が Ipp32s のタップに対するスケール係数 (整数バージョンのみ)。

説明

関数 ippsIIRSetTaps は、ipps.h ファイルで宣言される。この関数は、以前に初期化した IIR フィルタ・ステート構造体 *pState* に新しいタップ値を設定する。新しいタップ値は、配列 *pTaps* で指定する必要がある。整数タップをスケーリングするには、*tapsFactor* 値を使用する。

フィルタ・ステートは、関数 ippsIIRSetTaps を呼び出す前に初期化する必要がある。配列 *pTaps* の長さは、初期化されたフィルタ・ステートの *tapsLen* パラメータと同じでなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

IIRGetDlyLine, IIRSetDlyLine

IIR フィルタ・ステートの遅延線の内容を
取得し設定する。

```
IppStatus ippsIIRGetDlyLine_32f(const IppsIIRState_32f* pState,
    Ipp32f* pDlyLine);
IppStatus ippsIIRGetDlyLine_64f(const IppsIIRState_64f* pState,
    Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine_32fc(const IppsIIRState_32fc* pState,
    Ipp32fc* pDlyLine);
IppStatus ippsIIRGetDlyLine_64fc(const IppsIIRState_64fc* pState,
    Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine32f_16s(const IppsIIRState32f_16s* pState,
    Ipp32f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_16s(const IppsIIRState64f_16s* pState,
    Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32s(const IppsIIRState64f_32s* pState,
    Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine64f_32f(const IppsIIRState64f_32f* pState,
    Ipp64f* pDlyLine);
IppStatus ippsIIRGetDlyLine32s_16s(const IppsIIRState32s_16s* pState,
    Ipp32s* pDlyLine);
IppStatus ippsIIRGetDlyLine32sc_16sc(const IppsIIRState32sc_16sc*
    pState, Ipp32sc* pDlyLine);
IppStatus ippsIIRGetDlyLine32fc_16sc(const IppsIIRState32fc_16sc* pState,
    Ipp32fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_16sc(const IppsIIRState64fc_16sc*
    pState, Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32sc(const IppsIIRState64fc_32sc*
    pState, Ipp64fc* pDlyLine);
IppStatus ippsIIRGetDlyLine64fc_32fc(const IppsIIRState64fc_32fc*
    pState, Ipp64fc* pDlyLine);
```

```
IppStatus ippsIIRSetDlyLine_32f(IppsIIRState_32f* pState,
   const Ipp32f* pDlyLine);
IppStatus ippsIIRSetDlyLine_64f(IppsIIRState_64f* pState,
   const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine_32fc(IppsIIRState_32fc* pState,
   const Ipp32fc* pDlyLine);
IppStatus ippsIIRSetDlyLine_64fc(IppsIIRState_64fc* pState,
   const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine32s_16s(IppsIIRState32s_16s* pState,
   const Ipp32s* pDlyLine);
IppStatus ippsIIRSetDlyLine32f_16s(IppsIIRState32f_16s* pState,
   const Ipp32f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_16s(IppsIIRState64f_16s* pState,
   const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_32s(IppsIIRState64f_32s* pState,
   const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine64f_32f(IppsIIRState64f_32f* pState,
   const Ipp64f* pDlyLine);
IppStatus ippsIIRSetDlyLine32sc_16sc(IppsIIRState32sc_16sc* pState,
   const Ipp32sc* pDlyLine);
IppStatus ippsIIRSetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState,
   const Ipp32fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState,
   const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState,
   const Ipp64fc* pDlyLine);
IppStatus ippsIIRSetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState,
   const Ipp64fc* pDlyLine);
```

引数

pState	IIR フィルタ・ステート構造体へのポインタ。
pDlyLine	遅延線の値を格納した配列へのポインタ。配列内の要素の数は、任意順序のフィルタの場合は *order*、BQ フィルタの場合は 2**numBq* になる。ポインタが NULL の場合、ステート構造体の遅延線の値はゼロに初期化される。

説明

関数 ippsIIRGetDlyLine と ippsIIRSetDlyLine は、ipps.h ファイルで宣言される。これらの関数は、IIR フィルタ・ステートの遅延線の値を取得し設定する。

ippsIIRGetDlyLine。関数 ippsIIRGetDlyLine は、ステート構造体 *pState* から遅延線の値をコピーし、それらを配列 *pDlyLine* に格納する。

ippsIIRSetDlyLine。関数 ippsIIRSetDlyLine は、*pDlyLine* から遅延線の値をコピーし、それらをステート構造体 *pState* に格納する。

ippsIIRGetDlyLine または ippsIIRSetDlyLine は、関数 ippsIIRInitAlloc または ippsIIPBQInitAlloc を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

IIROne

IIR フィルタを使用して単一のサンプルをフィルタリングする。

```
IppStatus ippsIIROne_32f(Ipp32f src, Ipp32f* pDstVal,
   IppsIIRState_32f* pState);
IppStatus ippsIIROne_64f(Ipp64f src, Ipp64f* pDstVal,
   IppsIIRState_64f* pState);
IppStatus ippsIIROne64f_32f(Ipp32f src, Ipp32f* pDstVal,
   IppsIIRState64f_32f* pState);
IppStatus ippsIIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal,
   IppsIIRState_32fc* pState);
IppStatus ippsIIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal,
   IppsIIRState_64fc* pState);
IppStatus ippsIIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal,
   IppsIIRState64fc_32fc* pState);

IppStatus ippsIIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
   IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
   IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
   IppsIIRState64f_16s* pState, int scaleFactor);
```

```
IppStatus ippsIIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
   IppsIIRState64f_32s* pState, int scaleFactor);

IppStatus ippsIIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
   IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
   IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
   IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
   IppsIIRState64fc_32sc* pState, int scaleFactor);
```

引数

pState	IIR フィルタ・ステート構造体へのポインタ。
src	関数 ippsIIROne でフィルタリングされる入力サンプル。
pDstVal	関数 ippsIIROne でフィルタリングされた出力サンプルへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。.

説明

関数 ippsIIROne は、ipps.h ファイルで宣言される。この関数は、実数のタップを持つ IIR フィルタを使用して単一のサンプル *src* をフィルタリングし、その結果を *pDstVal* に格納する。フィルタのパラメータは、*pState* で指定される。整数サンプルの出力は、*scaleFactor* に従ってスケーリングするが、飽和される場合もある。

ステート構造体を変更しない限り、*scaleFactor* の値を変更してはならない。

関数 ippsIIR または ippsFIROne は、ippsIIRInitAlloc または ippsIIRInitAlloc_BiQuad を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。また、タップの数 *tapsLen*、*pTaps* 内のタップの値、*pDlyLine* 内の遅延線の値、*order* または *numBq* の値はあらかじめ指定しておく。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

IIR

IIR フィルタを使用してサンプルのブロックをフィルタリングする。

```
IppStatus ippsIIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    IppsIIRState_32f* pState);
IppStatus ippsIIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
    IppsIIRState_64f* pState);
IppStatus ippsIIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    IppsIIRState_32fc* pState);
IppStatus ippsIIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
    IppsIIRState_64fc* pState);
IppStatus ippsIIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    IppsIIRState64f_32f* pState);
IppStatus ippsIIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
    IppsIIRState64fc_32fc* pState);

IppStatus ippsIIR_32f_I(Ipp32f* pSrcDst, int len,
    IppsIIRState_32f* pState);
IppStatus ippsIIR_64f_I(Ipp64f* pSrcDst, int len,
    IppsIIRState_64f* pState);
IppStatus ippsIIR_32fc_I(Ipp32fc* pSrcDst, int len,
    IppsIIRState_32fc* pState);
IppStatus ippsIIR_64fc_I(Ipp64fc* pSrcDst, int len,
    IppsIIRState_64fc* pState);
IppStatus ippsIIR64f_32f_I(Ipp32f* pSrcDst, int len,
    IppsIIRState64f_32f* pState);
IppStatus ippsIIR64fc_32fc_I(Ipp32fc* pSrcDst, int len,
    IppsIIRState64fc_32fc* pState);

IppStatus ippsIIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    IppsIIRState64f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
    IppsIIRState64f_32s* pState, int scaleFactor);
```

```
IppStatus ippsIIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len, IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len, IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    int len, IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
    int len, IppsIIRState64fc_32sc* pState, int scaleFactor);
IppStatus ippsIIR32s_16s_ISfs(Ipp16s* pSrcDst, int len,
    IppsIIRState32s_16s* pState, int scaleFactor);
IppStatus ippsIIR32f_16s_ISfs(Ipp16s* pSrcDst, int len,
    IppsIIRState32f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_16s_ISfs(Ipp16s* pSrcDst, int len,
    IppsIIRState64f_16s* pState, int scaleFactor);
IppStatus ippsIIR64f_32s_ISfs(Ipp32s* pSrcDst, int len,
    IppsIIRState64f_32s* pState, int scaleFactor);
IppStatus ippsIIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
    IppsIIRState32fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
    IppsIIRState32sc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
    IppsIIRState64fc_16sc* pState, int scaleFactor);
IppStatus ippsIIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int len,
    IppsIIRState64fc_32sc* pState, int scaleFactor);
```

引数

pState	IIR フィルタ・ステート構造体へのポインタ。
pSrc	関数 ippsIIR でフィルタリングする入力配列へのポインタ。
pDst	関数 ippsIIR でフィルタリングされた出力配列へのポインタ。
pSrcDst	関数 ippsIIR でフィルタリングする入出力配列（インプレース演算用）へのポインタ。
len	関数 ippsIIR でフィルタリングするサンプルの数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsIIR は、ipps.h ファイルで宣言される。この関数は、実数タップを持つ IIR フィルタを使用して入力配列 *pSrc* または *pSrcDst* 内の *len* 個のサンプルをフィルタリングし、その結果をそれぞれ *pDst* または *pSrcDst* に格納する。フィルタのパラメータは、*pState* で指定される。整数サンプルの出力は、*scaleFactor* に従ってスケーリングされるが、飽和される場合もある。

ステート構造体を変更しない限り、*scaleFactor* の値を変更してはならない。

ippsIIR は、ippsIIRInitAlloc または ippsIIRInitAlloc_BiQuad を呼び出し、フィルタ・ステートを初期化してから呼び出す必要がある。また、タップの数 *tapsLen*、*pTaps* 内のタップの値、*pDlyLine* 内の遅延線の値、*order* または *numBq* の値はあらかじめ指定しておく。

例 6-6 は、ippsIIR_32f を使用して 60Hz の信号を除去する例を示している。

例 6-7 は、ippsIIR を使用してサンプルをフィルタリングする例を示している。関数 ippsCnvrt_64f32s_Sfs は、ippsIIRInitAlloc_32s を呼び出す前に、浮動小数点タップを整数データ型に変換する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 6-6 ippsIIR_32f 関数による 60Hz 信号の除去

```
IppStatus iir( void ) {
#undef NUMITERS
#define NUMITERS 150
     int n;
     IppStatus status;
     IppsIIRState_32f *ctx;
     Ipp32f *x = ippsMalloc_32f( NUMITERS ), *y = ippsMalloc_32f( NUMITERS );
     /// A second-order notch filter having notch freq at 60 Hz
     const float taps[] = {
        0.940809f,-1.105987f,0.940809f,1,-1.105987f,0.881618f
     };
     /// generate a signal having 60 Hz freq sampled with 400 Hz freq
     for(n=0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *60 /400);
     ippsIIRInitAlloc_32f( &ctx, taps, 2, NULL );
     status = ippsIIR_32f( x, y, NUMITERS, ctx );
     printf_32f( " IIR 32f output+120 =", y+120, 5, status );
     ippsIIRFree_32f( ctx );
     ippsFree( y );
     ippsFree( x );
     return status;
}
Output:
    IIR 32f output + 120 =  -0.000094 0.000339 0.000458 0.000208 -0.000173
Matlab* Analog:
    >> B = [0.940809,-1.105987,0.940809]; A = [1,-1.105987,0.881618];
    n = 0:150; x = sin(2*pi*n*60/400); y = filter(B,A,x); y(121:125)
```

例 6-7　ippsIIR 関数によるサンプルのフィルタリング

```
IppStatus iir16s( void ) {
#undef NUMITERS
#define NUMITERS 150
     int n, tapsfactor = 30;
     IppStatus status;
     IppsIIRState32s_16s *ctx;
     Ipp16s *x = ippsMalloc_16s( NUMITERS ), *y = ippsMalloc_16s( NUMITERS );
     /// A second-order notch filter having notch freq at 60 Hz
     Ipp64f taps[6] = {
        0.940809f,-1.105987f,0.940809f,1,-1.105987f,0.881618f
     };
     Ipp32s taps32s[6];
     Ipp64f tmax, tmp[6];
     ippsAbs_64f( taps, tmp, 6 );
     ippsMax_64f( tmp, 6, &tmax );
     tapsfactor = 0;
     if( tmax > IPP_MAX_32S )
        while( (tmax/=2) > IPP_MAX_32S ) ++tapsfactor;
     else
        while( (tmax*=2) < IPP_MAX_32S ) --tapsfactor;
     if(tapsfactor >0 )
        ippsDivC_64f_I( (float)(1<<(++tapsfactor)), taps, 6 );
     else if( tapsfactor < 0 )
        ippsMulC_64f_I( (float)(1<<(-(tapsfactor))), taps, 6 );
     ippsConvert_64f32s_Sfs( taps, taps32s, 6, ippRndNear, 0 );
     /// generate a signal of 60 Hz freq that is sampled with 400 Hz freq
     for(n=0; n<NUMITERS; ++n) x[n] = (Ipp16s)(1000*sin(IPP_2PI*n*60/400));
     ippsIIRInitAlloc32s_16s( &ctx, taps32s, 2, tapsfactor, NULL );
     status = ippsIIR32s_16s_Sfs( x, y, NUMITERS, ctx, 0 );
     printf_16s( " IIR 32s output+120 =", y+120, 5, status );
     ippsIIRFree32s_16s( ctx );
     ippsFree( y );
     ippsFree( x );
     return status;
}
Output:
    IIR 32s output + 120 =  0 0 0 0 0
```

IIROne_Direct, IIROne_BiQuadDirect

IIR フィルタを使用して単一のサンプルを
直接フィルタリングする。

```
IppStatus ippsIIROne_Direct_16s(Ipp16s src, Ipp16s* pDtsVal, const Ipp16s*
    pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippsIIROne_BiQuadDirect_16s(Ipp16s src, Ipp16s* pDtsVal,
    const Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
IppStatus ippsIIROne_Direct_16s_I(Ipp16s* pSrcDtsVal, const Ipp16s* pTaps,
    int order, Ipp32s* pDlyLine);
IppStatus ippsIIROne_BiQuadDirect_16s_I(Ipp16s* pSrcDtsVal,
    const Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
```

引数

src	関数でフィルタリングされる入力サンプル。
pDstVal	出力サンプルへのポインタ。
pSrcDstVal	インプレース演算用の入力と出力サンプルへのポインタ。
pTaps	タップが格納された配列へのポインタ。配列内の要素の数は、任意順序のフィルタの場合は 2*(*order*+1)、BQ フィルタの場合は 6**numBq* になる。
order	任意順序 IIR フィルタの次数。
numBq	BiQuad 列の数。
pDlyLine	遅延線の値が格納された配列へのポインタ。配列内の要素の数は、任意順序のフィルタの場合は *order*、BQ フィルタの場合は 2**numBq* になる。

説明

関数 `ippsIIROne_Direct` と `ippsIIROne_BiQuadDirect` は、`ipps.h` ファイルで宣言される。

`ippsIIROne_Direct`。関数 `ippsIIROne_Direct` は、任意順序 IIR フィルタを使用して単一のサンプル *src*（インプレース演算では *pSrcDstVal*）をフィルタリングし、その結果を *pDstVal*（*pSrcDstVal*）に格納する。フィルタの次数は、*order* の値で定義される。ゼロ次のフィルタの場合、この値はゼロである。長さが *order* の配列 *pDlyLine* は、遅延線の値を指定する。長さが 2*（*order* + 1）の配列 *pTaps* は、配列内に配列されるタップを次のように指定する。

$B_0, B_1, ..., B_{order}, A_0, A_1, ..., A_{order}$

値 $A_0 \geq 0$ は、その他すべてのタップのスケール係数を格納する(除数は格納しない)。

ippsIIROne_BiQuadDirect。関数 ippsIIROne_BiQuadDirect は、BiQuad IIR フィルタを使用して、単一のサンプル *src*（インプレース演算では *pSrcDstVal*）をフィルタリングし、その結果を *pDstVal*（*pSrcDstVal*）に格納する。BiQuad 列の数は、*numBq* の値で定義される。長さが 2**numBq* の配列 *pDlyLine* は、遅延線の値を指定する。長さが 6**numBq* の配列 *pTaps* は、配列内に配列されるタップを次のように指定する。

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2};\ \ B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; ...A_{numBq\text{-}1,2}$

値 $A_{n,0} \geq 0$, $B_{n,0} \geq 0$ は、各セクションにあるその他すべてのタップのスケール係数を格納する（除数は格納しない)。

戻り値

ippStsNoErr	エラーなし。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsIIROrderErr	エラー。*order* が負、または *numBq* がゼロ以下。
ippStsScaleRangeErr	エラー。A_0 または $A_{n,0}$ がゼロ以下。
IppStsContextMatchErr	エラー。ステート識別子が正しくない。

IIR_Direct, IIR_BiQuadDirect

IIR フィルタを使用してサンプルのブロックを直接フィルタリングする。

```
IppStatus ippsIIR_Direct_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    const Ipp16s* pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippsIIR_BiQuadDirect_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, const Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
IppStatus ippsIIR_Direct_16s_I(Ipp16s* pSrcDst, int len, const Ipp16s*
    pTaps, int order, Ipp32s* pDlyLine);
IppStatus ippsIIR_BiQuadDirect_16s_I(Ipp16s* pSrcDst, int len, const
    Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);
```

引数

pSrc	フィルタリングされる入力配列へのポインタ。
pDst	出力（フィルタリングされた）配列へのポインタ。
pSrcDst	関数でフィルタリングする入出力配列（インプレース演算用）へのポインタ。
len	フィルタリングするサンプルの数。
pTaps	タップが格納された配列へのポインタ。配列内の要素の数は、任意順序のフィルタの場合は 2*(*order*+1)、BQ フィルタの場合は 6*numBq になる。
order	任意順序 IIR フィルタの次数。
numBq	BiQuad 列の数。
pDlyLine	遅延線の値が格納された配列へのポインタ。配列内の要素の数は、任意順序のフィルタの場合は *order*、BQ フィルタの場合は 2*numBq になる。

説明

関数 ippsIIR_Direct と ippsIIR_BiQuadDirect は、ipps.h ファイルで宣言される。

ippsIIR_Direct。関数 ippsIIR_Direct は、入力配列 *pSrc* または *pSrcDst* 内の *len* 個のサンプルを任意順序 IIR フィルタを使用してフィルタリングし、その結果をそれぞれ *pDst* または *pSrcDst* に格納する。フィルタの次数は *order* の値で定義される。ゼロ次のフィルタの場合、この値はゼロである。長さが *order* の配列 *pDlyLine* は、遅延線の値を指定する。長さが 2*（*order* + 1）の配列 *pTaps* は、配列内に配列されるタップを次のように指定する。

$B_0, B_1, ..., B_{order}, A_0, A_1, ..., A_{order}$

値 $A_0 \geq 0$ は、その他すべてのタップのスケール係数を格納する（除数は格納しない）。

ippsIIR_BiQuadDirect. 関数 ippsIIR_BiQuadDirect は、入力配列 *pSrc* または *pSrcDst* 内の *len* 個のサンプルを BiQuad IIR フィルタを使用してフィルタリングし、その結果をそれぞれ *pDst* または *pSrcDst* に格納する。BiQuad 列の数は、*numBq* の値で定義される。長さが 2*numBq の配列 *pDlyLine* は、遅延線の値を指定する。長さが 6*numBq の配列 *pTaps* は、配列内に配列されるタップを次のように指定する。

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2};\ B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; ...A_{numBq\text{-}1,2}$

値 $A_{n,0} \geq 0, B_{n,0} \geq 0$ は、各セクションにあるその他すべてのタップのスケール係数を格納する（除数は格納しない）。

戻り値

ippStsNoErr	エラーなし。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsIIROrderErr	エラー。*order* が負、または *numBq* がゼロ以下。
ippStsScaleRangeErr	エラー。A_0 または $A_{n,0}$ がゼロ以下。

メディアン・フィルタ関数

メディアン・フィルタは、入力配列の各要素をメディアン値で置き換える処理をベースにした非線形のランクオーダ・フィルタであり、処理要素に対する固定近傍（マスク）よりも多く利用されている。メディアン・フィルタは、画像処理や信号処理の用途で幅広く使用される。メディアン・フィルタを使用すると、インパルス・ノイズを除去し、信号雑音を最小限に抑えられる。通常、マスク・サイズ（または Window 幅）は、関数の実装を容易にし、出力信号のバイアスを抑えるため、奇数値に設定する。インテル® IPP におけるメディアン関数の実装では、マスクは常に、メディアン値を計算する入力要素に中心を合わせて配置する。関数を呼び出す際に偶数のマスク・サイズを使用できるが、その場合マスク・サイズは、内部で 1 を減算して奇数に変更する。インテル IPP のメディアン関数のその他の特徴は、「境界要素」のメディアン値を決定するのに必要な入力配列の外部の要素が、入力配列の対応するエッジ要素と等しくなるように設定することである。すなわち、入力配列の外部の要素は、エッジ要素をコピーしてパディングされる。

FilterMedian

入力配列の各要素のメディアン値を計算する。

```
IppStatus ippsFilterMedian_8u(const Ipp8u* pSrc, Ipp8u* pDst,
    int len, int maskSize);
IppStatus ippsFilterMedian_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, int maskSize);
```

```
IppStatus ippsFilterMedian_32s(const Ipp32s* pSrc, Ipp32s* pDst,
    int len, int maskSize);
IppStatus ippsFilterMedian_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len, int maskSize);
IppStatus ippsFilterMedian_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    int len, int maskSize);

IppStatus ippsFilterMedian_8u_I(Ipp8u* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_16s_I(Ipp16s* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_32s_I(Ipp32s* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_32f_I(Ipp32f* pSrcDst, int len, int maskSize);
IppStatus ippsFilterMedian_64f_I(Ipp64f* pSrcDst, int len, int maskSize);
```

引数

pSrcDst	入出力配列（インプレース演算用）へのポインタ。
pSrc	関数 ippsFilterMedian でフィルタリングされる入力配列へのポインタ。
pDst	関数 ippsFilterMedian フィルタリングされた出力配列へのポインタ。
len	配列内の要素の数。
maskSize	メディアン・マスクのサイズ。この値は、正の整数でなければならない。偶数値を指定すると 1 が減算され、奇数値のフィルタ・マスクでメディアン・フィルタリングを実行する。

説明

関数 ippsFilterMedian は、ipps.h ファイルで宣言される。この関数は、入力配列 *pSrc* または *pSrcDst* の各要素のメディアン値を計算し、その結果をそれぞれ *pDst* または *pSrcDst* に格納する。

注： 存在しない点の値は、最後の点の値に等しい。例 : $x[-1]=x[0]$ または $x[len]=x[len-1]$。

例 6-8 は、ippsFilterMedian_16s_I を使用してシングルレート・フィルタリングを実行する例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsEvenMedianMaskSize	警告。メディアン・マスクの長さが偶数。

例 6-8　ippsFilterMedian 関数によるシングルレート・フィルタリング

```
void median(void) {
      Ipp16s x[8] = {1,2,127,4,5,0,7,8};
      IppStatus status = ippsFilterMedian_16s_I(x, 8, 3);
      printf_16s(“median =”, x,8, status);
}
Output:
    median =  1 2 4 5 4 5 7 8
Matlab* Analog:
    >> x = [1 2 127 4 5 0 7 8]; medfilt1(x)
```

変換関数

7

本章では、信号のフーリエ変換、離散コサイン変換 (DCT)、ヒルベルト変換、ウェーブレット変換を実行するインテル® IPP 関数について説明する。

このグループのすべての関数を表 7-1 に示す。

表 7-1 インテル ® IPP 変換関数

関数の基本名	操作
サポート関数	
ConjPerm	Perm 形式のデータを複素数データ形式に変換する。
ConjPack	Pack 形式のデータを複素数データ形式に変換する。
ConjCcs	CCS 形式のデータを複素数データ形式に変換する。
MulPack, MulPerm	Pack 形式または Perm 形式で格納された 2 つのベクトルの要素を掛ける。
MulPackConj	ベクトルの要素に、Pack 形式で格納された共役複素ベクトルの要素を掛ける。
高速フーリエ変換関数	
FFTInitAlloc_C, FFT InitAlloc_R	実数信号および複素数信号に対する高速フーリエ変換の構造体を初期化する。
FFTFree_C, FFTFree_R	実数信号および複素数信号に対する高速フーリエ変換の構造体をクローズする。
FFT GetBufSize_C, FFTGetBufSize_R	FFT 作業バッファのサイズ（バイト）を取得する。
FFTFwd_CToC, FFTInv_CToC	複素数信号に対する順方向または逆方向の高速フーリエ変換（FFT）を計算する。
FFTFwd_RToPerm, FFTInv_PermToR, FFTFwd_RToPack, FFTInv_PackToR, FFTFwd_RToCCS, FFTInv_CCS_ToR	実数信号に対する順方向または逆方向の高速フーリエ変換（FFT）を計算する。
離散フーリエ変換関数	
DFTInitAlloc_C, DFTInitAlloc_R	実数信号および複素数信号に対する離散フーリエ変換の構造体を初期化する。
DFTFree_C, DFTFree_R	実数信号および複素数信号に対する離散フーリエ変換の指定をクローズする。
DFTGetBufSize_C, DFTGetBufSize_R	DFT 作業バッファのサイズ（バイト）を取得する。
DFTFwd_CToC, DFTInv_CToC	複素数信号に対する順方向または逆方向の離散フーリエ変換（DFT）を計算する。

表 7-1 インテル® IPP 変換関数 （続き）

関数の基本名	操作
DFTFwd_RToPerm, DFTInv_PermToR, DFTFwd_RToPack, DFTInv_PackToR, DFTFwd_RToCCS, DFTInv_CCS_ToR	実数信号に対して順方向または逆方向の離散フーリエ変換（DFT）を計算する。
DFTOutOrdInitAlloc_C	異常がある離散フーリエ変換の構造体を初期化する。
DFTOutOrdFree_C	異常がある離散フーリエ変換の構造体を終了する。
DFTOutOrdGetBufSize_C	異常がある離散フーリエ変換の構造体の作業バッファのサイズを計算する。
DFTOutOrdFwd_CToC, DFTOutOrdInv_CToC	異常がある順方向または逆方向の離散フーリエ変換を計算する。
Goertz	単一の複素数信号の特定の周波数に対する DFT を計算する。
GoertzTwo	単一の複素数信号の特定周波数に対して 2 つの DFT を計算する。
離散コサイン変換（DCT）関数	
DCTFwdInitAlloc, DCTInvInitAlloc	離散コサイン変換の構造体を初期化する。
DCTFwdFree, DCTInvFree	離散コサイン変換の構造体をクローズする。
DCTFwdGetBufSize, DCTInvGetBufSize	DCT 作業バッファのサイズ（バイト）を取得する。
DCTFwd, DCTInv	信号に対する順方向または逆方向の離散コサイン変換（DCT）を計算する。
ヒルベルト変換関数	
HilbertInitAlloc	ヒルベルト変換の構造体を初期化する。
HilbertFree	ヒルベルト変換の構造体をクローズする。
Hilbert	ヒルベルト変換を使用して分析信号を計算する。
ウェーブレット変換関数	
WTHaarFwd, WTHaarInv	順方向または逆方向の単一レベル離散ウェーブレット Haar 変換を実行する。
WTFwdInitAlloc, WTInvInitAlloc	ウェーブレット変換の構造体を初期化する。
WTFwdFree, WTInvFree	ウェーブレット変換の構造体をクローズする。
WTFwd	順方向ウェーブレット変換を計算する。
WTFwdSetDlyLine, WTFwdGetDlyLine	順方向ウェーブレット変換の遅延線を設定および取得する。
WTInv	逆方向ウェーブレット変換を計算する。
WTInvSetDlyLine, WTInvGetDlyLine	逆方向ウェーブレット変換の遅延線を設定および取得する。

フーリエ変換関数

この項では、インテル® IPP のフーリエ変換関数と離散コサイン関数について説明する。インテル IPP では、信号サンプルに対して離散フーリエ変換（DFT）、高速フーリエ変換（FFT）、離散コサイン変換（DCT）を実行する関数のほか、各種のアプリケーション要件をサポートする基本関数のバリエーションも提供している。

変換サポート関数

この項では、`flag` 引数と `hint` 引数について説明する。これらの引数は、フーリエ変換関数、離散変換関数、主要なパックド形式の `Perm`、`Pack`、`CCS` で使用する。また、これらのパックド形式で格納されるデータのアンパック、変換、乗算を実行する関数でも使用される。

flag 引数と hint 引数

フーリエ変換関数には、`flag` 引数と `hint` 引数を指定する必要がある。

`flag` 引数は、結果を正規化するメソッドを指定する。表 7-2 は、`flag` 引数に指定可能な値を示す。`flag` 引数には、これらの値の 1 つだけを指定する。*A* と *B* の係数は、DFT の計算で使用される乗数を示す。

表 7-2　フーリエ変換関数の flag 引数

値	A	B	説明
IPP_FFT_DIV_FWD_BY_N	$1/N$	1	順方向変換を実行し、1/N の正規化を行う。
IPP_FFT_DIV_INV_BY_N	1	$1/N$	逆方向変換を実行し、1/N の正規化を行う。
IPP_FFT_DIV_BY_SQRTN	$1/N^{1/2}$	$1/N^{1/2}$	順方向変換と逆方向変換を実行し、1/N1/2 の正規化を行う。
IPP_FFT_NODIV_BY_ANY	1	1	順方向変換または逆方向変換を実行するが、1/N または 1/N1/2 の正規化は行わない。

`hint` 引数は、高速だが精度の低い計算、または低速だが精度の高い計算を行うための特別なコードの使用を指定する。表 7-3 は、`hint` 引数に指定可能な値を示している。

`hint` 引数をトーンの生成で使用する場合は、第 4 章の「トーン生成関数」を参照のこと。対数的な加算で使用する場合は、第 8 章の「モデル評価」を参照のこと。

表 7-3　フーリエ変換関数の hint 引数

値	説明
ippAlgHintNone	指定なし。関数は所定のアルゴリズムを選択する。
ippAlgHintFast	高速なアルゴリズムで計算するときに指定。
ippAlgHintAccurate	精度の高いアルゴリズムで計算するときに指定。

Pack 形式

Pack は、複素共役対称シーケンスをコンパクトに表現するのに便利な形式である。この形式の欠点は、実数型の FFT アルゴリズムで使用される自然な形式ではないことである（基数 2 の複素数型 FFT では、ビットの順序が逆になっている方が自然）。Pack 形式では、FFT の出力サンプルは、表 7-4 のように配列される。出力信号は、関数 ippsConjPack を使用して複素数信号にアンパックできる。詳しくは、7-7 ページの「ConjPack」を参照のこと。

Perm 形式

Perm 形式では、FFT アルゴリズムで使用される順番で値が格納される。FFT アルゴリズムを使用する場合は、この方法で値を格納するのが一番自然である。Perm 形式は、Pack 形式における任意の置換の 1 つである。Perm 形式の重要な特性の 1 つは、特定のサンプルの実数部と虚数部が隣接する必要がないことである。Perm形式では、FFT の出力サンプルは、表 7-4のように配列される。出力信号は、関数 ippsConjPerm を使用して複素数信号にアンパックできる。詳しくは、7-5 ページの「ConjPerm」を参照のこと。

CCS 形式

CCS 形式は、順方向 FFT により生成された出力複素数信号の値を最初の半分だけ格納する。CCS 形式では、FFT の出力サンプルは、表 7-4 のように配列される。CCS 形式で格納される信号は、複素数要素 1 つ分だけ長いことに注意する必要しなければならない。出力信号は、関数 ippsConjCcs を使用して複素数信号にアンパックできる。詳しくは、7-9 ページの「ConjCcs」を参照のこと。

表 7-4　Pack、Perm、CCS 形式での順方向 FFT の結果の表示

FFTReal	0	1	2	3	. . .	N-2	N-1	N	N+1
Pack	R_0	R_1	I_1	R_2	. . .	$I_{N/2-1}$	$R_{N/2}$		
Perm	R_0	$R_{N/2}$	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$		
CCS	R_0	0	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

パックド・データのアンパック

関数 ConjPerm、ConjPack、ConjCcs は、実数データを変換する場合、FFT 対称特性を使用してパックド形式から通常の複素数データ形式にデータを変換する。出力データは複素数であり、出力配列の長さは出力ベクトル内の複素数の要素数で決まる。出力配列のサイズは入力配列のサイズの 2 倍である点に注意する。CCS 形式で格納されるデータには、他の形式の場合よりも大きな配列が必要である。偶数長の配列と奇数長の配列にはそれぞれ特定の機能があるが、それらの機能については、関数ごとに個別に説明する。

ConjPerm

Perm 形式のデータを複素数データ形式に変換する。

```
IppStatus ippsConjPerm_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int
    lenDst);
IppStatus ippsConjPerm_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int
    lenDst);
IppStatus ippsConjPerm_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int
    lenDst);
IppStatus ippsConjPerm_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	ソースおよびデスティネーション・ベクトル（インプレース演算用）へのポインタ。
lenDst	ベクトル内の要素の数。

説明

関数 ippsConjPerm は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* 内にある Perm 形式のデータを複素数データ形式に変換し、その結果を *pDst* に格納する。

インプレース関数 `ippsConjPerm` は、ベクトル *pSrcDst* 内にある Perm 形式のデータを複素数データ形式に変換し、その結果を *pSrcDst* に格納する。

表 7-5 は、Perm 形式からのアンパックの例を示す。Data 列には順方向 FFT 変換でパックド・データに変換される実数入力データが入り Packed 列にはパックド実数データが入っている。Extend 列には、出力結果が複素数データ・ベクトルとして入っており、Length 列には、ベクトルの要素の数が入っている。例 7-1 は、関数 `ippsConjPerm` の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst*、*pDst*、または *pSrc* が NULL。
ippStsSizeErr	エラー。*lenDst* がゼロ以下。

表 7-5　FFT により取得されたパックド・データの例

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -7, -2, 7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

例 7-1　　ippsConjPerm 関数の使用例

```
void ConjPerm(void) {
        Ipp16s x[8] = {1,2,3,5,6,7,8,9};
        Ipp16sc zero={0,0}, y[6];
        IppStatus st;
        ippsSet_16sc( zero, y, 6 );
        st = ippsConjPerm_16sc( x, y, 6 );
        printf_16sc("Perm 6:", y, 6, st );
        ippsSet_16sc( zero, y, 6 );
        st = ippsConjPerm_16sc( x, y, 5 );
        printf_16sc("Perm 5:", y, 5, st );
}
Output:
            Perm 6:  {1,0} {3,5} {6,7} {2,0} {6,-7} {3,-5}
      Perm 5:  {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjPack

Pack 形式のデータを複素数データ形式に変換する。

```
IppStatus ippsConjPack_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int
     lenDst);
IppStatus ippsConjPack_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int
     lenDst);
IppStatus ippsConjPack_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int
     lenDst);
IppStatus ippsConjPack_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPack_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPack_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。

pSrcDst ソースおよびデスティネーション・ベクトル（インプレース演算用）へのポインタ。

lenDst ベクトル内の要素の数。

説明

関数 ippsConjPack は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* 内にある Pack 形式のデータを複素数データ形式に変換し、その結果を *pDst* に格納する。

インプレース関数 ippsConjPack は、ベクトル *pSrcDst* 内にある Pack 形式のデータを複素数データ形式に変換し、その結果を *pSrcDst* に格納する。

表 7-6 は、Pack 形式からのアンパックの例を示す。Data 列には、順方向 FFT 変換でパックド・データに変換する実数入力データが入り、Packed 列には、パックド実数データが入っている。Extended 列には、出力結果が複素数データ・ベクトルとして入っており、Length 列には、ベクトルの要素の数が入っている。例 7-2 は、関数 ippsConjPack の使用例を示す。

戻り値

ippStsNoErr エラーなし。

ippStsNullPtrErr エラー。ポインタ *pSrcDst*、*pDst*、または *pSrc* が NULL。

ippStsSizeErr エラー。*lenDst* がゼロ以下。

表 7-6 Pack 形式からのアンパックの例

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -2, 7, -7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

例 7-2 ippsConjPack 関数の使用例

```
void ConjPack(void) {
        Ipp16s x[8] = {1,2,3,5,6,7,8,9};
        Ipp16sc zero={0,0}, y[6];
        IppStatus st;
        ippsSet_16sc( zero, y, 6 );
        st = ippsConjPack_16sc( x, y, 6 );
        printf_16sc("pack 6", y, 6, st );
        ippsSet_16sc( zero, y, 6 );
        st = ippsConjPack_16sc( x, y, 5 );
        printf_16sc("pack 5", y, 5, st );
}
Output:
            Pack 6: {1,0} {2,3} {5,6} {7,0} {5,-6} {2,-3}
     Pack 5: {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

ConjCcs

CCS 形式のデータを複素数データ形式に変換する

```
IppStatus ippsConjCcs_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int
    lenDst);
IppStatus ippsConjCcs_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int
    lenDst);
IppStatus ippsConjCcs_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int
    lenDst);
IppStatus ippsConjCcs_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。

pSrcDst	ソースおよびデスティネーション・ベクトル（インプレース演算用）へのポインタ。
lenDst	ベクトル内の要素の数。

説明

関数 `ippsConjCcs` は、`ipps.h` ファイルで宣言される。この関数は、ベクトル *pSrc* 内にある CCS 形式のデータを複素数データ形式に変換し、その結果を *pDst* に格納する。

インプレース関数 `ippsConjCcs` は、ベクトル *pSrcDst* 内にある CCS 形式のデータを複素数データ形式に変換し、その結果を *pSrcDst* に格納する。

表 7-7 は、CCS 形式からのアンパックの例を示す。`Data` 列には、順方向 FFT 変換でパックド・データに変換する実数入力データが入っており、`Packed` 列には、パックド実数データが入っている。`Extended` 列には、出力結果が複素数データ・ベクトルとして入っており、`Length` 列には、ベクトルの要素の数が入っている。CCS 形式で格納されたデータは、実数要素 2 つ分だけ長い。例 7-3 は、関数 `ippsConjCcs` の使用例を示す。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcDst*、*pDst*、または *pSrc* が NULL。
`ippStsSizeErr`	エラー。*lenDst* がゼロ以下。

表 7-7 CCS 形式からのアンパックの例

Data	Packed	Extended	Length
FFT([1])	1, 0	{1, 0}	1
FFT([1 2])	3, 0, -1, 0	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, 0, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, 0, -2, 7, -7, 0	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

例 7-3　ippsConjCcs 関数の使用例

```
void ConjCcs(void) {
      Ipp16s x[8] = {1,2,3,5,6,7,8,9};
      Ipp16sc zero={0,0}, y[6];
      IppStatus st;
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjCcs_16sc( x, y, 6 );
      printf_16sc("CCS 6:", y, 6, st );
      ippsSet_16sc( zero, y, 6 );
      st = ippsConjCcs_16sc( x, y, 5 );
      printf_16sc("CCS 5:", y, 5, st );
}
Ouput:
         CCS 6:  {1,2} {3,5} {6,7} {8,9} {6,-7} {3,-5}
    CCS 5:  {1,2} {3,5} {6,7} {6,-7} {3,-5}
```

パックド・データの乗算

この項では、Pack 形式または Perm 形式で格納されたベクトルの複素数乗算を要素単位で実行する関数について説明する。これらの関数は、関数 ippsFFTFwd_RToPack と ippsFFTInv_PackToR とともに使用し、実数信号の高速たたみ込みを実行する。

標準のベクトル乗算関数 ippsMul は、次の理由で、Pack 形式または Perm 形式のベクトルの乗算には使用できない。

- Pack 形式では、2 つの実数サンプルが格納される。
- Perm 形式では、信号の実数部と、それに対応する虚数部が対にならない可能性がある。

引数 *order* は、FFT の長さである *N* に対して、底 2 の対数を指定する（$N=2^{order}$）。

MulPack, MulPerm

Pack 形式または Perm 形式で格納された
2 つのベクトルの要素を掛ける。

```
IppStatus ippsMulPack_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int length);
IppStatus ippsMulPack_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int length);
IppStatus ippsMulPack_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int length, int scaleFactor);
IppStatus ippsMulPack_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst,
    int length);
IppStatus ippsMulPack_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst,
    int length);
IppStatus ippsMulPack_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
    int length, int scaleFactor);
```

```
IppStatus ippsMulPerm_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int length);
IppStatus ippsMulPerm_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int length);
IppStatus ippsMulPerm_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int length, int scaleFactor);
IppStatus ippsMulPerm_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst,
    int length);
IppStatus ippsMulPerm_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst,
    int length);
IppStatus ippsMulPerm_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst,
    int length, int scaleFactor);
```

引数

pSrc1, pSrc2	要素が互いに掛け合わされるベクトルへのポインタ。
pDst	乗算 *pSrc1[n] * pSrc2[n]* の結果を格納するデスティネーション・ベクトルへのポインタ。
pSrc	*pSrcDst* の要素がインプレースで掛けられる要素を持つベクトルへのポインタ。
pSrcDst	ソースおよびデスティネーション・ベクトル（インプレース演算用）へのポインタ。

length	ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsMulPack と ippsMulPerm は、ipps.h ファイルで宣言される。これらは、ベクトル *pSrc1* の要素にベクトル *pSrc2* の要素を掛け、その結果を *pDst* に格納する。

ippsMulPack と ippsMulPerm（インプレース関数）は、ベクトル *pSrc* の要素にベクトル *pSrcDst* の要素を掛け、その結果を *pSrcDst* に格納する。

これらの関数は、それぞれのパックド形式に従ってパックド・データの乗算を行う。パックド形式 Perm および Pack のデータは、いくつかの実数値と複素数で構成される。したがって、実数要素に対しては実数の乗算を実行し、複素数データに対しては複素数の乗算が実行される。一般に、周波数領域で要素単位の乗算を実行する場合は、FFT 変換でフィルタリングしながら、信号に対してこのパックド・データの乗算を使用する。

ippsMulPack。関数 ippsMulPack は、Pack 形式で格納したデータの乗算を実行する。

ippsMulPerm。関数 ippsMulPerm は、Perm 形式で格納したデータの乗算を実行する。

CCS 形式で格納したベクトルを乗算するには、複素数データ乗算用の標準関数を使用する。Sfs サフィックスの付いた関数では、*scaleFactor* 値に従ってスケーリングが実行される。出力値がデータ範囲を超えると、結果が飽和される場合がある。

例 7-4 は、関数 ippsMulPack_32f_I の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst*、*pDst*、*pSrc1*、*pSrc2*、または *pSrc* が NULL。
ippStsSizeErr	エラー。*length* がゼロ以下。

例 7-4　ippsMulPack 関数の使用例

```
void mulpack( void ) {
     Ipp32f x[8], X[8], h[8]={1.0f/3,1.0f/3,1.0f/3,0,0,0,0,0}, H[8];
     IppStatus st;
     IppsFFTSpec_R_32f* spec;
     st = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
              ippAlgHintNone);
     ippsSet_32f( 3, x, 8 );
     x[3] = 5;
     st = ippsFFTFwd_RToPack_32f( x, X, spec, NULL );
     st = ippsFFTFwd_RToPack_32f( h, H, spec, NULL );
     ippsMulPack_32f_I( H, X, 8 );
     st = ippsFFTInv_PackToR_32f( X, x, spec, NULL );
     printf_32f("filtered =", x, 8, st );
     ippsFFTFree_R_32f( spec );
}
Output:
         filtered =  3.0 3.0 3.0 3.666667 3.666667 3.666667 3.0 3.0
Matlab* analog:
         >> x=3*ones(1,8); x(4)=5;h=zeros(1,8); h(1:3)=1/3;
         real(ifft(fft(x).*fft(h)))
```

MulPackConj

ベクトルの要素に、Pack 形式で格納され
た共役複素ベクトルの要素を掛ける。

```
IppStatus ippsMulPackConj_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst,
    int length);
IppStatus ippsMulPackConj_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst,
    int length);
```

引数

pSrc	1 番目のソース・ベクトルへのポインタ。

pSrcDst	2 番目のソースおよびデスティネーション・ベクトルへのポインタ。
length	ベクトル内の要素の数。

説明

関数 ippsMulPackConj は、ipps.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc* の要素に、ソース・ベクトル *pSrcDst* に対する共役複素ベクトルの要素を掛けて、その結果を *pSrcDst* に格納する。この関数は、Pack 形式で格納されたデータのインプレース演算のみを実行する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*length* がゼロ以下。

高速フーリエ変換関数

この項では、実数信号と複素数信号に対して順方向と逆方向の高速フーリエ変換（FFT）を計算する関数について説明する。FFT は離散フーリエ変換（DFT）とよく似ているが、処理がより高速化されている。FFT で変換されるベクトルの長さは、2 の累乗でなければならない。

FFT 関数を使用するには、回転因子のテーブルのようなデータを含む指定構造体を初期化する必要がある。初期化関数では、順方向と逆方向の変換について、それぞれ指定が生成される。したがって、前計算の量は削減され、全体的なパフォーマンスは向上する。

初期化関数に渡される *hint* 引数は、特別なアルゴリズム（より高速なアルゴリズムまたはより精度の高いアルゴリズム）の使用を指定する。*flag* 引数は、結果を正規化するメソッドを指定する。複素数信号は、複素数要素を含む単一の配列、または実数部と虚数部を別々に含む2つの配列として表わされる。FFT の出力結果は、Perm、Pack、または CCS 形式でパックできる。

SIMD 命令を備えたインテル・プロセッサを外部バッファとともに使用すると、FFT の処理をさらに高速化できる。外部バッファを使用すると、内部バッファの割り当て / 割り当て解除の必要がなく、キャッシュにデータを保存できるため、パフォーマンスが向上する。

FFTInitAlloc_C, FFT InitAlloc_R

実数信号および複素数信号に対する
高速フーリエ変換の構造体を初期化する。

```
IppStatus ippsFFTInitAlloc_R_16s(IppsFFTSpec_R_16s** pFFTSpec,
    int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_16s(IppsFFTSpec_C_16s** pFFTSpec,
    int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_16sc(IppsFFTSpec_C_16sc** pFFTSpec,
    int order, int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_R_32f(IppsFFTSpec_R_32f** pFFTSpec,
    int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32f(IppsFFTSpec_C_32f** pFFTSpec,
    int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_32fc(IppsFFTSpec_C_32fc** pFFTSpec,
    int order, int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_R_64f(IppsFFTSpec_R_64f** pFFTSpec,
    int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_64f(IppsFFTSpec_C_64f** pFFTSpec,
    int order, int flag, IppHintAlgorithm hint);
IppStatus ippsFFTInitAlloc_C_64fc(IppsFFTSpec_C_64fc** pFFTSpec,
    int order, int flag, IppHintAlgorithm hint);
```

引数

flag	結果を正規化するメソッドを指定する。*flag* 引数の値は、「flag 引数と hint 引数」を参照。
hint	特別なコードを使用して計算することを指定する。*hint* 引数の値は、「flag 引数と hint 引数」を参照。
order	FFT の次数。入力信号の長さは、$N=2^{order}$ で表される。
pFFTSpec	生成される FFT 指定構造体へのポインタ。

intel®

説明

関数 ippsFFTInitAlloc_C と ippsFFTInitAlloc_R は、ipps.h ファイルで宣言されます。これらの関数は、変換パラメータ *order*、正規化パラメータ *flag*、特別なコードを指定するパラメータ *hint* を使用して、FFT 指定構造体 *pFFTSpec* を生成し、初期化する。*order* 引数は、変換の長さを指定する。したがって、入力信号と出力信号は、長さ 2^{order} の配列になる。

ippsFFTInitAlloc_C。関数 ippsFFTInitAlloc_C は、複素数型 FFT の指定構造体を初期化する。

ippsFFTInitAlloc_R。関数 ippsFFTInitAlloc_R は、実数型 FFT の指定構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pFFTSpec* ポインタが NULL。
ippStsFftOrderErr	エラー。*order* の値が正しくない。
ippStsFftFlagErr	エラー。*flag* の値が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

FFTFree_C, FFTFree_R

実数信号および複素数信号に対する
高速フーリエ変換の構造体をクローズする。

```
IppStatus ippsFFTFree_R_16s(IppsFFTSpec_R_16s* pFFTSpec);
IppStatus ippsFFTFree_C_16s(IppsFFTSpec_C_16s* pFFTSpec);
IppStatus ippsFFTFree_C_16sc(IppsFFTSpec_C_16sc* pFFTSpec);

IppStatus ippsFFTFree_R_32f(IppsFFTSpec_R_32f* pFFTSpec);
IppStatus ippsFFTFree_C_32f(IppsFFTSpec_C_32f* pFFTSpec);
IppStatus ippsFFTFree_C_32fc(IppsFFTSpec_C_32fc* pFFTSpec);
IppStatus ippsFFTFree_R_64f(IppsFFTSpec_R_64f* pFFTSpec);
IppStatus ippsFFTFree_C_64f(IppsFFTSpec_C_64f* pFFTSpec);
IppStatus ippsFFTFree_C_64fc(IppsFFTSpec_C_64fc* pFFTSpec);
```

引数

pFFTSpec	クローズする FFT 指定構造体へのポインタ。

説明

関数 ippsFFTFree は、ipps.h ファイルで宣言される。この関数は、ippsFFTInit_C または ippsFFTInit_R により生成された指定に関連付けられているメモリをすべて解放すると、FFT 指定構造体 *pFFTSpec* をクローズする。変換が完了したら、ippsFFTFree を呼び出す。

ippsFFTFree_C。関数 ippsFFTFree_C は、複素数型の FFT 指定構造体をクローズする。

ippsFFTFree_R。関数 ippsFFTFree_R は、実数型の FFT 指定構造体をクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pFFTSpec* ポインタが NULL。
ippStsContextMatchErr	エラー。指定識別子 *pFFTSpec* が正しくない。

FFT GetBufSize_C, FFTGetBufSize_R

FFT 作業バッファのサイズ（バイト）を取得する。

```
IppStatus ippsFFTGetBufSize_R_16s(const IppsFFTSpec_R_16s* pFFTSpec,
    int* pSize);
IppStatus ippsFFTGetBufSize_C_16s(const IppsFFTSpec_C_16s* pFFTSpec,
    int* pSize);
IppStatus ippsFFTGetBufSize_C_16sc(const IppsFFTSpec_C_16sc* pFFTSpec,
    int* pSize);

IppStatus ippsFFTGetBufSize_R_32f(const IppsFFTSpec_R_32f* pFFTSpec,
    int* pSize);
IppStatus ippsFFTGetBufSize_C_32f(const IppsFFTSpec_C_32f* pFFTSpec,
    int* pSize);
IppStatus ippsFFTGetBufSize_C_32fc(const IppsFFTSpec_C_32fc* pFFTSpec,
    int* pSize);
```


intel®

```
IppStatus ippsFFTGetBufSize_R_64f(const IppsFFTSpec_R_64f* pFFTSpec,
    int* pSize);
IppStatus ippsFFTGetBufSize_C_64f(const IppsFFTSpec_C_64f* pFFTSpec,
    int* pSize);
IppStatus ippsFFTGetBufSize_C_64fc(const IppsFFTSpec_C_64fc* pFFTSpec,
    int* pSize);
```

引数

pFFTSpec	FFT 指定構造体へのポインタ。
pSize	FFT 作業バッファ・サイズ値へのポインタ。

説明

関数 ippsFFTGetBufSize は、ipps.h ファイルで宣言される。この関数は、指定構造体 *pFFTSpec* により記述されている FFT 作業バッファ・サイズ（バイト）を取得し、それを *pSize* に格納する。

ippsFFTGetBufSize_C。関数 ippsFFTGetBufSize_C は、複素数型 FFT の作業バッファ・サイズを取得する。

ippsFFTGetBufSize_R。関数 ippsFFTGetBufSize_R は、実数型 FFT の作業バッファ・サイズを取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pFFTSpec* ポインタが NULL。
ippStsContextMatchErr	エラー。指定識別子 *pFFTSpec* が正しくない。

FFTFwd_CToC, FFTInv_CToC

複素数信号に対する順方向または逆方向の
高速フーリエ変換（FFT）を計算する。

```
IppStatus ippsFFTFwd_CToC_32f(const Ipp32f* pSrcRe,
    const Ipp32f* pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm,
    const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32f(const Ipp32f* pSrcRe,
    const Ipp32f* pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm,
    const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_64f(const Ipp64f* pSrcRe,
    const Ipp64f* pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm,
    const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_64f(const Ipp64f* pSrcRe,
    const Ipp64f* pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm,
    const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    const IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    const IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    const IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    const IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe,
    const Ipp16s* pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm, const
    IppsFFTSpec_C_16s* pFFTSpecx, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe,
    const Ipp16s* pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm, const
    IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    const IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsFFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    const IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
```

引数

pFFTSpec	FFT 指定構造体へのポインタ。
pSrc	複素数値が格納された入力配列へのポインタ。
pDst	複素数値が格納された出力配列へのポインタ。
pSrcRe	信号の実数部が格納された入力配列へのポインタ。
pSrcIm	信号の虚数部が格納された入力配列へのポインタ。
pDstRe	信号の実数部が格納された出力配列へのポインタ。
pDstIm	信号の虚数部が格納された出力配列へのポインタ。
pBuffer	FFT 作業バッファへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsFFTFwd_CToC と ippsFFTInv_CToC は、ipps.h ファイルで宣言される。これらの関数は、*pFFTSpec* 指定パラメータ（変換パラメータ order、正規化パラメータ *flag*、特別なコードを指定するパラメータ *hint*）に従って、複素数信号に対する順方向または逆方向の FFT を計算する。

複素数データ型を使用する関数（32fc サフィックスが付いた関数など）は、入力複素数配列 *pSrc* を処理し、その結果を *pDst* に格納する。

実数データ型を使用し、独立した実数部 *pSrcRe* と虚数部 *pSrcIm* で表現される複素数信号を処理する関数（32f サフィックスが付いた関数など）は、結果をそれぞれ *pDstRe* と *pDstIm* に格納する。

整数データ型の場合、出力結果は *scaleFactor* 値に従ってスケーリングされ、出力信号の範囲と精度を保持する。*pBuffer* 引数は、必要な作業メモリを FFT 関数に提供し、関数内のメモリ割り当てが行われないようにする。バッファによりパフォーマンスが向上するのは、キャッシュに保存された前の演算結果を FFT 関数が入力配列として使用する場合である。バッファは、連続する 2 つの演算で同じである。

ippsFFTFwd_CToC。関数 ippsFFTFwd_CToC は、複素数型の順方向 FFT を計算する。FFT の長さは、2 の累乗でなければならない。

ippsFFTInv_CToC。関数 ippsFFTInv_CToC は、複素数型の逆方向 FFT を計算する。FFT の長さは、2 の累乗でなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。データ配列へのポインタが NULL。
ippStsContextMatchErr	エラー。指定識別子 *pFFTSpec* が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

FFTFwd_RToPerm, FFTInv_PermToR, FFTFwd_RToPack, FFTInv_PackToR, FFTFwd_RToCCS, FFTInv_CCS ToR

実数信号に対する順方向または逆方向の
高速フーリエ変換（FFT）を計算する。

```
IppStatus ippsFFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
IppStatus ippsFFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsFFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
```


intel®

```
IppStatus ippsFFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsFFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsFFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsFFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
```

引数

pFFTSpec	FFT 指定構造体へのポインタ。
pSrc	順方向変換で生成した実数値と逆方向変換で生成したパックド複素数値が格納された入力配列へのポインタ。
pDst	逆方向変換で生成した実数値と順方向変換で生成したパックド複素数値が格納された出力配列へのポインタ。
pBuffer	作業バッファへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

これらの関数は ipps.h ファイルで宣言される。これらの関数は、*pFFTSpec* 指定パラメータ（変換パラメータ *order*、正規化パラメータ *flag*、特別なコードを指定するパラメータ *hint*）に従って、実数信号に対する順方向または逆方向の FFT を計算する。

実数信号に対する順方向変換の結果（すなわち、周波数領域内の結果）は、Pack、Perm、または CCS のパックド形式で表現される。データはパック可能だが、これは、実数信号に対する FFT 変換が対称特性を持つためである。

整数データ型の場合、出力結果は *scaleFactor* 値に従ってスケーリングされるので、出力信号の範囲と精度は保持される。*pBuffer* 引数は、必要な作業メモリを FFT 関数に提供し、関数内のメモリ割り当てが行われないようにする。バッファによりパフォーマンスが向上するのは、キャッシュに保存された前の演算結果を FFT 関数が入力配列として使用する場合である。2 つの連続する演算で同じバッファが使用される。

ippsFFTFwd_RToPerm, **ippsFFTInv_PermToR**。これらの関数は、順方向または逆方向の FFT を計算し、その結果を `Perm` 形式で格納する。FFT の長さは、2 の累乗でなければならない。

ippsFFTFwd_RToPack, **ipps FFT Inv_PackToR**。これらの関数は、順方向または逆方向の FFT を計算し、その結果を `Pack` 形式で格納する。FFT の長さは、2 の累乗でなければならない。

ippsFFTFwd_RToCCS , **ipps FFT Inv_CCS ToR**。これらの関数は、順方向または逆方向の FFT を計算し、その結果を `CCS` 形式で格納する。FFT の長さは、2 の累乗でなければならない。

表 7-8 は、`Pack`、`Perm`、`CCS` の各パックド形式で、出力結果がどのように表現されるかを示している。例 7-5 は、指定を初期化し、`ippsFFTFwd_RToCCS_32f` を呼び出す例を示している。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。データ配列へのポインタが NULL。
`IppStsContextMatchErr`	エラー。指定識別子 *pFFTSpec* が正しくない。
`ippStsMemAllocErr`	エラー。メモリが割り当てられていない。

表 7-8　Pack、Perm、CCS 形式での順方向 FFT の結果の表示

FFTReal	**0**	**1**	**2**	**3**	**. . .**	**N-2**	**N-1**	**N**	**N+1**
`Pack`	R_0	R_1	I_1	R_2	. . .	$I_{N/2-1}$	$R_{N/2}$		
`Perm`	R_0	$R_{N/2}$	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$		
`CCS`	R_0	0	R_1	I_1	. . .	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

例 7-5　ippsFFTFwd_RToCCS 関数の使用例

```
IppStatus fft( void ) {
     Ipp32f x[8], X[10];
     int n;
     IppStatus status;
     IppsFFTSpec_R_32f* spec;
     status = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
              ippAlgHintNone );
     for(n=0; n<8; ++n) x[n] = (float)cos(IPP_2PI *n *16/64);
     status = ippsFFTFwd_RToCCS_32f( x, X, spec, NULL );
     ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
     ippsFFTFree_R_32f( spec );
     printf_32f("fft magn =", x, 4, status );
     return status;
}
Output:
         fft magn = 0.000000 0.000000 4.000000 0.000000
Matlab* Analog:
         >> n=0:7; x=sin(2*pi*n*16/64); X=abs(fft(x)); X(1:4)
```

離散フーリエ変換関数

この項では、実数信号および複素数信号に対して順方向と逆方向の離散フーリエ変換（DFT）を計算する関数について説明する。DFT は高速フーリエ変換と比べて効率が劣るが、DFT で変換されるベクトルの長さは任意に設定できる。

初期化関数に渡される *hint* 引数は、特別なアルゴリズム（より高速なアルゴリズムまたはより精度の高いアルゴリズム）の使用を指定する。*flag* 引数は、結果を正規化するメソッドを指定する。複素数信号は、複素数要素を含む単一の配列、または実数部と虚数部を別々に含む2つの配列として表現できる。FFT の出力結果は、Perm、Pack、または CCS 形式でパックできる。

DFT 関数を使用するには、回転因子のテーブルのようなデータを含む指定構造体を初期化する必要がある。初期化関数では、順方向と逆方向の変換について、それぞれ指定が生成される。

離散フーリエ変換の高速計算については、[Mit93]、8-2 項、「DFT の高速計算」を参照のこと。

インテル® IPP 関数の特殊なセットは、複素数信号の 「アウト・オブ・オーダー (Out-Of-Order)」DFT を提供する。この場合、順方向変換と逆方向変換で使用する周波数領域内の要素を順序変更して、変換の計算速度を上げることができる。この順序変更は、ユーザには表示されず、異なる関数の実装では動作が異なる場合がある。ただし、順方向変換と逆方向変換の各関数ペアの可逆性は保証される。

DFTInitAlloc_C, DFTInitAlloc_R

実数信号および複素数信号に対する 離散フーリエ変換の構造体を初期化する。

```
IppStatus ippsDFTInitAlloc_R_16s(IppsDFTSpec_R_16s** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_16s(IppsDFTSpec_C_16s** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_16sc(IppsDFTSpec_C_16sc** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_R_32f(IppsDFTSpec_R_32f** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_32f(IppsDFTSpec_C_32f** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_32fc(IppsDFTSpec_C_32fc** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_R_64f(IppsDFTSpec_R_64f** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_64f(IppsDFTSpec_C_64f** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTInitAlloc_C_64fc(IppsDFTSpec_C_64fc** pDFTSpec,
    int length, int flag, IppHintAlgorithm hint);
```

引数

flag	結果を正規化するメソッドを指定する。*flag* 引数の値は、「flag 引数と hint 引数」を参照。
hint	特別なコードを使用して計算することを指定する。*hint* 引数の値は、「flag 引数と hint 引数」を参照。
length	DFT 変換の長さ。

pDFTSpec	生成される DFT 指定構造体へのポインタ。

説明

関数 ippsDFTInitAlloc_C と ippsDFTInitAlloc_R は、ipps.h ファイルで宣言される。これらの関数は、変換パラメータ *length*、正規化パラメータ *flag*、特別なコードを指定するパラメータ *hint* を使用して、DFT 指定構造体 *pDFTSpec* を生成し、初期化する。*length* 引数は、変換の長さを指定する。

ippsDFTInitAlloc_C。関数 ippsDFTInitAlloc_C は、複素数型 DFT の指定構造体を初期化する。

ippsDFTInitAlloc_R。関数 ippsDFTInitAlloc_R は、実数型 DFT の指定構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDFTSpec* ポインタが NULL。
ippStsSizeErr	エラー。*length* がゼロ以下。
ippStsFftFlagErr	エラー。*flag* の値が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

DFTFree_C, DFTFree_R

実数信号および複素数信号に対する
離散フーリエ変換の構造体をクローズする。

```
IppStatus ippsDFTFree_R_16s(IppsDFTSpec_R_16s* pDFTSpec);
IppStatus ippsDFTFree_C_16s(IppsDFTSpec_C_16s* pDFTSpec);
IppStatus ippsDFTFree_C_16sc(IppsDFTSpec_C_16sc* pDFTSpec);

IppStatus ippsDFTFree_R_32f(IppsDFTSpec_R_32f* pDFTSpec);
IppStatus ippsDFTFree_C_32f(IppsDFTSpec_C_32f* pDFTSpec);
IppStatus ippsDFTFree_C_32fc(IppsDFTSpec_C_32fc* pDFTSpec);

IppStatus ippsDFTFree_R_64f(IppsDFTSpec_R_64f* pDFTSpec);
IppStatus ippsDFTFree_C_64f(IppsDFTSpec_C_64f* pDFTSpec);
IppStatus ippsDFTFree_C_64fc(IppsDFTSpec_C_64fc* pDFTSpec);
```

引数

pDFTSpec	クローズする DFT 指定構造体へのポインタ。

説明

関数 ippsDFTFree は、ipps.h ファイルで宣言される。この関数は、ippsDFTInitAlloc_C または ippsDFTInitAlloc_R により生成された指定に関連付けられているメモリをすべて解放すると、DFT 指定構造体 *pDFTSpec* をクローズする。変換が完了したら、ippsDFTFree を呼び出す。

ippsDFTFree_C。関数 ippsDFTFree_C は、複素数型の DFT 指定構造体をクローズする。

ippsDFTFree_R。関数 ippsDFTFree_R は、実数型の DFT 指定構造体をクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDFTSpec* ポインタが NULL。
ippStsContextMatchErr	エラー。指定識別子 *pDFTSpec* が正しくない。

DFTGetBufSize_C, DFTGetBufSize_R

DFT 作業バッファのサイズ（バイト）を取得する。

```
IppStatus ippsDFTGetBufSize_R_16s(const IppsDFTSpec_R_16s* pDFTSpec,
    int* pSize);
IppStatus ippsDFTGetBufSize_C_16s(const IppsDFTSpec_C_16s* pDFTSpec,
    int* pSize);
IppStatus ippsDFTGetBufSize_C_16sc(const IppsDFTSpec_C_16sc* pDFTSpec,
    int* pSize);

IppStatus ippsDFTGetBufSize_R_32f(const IppsDFTSpec_R_32f* pDFTSpec,
    int* pSize);
IppStatus ippsDFTGetBufSize_C_32f(const IppsDFTSpec_C_32f* pDFTSpec,
    int* pSize);
IppStatus ippsDFTGetBufSize_C_32fc(const IppsDFTSpec_C_32fc* pDFTSpec,
    int* pSize);
```


intel®

```
IppStatus ippsDFTGetBufSize_R_64f(const IppsDFTSpec_R_64f* pDFTSpec,
    int* pSize);
IppStatus ippsDFTGetBufSize_C_64f(const IppsDFTSpec_C_64f* pDFTSpec,
    int* pSize);
IppStatus ippsDFTGetBufSize_C_64fc(const IppsDFTSpec_C_64fc* pDFTSpec,
    int* pSize);
```

引数

pDFTSpec	DFT 指定構造体へのポインタ。
pSize	DFT 作業バッファ・サイズ値へのポインタ。

説明

関数 ippsDFTGetBufSize は、ipps.h ファイルで宣言される。この関数は、指定構造体 *pDFTSpec* により記述されている DFT の外部メモリ・バッファのサイズ（バイト単位）を計算し、それを *pSize* に格納する。

pDCTSpec を初期化してから、この関数を呼び出す。

ippsDFTGetBufSize_C。関数 ippsDFTGetBufSize_C は、複素数型 DFT の作業バッファ・サイズを取得する。

ippsDFTGetBufSize_R。関数 ippsDFTGetBufSize_R は、実数型 DFT の作業バッファ・サイズを取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDFTSpec* が NULL。
ippStsContextMatchErr	エラー。指定識別子 *pDFTSpec* が正しくない。

DFTFwd_CToC, DFTInv_CToC

複素数信号に対する順方向または逆方向の
離散フーリエ変換を計算する。

```
IppStatus ippsDFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f*
    pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f*
    pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f*
    pSrcIm, Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f*
    pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f*
    pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f*
    pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f*
    pSrcIm, Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f*
    pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    const IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    const IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    const IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    const IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s*
    pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s*
    pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s*
    pSrcIm, Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s*
    pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    const IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsDFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
    const IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
```

引数

pDFTSpec	DFT 指定構造体へのポインタ。
pSrc	複素数値が格納された入力配列へのポインタ。
pDst	複素数値が格納された出力配列へのポインタ。
pSrcRe	信号の実数部が格納された入力配列へのポインタ。
pSrcIm	信号の虚数部が格納された入力配列へのポインタ。
pDstRe	信号の実数部が格納された出力配列へのポインタ。
pDstIm	信号の虚数部が格納された出力配列へのポインタ。
pBuffer	作業バッファへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

intel®

説明

関数 ippsDFTFwd_CToC と ippsDFTInv_CToC は、ipps.h ファイルで宣言される。これらの関数は、*pDFTSpec* 指定パラメータ（変換パラメータ *length*、正規化パラメータ *flag*、特別なコードを指定するパラメータ *hint*）に従って、複素数信号に対する順方向または逆方向の DFT を計算する。

複素数データ型を使用する関数（32fc サフィックスが付いた関数など）は、入力複素数配列 *pSrc* を処理し、その結果を *pDst* に格納する。

実数データ型を使用し、独立した実数部 *pSrcRe* と虚数部 *pSrcIm* で表現される複素数信号を処理する関数（32f サフィックスが付いた関数など）は、結果をそれぞれ *pDstRe* と *pDstIm* に格納する。

整数データ型の場合、出力結果は *scaleFactor* 値に従ってスケーリングされ、出力信号の範囲と精度を保持する。*pBuffer* 引数は、必要な作業メモリを DFT 関数に提供し、関数内のメモリ割り当てが行われないようにする。バッファによりパフォーマンスが向上するのは、キャッシュに保存された前の演算結果を DFT 関数が入力配列として使用する場合である。2 つの連続する演算で同じバッファが使用される。DFT 関数を呼び出す前に、必要なバッファ・サイズを対応する関数 ippsDFTGetBufSize_C で計算する必要がある。NULL ポインタが渡された場合は、内部で DFT 関数の メモリが割り当てられる。

順方向の DFT 関数は、次の式で表す。

$$X(k) = A\sum_{n=0}^{N-1} x(n)\cdot exp\left(-j2\pi\frac{kn}{N}\right)$$

k は周波数領域の要素のインデックス、*n* は時間領域の要素のインデックス、*N* は入力信号 *length*、*A* と *B* は *flag* で指定される乗数を表す。また、順方向の場合、*x(n)* は *pSrc[n]*、*X(k)* は *pDst[k]* となり、逆方向の場合、*x(n)* は *pSrc[n]*、*X(k)* は *pSrc[k]* となる。

逆方向の離散フーリエ変換の定義は、次の式で表す。

$$x(k) = B\sum_{n=0}^{N-1} X(n)\cdot exp\left(j2\pi\frac{kn}{N}\right)$$

ippsDFTFwd_CToC。関数 ippsDFTFwd_CToC は、複素数型の順方向 DFT を計算する。

ippsDFTInv_CToC。関数 ippsDFTInv_CToC は、複素数型の逆方向 DFT を計算する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。データ配列へのポインタが NULL。
ippStsContextMatchErr	エラー。指定識別子 *pDFTSpec* が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

DFTFwd_RToPerm, DFTInv_PermToR, DFTFwd_RToPack, DFTInv_PackToR, DFTFwd_RToCCS, DFTInv_CCS ToR

実数信号に対して順方向または逆方向の
離散フーリエ変換を計算する。

```
IppStatus ippsDFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsDFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsDFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsDFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsDFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
IppStatus ippsDFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    const IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u*
    pBuffer);
```

引数

pDFTSpec	DFT 指定構造体へのポインタ。
pSrc	順方向変換で生成した複素数値と逆方向変換で生成したパックド複素数値が格納された入力配列へのポインタ。
pDst	逆方向変換で生成した実数値と順方向変換で生成したパックド複素数値が格納された出力配列へのポインタ。
pBuffer	作業バッファへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

これらの関数は、ipps.h ファイルで宣言される。これらの関数は、*pDFTSpec* 指定パラメータ（変換パラメータ *length*、正規化パラメータ *flag*、特別なコードを指定するパラメータ *hint*）に従って、実数信号に対する順方向または逆方向の DFT を計算する。

実数信号に対する順方向変換の結果（すなわち、周波数領域内の結果）は、Pack、Perm、または CCS のパックド形式で表現される。データはパック可能だが、これは、実数信号に対する DFT 変換が対称特性を持つためである。

整数データ型の場合、出力結果は *scaleFactor* 値に従ってスケーリングされ、出力信号の範囲と精度を保持する。*pBuffer* 引数は、必要な作業メモリを DFT 関数に提供し、関数内のメモリ割り当てが行われないようにする。バッファによりパフォーマンスが向上するのは、キャッシュに保存された前の演算結果を DFT 関数が入力配列として使用する場合である。2 つの連続する演算で同じバッファが使用される。必要なバッファ・サイズを対応する関数 ippsDFTGetBufSize_R で計算する必要がある。NULL ポインタが渡された場合は、内部で DFT 関数の メモリが割り当てられる。

順方向の DFT 関数は、次の式で表される。

$$X(k) = A\sum_{n=0}^{N-1} x(n)\cdot exp\left(-j2\pi\frac{kn}{N}\right)$$

k は周波数領域の要素のインデックス、*n* は時間領域の要素のインデックス、*N* は入力信号 *length*、*A* と *B* は *flag* で指定される乗数を表す。また、順方向の場合、*x(n)* は *pSrc[n]*、*X(k)* は *pDst[k]* となり、逆方向の場合、*x(n)* は *pDst[n]*、*X(k)* は *pSrc[k]* となる。

逆方向の離散フーリエ変換の定義は、次の式で表される。

$$x(k) = B\sum_{n=0}^{N-1} X(n)\cdot exp\left(j2\pi\frac{kn}{N}\right)$$

ippsDFTFwd_RToPerm, **ippsDFTInv_PermToR**。これらの関数は、順方向または逆方向の DFT を計算し、その結果を Perm 形式で格納する。

ippsDFTFwd_RToPack, **ippsDFTInv_PackToR**。これらの関数は、順方向または逆方向の DFT を計算し、その結果を Pack 形式で格納する。

ippsDFTFwd_RToCCS , **ippsDFTInv_CCSToR**。これらの関数は、順方向または逆方向の DFT を計算し、その結果を CCS 形式で格納する。

例 7-6 は、指定を初期化し、ippsDFTFwd_RToCCS_32f を呼び出す例を示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。データ配列へのポインタが NULL。

ippStsContextMatchErr	エラー。指定識別子 *pDFTSpec* が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

例 7-6 ippsDFTFwd_RToCCS 関数の使用例

```
IppStatus dft( void ) {
      Ipp32f x[7], X[8];
      int n;
      IppStatus status;
      IppsDFTSpec_R_32f* spec;
      status = ippsDFTInitAlloc_R_32f(&spec, 7, IPP_FFT_DIV_INV_BY_N,
                 ippAlgHintNone);
      for( n=0; n<7; ++n ) x[n] = (float) cos(IPP_2PI * n * 14 / 49);
      status = ippsDFTFwd_RToCCS_32f( x, X, spec, NULL );
      ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
      ippsDFTFree_R_32f( spec );
      printf_32f("dft magn =", x, 4, status );
      return status;
}
Output:
         dft magn =  0.000000 0.000000 3.500000 0.000000
Matlab* analog:
         >> N=7;F=14/49;n=0:N-1;x=cos(2*pi*n*F);y=abs(fft(x));y(1:4)
```

DFTOutOrdInitAlloc_C

アウト・オブ・オーダー離散フーリエ変換
の構造体を初期化する。

```
IppStatus ippsDFTOutOrdInitAlloc_C_32fc(IppsDFTOutOrdSpec_C_32fc**
   pDFTSpec, int length, int flag, IppHintAlgorithm hint);
IppStatus ippsDFTOutOrdInitAlloc_C_64fc(IppsDFTOutOrdSpec_C_64fc**
   pDFTSpec, int length, int flag, IppHintAlgorithm hint);
```

引数

pDFTSpec	生成される DFT 指定構造体へのポインタ。
flag	結果を正規化するメソッドを指定する。*flag* 引数の値は、「flag 引数と hint 引数」を参照のこと。

hint	特別なコードを使用して計算することを指定する。引数の値は、「flag 引数と hint 引数」を参照のこと。
length	DFT 変換の長さ。

説明

関数 ippsDFTOutOrdInitAlloc_C は、ipps.h ファイルで宣言される。この関数は、変換パラメータ *length*、正規化パラメータ *flag*、特別なコードを指定するパラメータ *hint* を使用して、アウト・オブ・オーダー DFT の指定構造体 *pDFTSpec* を生成し、初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pFFTSpec* ポインタが NULL。
ippStsSizeErr	エラー。*length* がゼロ以下。
ippStsFftFlagErr	エラー。*flag* の値が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

DFTOutOrdFree_C

アウト・オブ・オーダー離散フーリエ変換の構造体をクローズする。

```
IppStatus ippsDFTOutOrdFree_C_32fc(IppsDFTOutOrdSpec_C_32fc* pDFTSpec);
IppStatus ippsDFTOutOrdFree_C_64fc(IppsDFTOutOrdSpec_C_64fc* pDFTSpec);
```

引数

pDFTSpec	クローズする DFT 指定構造体へのポインタ。

説明

関数 ippsDFTOutOrdFree は、ipps.h ファイルで宣言される。この関数は、ippsDFTOutOrdInitAlloc_C で生成された指定構造体に関連付けられているメモリをすべて解放することにより、アウト・オブ・オーダー DFT の指定構造体 *pDFTSpec* をクローズする。変換が完了したら、ippsDFTOutOrdFree を呼び出す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDFTSpec* ポインタが NULL。
ippStsContextMatchErr	エラー。指定識別子 *pDFTSpec* が正しくない。

DFTOutOrdGetBufSize_C

アウト・オブ・オーダー DFT の作業バッファ
のサイズを計算する。

```
IppStatus ippsDFTOutOrdGetBufSize_C_32fc(const
    IppsDFTOutOrdSpec_C_32fc* pDFTSpec, int* pSize);
IppStatus ippsDFTOutOrdGetBufSize_C_64fc(const
    IppsDFTOutOrdSpec_C_64fc* pDFTSpec, int* pSize);
```

引数

pDFTSpec	DFT 指定構造体へのポインタ。
pSize	DFT 作業バッファ・サイズ値へのポインタ。

説明

関数 ippsDFTOutOrdGetBufSize は、ipps.h ファイルで宣言される。この関数は、指定構造体 *pDFTSpec*, により記述されているアウト・オブ・オーダー DFT の外部メモリ・バッファのサイズ（バイト）を計算し、それを *pSize* に格納する。

外部バッファを使用する場合は、*pDFTSpec* を初期化してから、この関数を呼び出す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDFTSpec* が NULL。
ippStsContextMatchErr	エラー。指定識別子 *pDFTSpec* が正しくない。

DFTOutOrdFwd_CToC, DFTOutOrdInv_CToC

順方向または逆方向のアウト・オブ・
オーダー離散フーリエ変換を計算する。

```
IppStatus ippsDFTOutOrdFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    const IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    const IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
    const IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
IppStatus ippsDFTOutOrdInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
    const IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

引数

pSrc	ソース・データへのポインタ。
pDst	出力データへのポインタ。
pDFTSpec	DFT 指定構造体へのポインタ。
pBuffer	作業バッファへのポインタ。

説明

関数 ippsDFTOutOrdFwd_CToC と ippsDFTOutOrdInv_CToC は、ipps.h ファイルで宣言される。これらの関数は、*pDFTSpec* 指定パラメータ（変換パラメータ *length*、正規化パラメータ *flag*、特別なコードを指定するパラメータ *hint*）に従って、複素数信号 *pSrc* に対する順方向または逆方向のアウト・オブ・オーダー DFT を計算し、その結果を *pDst* に格納する。

これらの関数は、関数 ippsDFTFwd_CToC と ippsDFTInv_CToC の機能に似ているが、周波数領域内の要素（順方向変換は *pDst*、逆方向変換は *pSrc*）を順序変更して、計算速度を上げることが可能である点が異なる。この動作は、ユーザには表示されず、関数の実装によって異なる。ただし、順方向変換と逆方向変換の各関数ペアの可逆性は保証される。

pBuffer 引数は、必要な作業メモリを DFT 関数に提供し、関数内のメモリ割り当てが行われないようにする。バッファによりパフォーマンスが向上するのは、キャッシュに保存された前の演算結果を DFT 関数が入力配列として使用する場合である。2 つの連続する演算で同じバッファが使用される。必要なバッファのサイズは、DFT 計算関数を使用する前に、関数 ippsDFTOutOrdGetBufSize_C で計算する必要がある。NULL ポインタが渡されると、メモリは DFT 計算関数によって内部的に割り当てられる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pDFTSpec* が NULL。
ippStsContextMatchErr	エラー。指定識別子 *pDFTSpec* が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

特定周波数用 DFT（Goertzel）関数

この項では、特定の周波数に対する離散フーリエ変換を 1 つ、または複数個計算する関数について説明する。DFT が存在するのは、正規化された周波数、つまり 0、1/*N*、2/*N*、...（*N*-1）/*N* に対してだけであるのに注意する。N は、時間領域サンプルの数である。したがって、周波数値は上記のセットの中から選択する必要がある。

これらのインテル® IPP 関数は、Goertzel アルゴリズム [Mit98] を使用し、少数の DFT 値が必要な場合の効率を改善する。

関数によっては、1 つではなく、2 つの値を計算するものもある。こうした関数を使用すると、複数の値を計算するアプリケーション（デュアルトーン・マルチ周波数信号の検出など）は、より高速に処理を実行できる。SIMD 命令を備えたインテルのプロセッサ上では、特に処理が高速化される。

Goertz

単一の複素数信号の特定の周波数に対する
離散フーリエ変換を計算する。

```
IppStatus ippsGoertz_32f(const Ipp32f* pSrc, int len, Ipp32fc* pVal,
    Ipp32f freq);
IppStatus ippsGoertz_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pVal,
    Ipp32f freq);
IppStatus ippsGoertz_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pVal,
    Ipp64f freq);
IppStatus ippsGoertz_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16sc*
    pVal, Ipp32f freq, int scaleFactor))
IppStatus ippsGoertz_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc*
    pVal, Ipp32f freq, int scaleFactor);
```

引数

freq	単一の相対周波数値 [0, 1.0）へのポインタ。
pSrc	入力複素数データ・ベクトルへのポインタ。
len	ベクトル内の要素の数。
pVal	出力 DFT 値へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsGoertz は、ipps.h ファイルで宣言される。この関数は、特定の周波数 *freq* の長さ *len* の複素数入力信号 *pSrc* に対する DFT を計算し、その結果を *pVal* に格納する。

ippsGoertz 関数は、次の式で表される。

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot exp\left(-j2\pi\frac{kn}{N}\right) \quad ,$$

k/N は、DFT を計算する正規化された *freq* 値を表す。

例 7-7 は、DFT を計算する際に、Goertzel 関数を使用して特定の周波数の大きさを選択する例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。データ配列へのポインタが NULL。
ippStsRelFreqErr	エラー。*freq* が範囲外。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 7-7　　Goertzel 関数による特定の周波数の大きさの選択

```
IppStatus goertzel( void ) {
#undef  LEN
#define LEN 100
     IppStatus status;
     Ipp32fc *x = ippsMalloc_32fc( LEN ), y;
     int n;
     ///generate a signal of 60 Hz freq that
     /// is sampled with 400 Hz freq
     for( n=0; n<LEN; ++n) {
        x[n].re =(Ipp32f)sin(IPP_2PI * n * 60 / 400);
        x[n].im = 0;
     }
     status = ippsGoertz_32fc( x, LEN, &y, 60.0f / 400 );
     printf_32fc("goertz =", &y, 1, status );
     ippsFree( x );
     return status;
}
Output:
         goertz =  {0.000090,-50.000008}
Matlab* Analog
    >> N=100;F=60/400;n=0:N-1;x=sin(2*pi*n*F);y=fft(x);n=N*F;y(n+1)
```

GoertzTwo

単一の複素数信号の特定周波数に対して
2 つの離散フーリエ変換を計算する。

```
IppStatus ippsGoertzTwo_32fc(const Ipp32fc* pSrc, int len,
    Ipp32fc pVal[2], const Ipp32f freq[2]);
IppStatus ippsGoertzTwo_64fc(const Ipp64fc* pSrc, int len,
    Ipp64fc pVal[2], const Ipp64f freq[2]);
IppStatus ippsGoertzTwo_16sc_Sfs(const Ipp16sc* pSrc, int len,
    Ipp16sc pVal[2], const Ipp32f freq[2], int scaleFactor);
```

引数

freq	2 つの単一相対周波数値 [0, 1.0) へのポインタ。
pSrc	入力複素数データ・ベクトルへのポインタ。
len	ベクトル内の要素の数。
pVal	出力 DFT 値へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 `ippsGoertzTwo` は、`ipps.h` ファイルで宣言される。この関数は、2 つの特定周波数 *freq* の長さ *len* の複素数入力信号 *pSrc* に対する DFT を計算し、その結果を *pVal* に格納する。インテル® Pentium® III プロセッサ上では、1 つの DFT の計算に要する時間と 2 つの DFT の計算に要する時間が同じである。したがって、複数の DFT を計算するアプリケーションは高速に実行される。

`ippsGoertz` 関数は、次の式で表される。

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot exp\left(-j2\pi\frac{kn}{N}\right)$$

k/N は、DFT を計算する正規化された *freq* 値の 1 つを表す。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。データ配列へのポインタが `NULL`。
`ippStsRelFreqErr`	エラー。*freq* が範囲外。
`ippStsSizeErr`	エラー。*len* がゼロ以下。

離散コサイン変換関数

この項では、信号に対して離散コサイン変換を計算する関数について説明する。インテル® IPP 信号処理データ領域で使用される DCT 関数は、[Rao90] で提案されている修正された計算アルゴリズムを実行する。

DCTFwdInitAlloc, DCTInvInitAlloc

離散コサイン変換の構造体を初期化する。

```
IppStatus ippsDCTFwdInitAlloc_16s(IppsDCTFwdSpec_16s** pDCTSpec,
    int length, IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_16s(IppsDCTInvSpec_16s** pDCTSpec,
    int length, IppHintAlgorithm hint);

IppStatus ippsDCTFwdInitAlloc_32f(IppsDCTFwdSpec_32f** pDCTSpec,
    int length, IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_32f(IppsDCTInvSpec_32f** pDCTSpec,
    int length, IppHintAlgorithm hint);

IppStatus ippsDCTFwdInitAlloc_64f(IppsDCTFwdSpec_64f** pDCTSpec,
    int length, IppHintAlgorithm hint);
IppStatus ippsDCTInvInitAlloc_64f(IppsDCTInvSpec_64f** pDCTSpec,
    int length, IppHintAlgorithm hint);
```

引数

flag	結果を正規化するメソッドを指定する。*flag* 引数の値は、「flag 引数と hint 引数」を参照。
hint	特別なコードを使用して計算することを指定する。*hint* 引数の値は、「flag 引数と hint 引数」を参照。
length	DCT 内のサンプルの数。
pDCTSpec	生成される DCT 指定構造体へのポインタ。

説明

関数 ippsDCTFwdInitAlloc と ippsDCTInvInitAlloc は、ipps.h ファイルで宣言される。これらの関数は、変換パラメータ *length*、正規化パラメータ *flag*、特別なコードを指定するパラメータ *hint* を使用して、DCT 指定構造体 *pDCTSpec* を生成し、初期化する。*Length* 引数は、変換の長さを指定する。

ippsDCTFwdInitAlloc。関数 ippsDCTFwdInitAlloc は、順方向 DCT の指定構造体を初期化する。

ippsDCTInvInitAlloc。関数 ippsDCTInvInitAlloc は、逆方向 DCT の指定構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDCTSpec* ポインタが NULL。
ippStsSizeErr	エラー。*length* がゼロ以下。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

DCTFwdFree, DCTInvFree

離散コサイン変換の構造体をクローズする。

```
IppStatus ippsDCTFwdFree_16s(IppsDCTFwdSpec_16s* pDCTSpec);
IppStatus ippsDCTInvFree_16s(IppsDCTInvSpec_16s* pDCTSpec);
IppStatus ippsDCTFwdFree_32f(IppsDCTFwdSpec_32f* pDCTSpec);
IppStatus ippsDCTInvFree_32f(IppsDCTInvSpec_32f* pDCTSpec);
IppStatus ippsDCTFwdFree_64f(IppsDCTFwdSpec_64f* pDCTSpec);
IppStatus ippsDCTInvFree_64f(IppsDCTInvSpec_64f* pDCTSpec);
```

引数

pDCTSpec	クローズする DCT 指定構造体へのポインタ。

説明

関数 ippsFFTFwdFree と ippsFFTInvFree は、ipps.h ファイルで宣言される。これらの関数は、ippsDCTFwdInitAlloc または ippsDCTInvInitAlloc により生成された指定に関連付けられているメモリをすべて解放すると、DCT 指定構造体 *pDCTSpec* をクローズする。ippsDCTFwdFree または ippsDCTInvFree は、変換の完了後に呼び出す必要がある。

ippsDCTFwdFree。関数 ippsDCTFwdFree は、順方向 DCT の指定構造体をクローズする。

ippsDCTInvFree。関数 ippsDCTInvFree は、逆方向 DCT の指定構造体をクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDCTSpec* ポインタが NULL。

ippStsContextMatchErr	エラー。指定識別子 *pDCTSpec* が正しくない。

DCTFwdGetBufSize, DCTInvGetBufSize

DCT 作業バッファのサイズ（バイト）を取得する。

```
IppStatus ippsDCTFwdGetBufSize_16s(const IppsDCTFwdSpec_16s* pDCTSpec,
    int* pSize);
IppStatus ippsDCTInvGetBufSize_16s(const IppsDCTInvSpec_16s* pDCTSpec,
    int* pSize);

IppStatus ippsDCTFwdGetBufSize_32f(const IppsDCTFwdSpec_32f* pDCTSpec,
    int* pSize);
IppStatus ippsDCTInvGetBufSize_32f(const IppsDCTInvSpec_32f* pDCTSpec,
    int* pSize);

IppStatus ippsDCTFwdGetBufSize_64f(const IppsDCTFwdSpec_64f* pDCTSpec,
    int* pSize);
IppStatus ippsDCTInvGetBufSize_64f(const IppsDCTInvSpec_64f* pDCTSpec,
    int* pSize);
```

引数

pDCTSpec	DCT 指定構造体へのポインタ。
pSize	DCT 作業バッファ・サイズ値へのポインタ。

説明

関数 ippsDCTFwdGetBufSize と ippsDCTInvGetBufSize は、ipps.h ファイルで宣言される。これらの関数は、指定構造体 *pDCTSpec* により記述されている DCT 作業バッファ・サイズ（バイト）を取得し、それを *pSize* に格納する。

ippsDCTFwdGetBufSize。関数 ippsDCTFwdGetBufSize は、順方向 DCT の作業バッファ・サイズを取得する。

ippsDCTInvGetBufSize。関数 ippsDCTInvGetBufSize は、逆方向 DCT の作業バッファ・サイズを取得する。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。*pDCTSpec* ポインタが NULL。
ippStsContextMatchErr	エラー。指定識別子 *pDCTSpec* が正しくない。

DCTFwd, DCTInv

信号に対する順方向または逆方向の
離散コサイン変換を計算する。

```
IppStatus ippsDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsDCTFwdSpec_32f* pDCTSpec, Ipp8u* pBuffer);
IppStatus ippsDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const IppsDCTInvSpec_32f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippsDCTFwd_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsDCTFwdSpec_64f* pDCTSpec, Ipp8u* pBuffer);
IppStatus ippsDCTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst,
    const IppsDCTInvSpec_64f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippsDCTFwd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDCTFwdSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
IppStatus ippsDCTInv_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDCTInvSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
```

引数

pDCTSpec	DCT 指定構造体へのポインタ。
pSrc	入力データ配列へのポインタ。
pDst	出力データ配列へのポインタ。
pBuffer	DCT 作業バッファへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDCTFwd と ippsDCTInv は、ipps.h ファイルで宣言される。これらの関数は、順方向と逆方向の離散コサイン変換（DCT）を計算する。*length* が 2 の累乗の場合は、DCT を直接計算するより大幅に速い効率的なアルゴリズムが使用される。*length* がその他の値の場合は、次に示す式を直接計算する。ただし、コサイン関数の対称性が考慮されるため、式内の乗算演算のおよそ半分を実行するだけで済む。

次に示す DCT の定義で、*N=length*、

$C(k) = \frac{1}{\sqrt{N}}$（$k$=0 の場合）、$C(k) = \frac{\sqrt{2}}{\sqrt{N}}$（$k$>0 の場合）

順方向の DCT では、$x(n)$ は *pSrc[n]*、$y(k)$ は *pDst[k]*、

逆方向の DCT では、$x(n)$ は *pDst[n]*、$y(k)$ は *pSrc[k]* になる。

順方向の DCT は、次の式で定義される。

$$y(k) = C(k)\sum_{n=0}^{N-1} x(n)\cdot\cos\frac{(2n+1)\pi k}{2N}$$

逆方向の離散コサイン変換の定義は、次のとおりである。

$$x(n) = \sum_{k=0}^{N-1} C(k)y(k)\cdot\cos\frac{(2n+1)\pi k}{2N}$$

整数データ型の場合、出力結果は *scaleFactor* 値に従ってスケーリングされ、出力信号の範囲と精度を保持する。*pBuffer* 引数は、必要な作業メモリを DCT 関数に提供し、関数内のメモリ割り当てが行われないようにする。バッファによりパフォーマンスが向上するのは、キャッシュに保存された前の演算結果を DCT 関数が入力配列として使用する場合である。2 つの連続する演算で同じバッファが使用される。

ippsDCTFwd。関数 ippsDCTFwd は、順方向の DCT を計算する。

ippsDCTInv。関数 ippsDCTInv は、逆方向の DCT を計算する。

例 7-8 は、関数 ippsDCTFwd_32f と ippsDCTInv_32f を使用し、信号の圧縮と展開を行う方法を示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pDCTSpec* が NULL。
ippStsContextMatchErr	エラー。指定識別子 *pDCTSpec* が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

例 7-8　ippsDCTFwd と ippsDCTInv による信号の圧縮と展開

```
void dct( void ) {
#define LEN 256
      Ipp32f x[LEN], y[LEN];
      int n;
      IppsDCTFwdSpec_32f* fspec;
      IppsDCTInvSpec_32f* ispec;
      IppStatus status;
      /// data: Gaussian function, magn =1 and sigma=N/3
      for(n=0; n<LEN; ++n)
          x[n] = (float)(exp(-0.5*(LEN/2-n)*(LEN/2-n)/(LEN/3.0)));
      /// get cosine transform coefficients
      status = ippsDCTFwdInitAlloc_32f( &fspec, LEN, ippAlgHintNone );
      status = ippsDCTFwd_32f( x, y, fspec, NULL );
      ippsDCTFwdFree_32f( fspec );
      /// Set 3/4 of these coefficients to zero
      for(n=LEN/4; n<LEN; ++n) y[n] = 0.0f;
      /// restore signal using len/4 values
      status = ippsDCTInvInitAlloc_32f( &ispec, LEN, ippAlgHintNone );
      status = ippsDCTInv_32f( y, x, ispec, NULL );
      ippsDCTInvFree_32f( ispec );
}
```

ヒルベルト変換関数

この項で説明する関数は、ヒルベルト変換を使用して、実数データ・シーケンスから離散時間分析信号を計算する。この分析信号は、元のデータの複製を実数部に格納し、ヒルベルト変換を虚数部に格納する複素数信号である。つまり、虚数部は、元の実数データの位相を 90 度シフトしたデータになる。ヒルベルト変換データの内容には、元の実数データと同じ大きさと周波数以外に、追加の位相情報が含まれる。

HilbertInitAlloc

ヒルベルト変換の構造体を初期化する。

```
IppStatus ippsHilbertInitAlloc_32f32fc(IppsHilbertSpec_32f32fc**
    pSpec, int length, IppHintAlgorithm hint);
IppStatus ippsHilbertInitAlloc_16s32fc(IppsHilbertSpec_16s32fc**
    pSpec, int length, IppHintAlgorithm hint);
IppStatus ippsHilbertInitAlloc_16s16sc(IppsHilbertSpec_16s16sc**
    pSpec, int length, IppHintAlgorithm hint);
```

引数

pSpec	初期化されるヒルベルト・コンテキスト構造体へのポインタ。
length	ヒルベルト変換のサンプル数。
hint	計算に特別なコードを使用するように指定する。*hint* 引数の値は、「flag 引数と hint 引数」に記載されている。

説明

関数 `ippsHilbertInitAlloc` は、`ipps.h` ファイルで宣言される。この関数は、指定されたパラメータ（変換の長さを指定する *length* と、特別なコードのヒント *hint*）を使用して、ヒルベルト指定構造体 *pSpec* を作成し、初期化する。ヒルベルト変換関数 `ippsHilbert` を実際に使用する前に、この関数を呼び出す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSpec* ポインタが NULL。
ippStsSizeErr	エラー。*length* がゼロ以下。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

HilbertFree

ヒルベルト変換の構造体をクローズする。

```
IppStatus ippsHilbertFree_32f32fc(IppsHilbertSpec_32f32fc* pSpec);
```

```
IppStatus ippsHilbertFree_16s32fc(IppsHilbertSpec_16s32fc* pSpec);
IppStatus ippsHilbertFree_16s16sc(IppsHilbertSpec_16s16sc* pSpec);
```

引数

pSpec	クローズされるヒルベルト指定構造体へのポインタ。

説明

関数 ippsHilbertFree は、ipps.h ファイルで宣言される。この関数は、ippsHilbertInitAlloc 関数によって作成されたヒルベルト指定に割り当てられたすべてのメモリを解放して、ヒルベルト指定構造体 *pSpec* をクローズする。変換が完了したら、ippsHilbertFree を呼び出す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSpec* ポインタが NULL。
ippStsContextMatchErr	エラー。指定識別子 *pSpec* が正しくない。

Hilbert

ヒルベルト変換を使用して分析信号を計算する。

```
IppStatus ippsHilbert_32f32fc(const Ipp32f* pSrc, Ipp32fc* pDst,
    IppsHilbertSpec_32f32fc* pSpec);
IppStatus ippsHilbert_16s32fc(const Ipp16s* pSrc, Ipp32fc* pDst,
    IppsHilbertSpec_16s32fc* pSpec);
IppStatus ippsHilbert_16s16sc_Sfs(const Ipp16s* pSrc, Ipp16sc* pDst,
    IppsHilbertSpec_16s16sc* pSpec, int scaleFactor);
```

引数

pSpec	ヒルベルト指定構造体へのポインタ。
pSrc	元の実数データを格納しているベクトルへのポインタ。
pDst	複素数データを格納する出力配列へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsHilbert は、ipps.h ファイルで宣言される。この関数は、元の実数信号 *pSrc* を実数部に格納し、計算されたヒルベルト変換を虚数部に格納する、複素数分析信号 *pDst* を計算する。ヒルベルト変換は、*pSpec* 指定パラメータ（サンプルの数を指定する *length* と、特別なコードを指定する *hint*）に従って実行される。入力データは、必要に応じて、*length* のサイズに合わせてゼロでパディングされるか、切り捨てられる。
整数データ型の場合、出力結果は *scaleFactor* の値に従ってスケーリングされる。これによって、出力信号の範囲と精度が保証される。

例 7-9 は、指定構造体の初期化と関数 ippsHilbert_32f32fc の使用例を示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pSpec* が NULL。
ippStsContextMatchErr	エラー。指定識別子 *pSpec* が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

例 7-9　　ヒルベルト関数の使用例

```
IppStatus hilbert(void) {
    Ipp32f x[10];
    Ipp32fc y[10];
    int n;
    IppStatus status;
    IppsHilbertSpec_32f32fc* spec;

    status = ippsHilbertInitAlloc_32f32fc(&spec, 10,
     ippAlgHintNone);
    for(n = 0; n < 9; n ++) {
        x[n] = (Ipp32f)cos(IPP_2PI * i * 2 / 9);
    }
    status = ippsHilbert_32f32fc(x, y, spec);
    ippsMagnitude_32fc((Ipp32fc*)y, x, 5);
    ippsHilbertFree_32f32fc(spec);
    printf_32f("hilbert magn =", x, 5, status);
    return status;
}

Output:
hilbert magn = 1.0944 1.1214 1.0413 0.9707 0.9839

Matlab* Analog:
>> n=0:9; x=cos(2*pi*n*2/9); y=abs(hilbert(x)); y(1:5)
```

ウェーブレット変換関数

この項では、インテル® IPP でサポートしているウェーブレット変換関数について説明する。

信号処理では、信号を周波数領域、および時間－周波数領域で表せる。多くの場合、ウェーブレット変換は、短時間フーリエ変換の代わりとして使用できる。

intel®

離散ウェーブレット信号は、「周波数」特性と「タイム・ロケーション」の 2 つのインデックスを持つ一連の係数 $a_{i,k}$ と見なせる。係数の値は、局所化された波形の大きさか基本変換関数の 1 つに対応する。「周波数」インデックスは、局所化された波形の時間スケールを示す。1 つの局所波形を時間領域で 2^n だけ減衰させて生じる関数の基底が最も広く使用されている。このような変換は、高速フーリエ変換の類推によって、高速ウェーブレット変換と呼ばれる非常に効率的な実装を実現するのに使用される。図 7-1 は、時間－周波数平面がどのようにして局所波形の大きさに対応した領域に分割されるかを示している。インテル IPP にはこのようなタイプの変換が実装されており、離散ウェーブレット変換（DWT）と呼ばれている。

DWT は、スケール係数 2 に関連する基本関数から派生したウェーブレット分析手法の一つである。したがって、DWT とその他のパケット分析手法で共用される共通の基本要素が存在する。

同様に、信号の再構成または合成用に別の基本要素も定義可能で、これは、1 レベルの逆方向 DWT と呼ばれている。図 7-2 は、順方向 DWT のダイアグラムを示している。これは、図 7-1 で示す時間－周波数表現に切り換えられる。このダイアグラムには、3 レベルの分解を説明している。それに対し、図 7-3 は、基本的な 1 レベルの逆方向変換をベースにした信号再構成手順を示す。

離散マルチスケール変換は、再サンプリング係数 2 が指定された補間フィルタおよびデシメーション・フィルタの使用をベースにして実装される。マルチスケール信号を分解および再構成する関数の基本により、フィルタ・パラメータは一意に定義される。インテル IPP マルチスケール変換関数は、有限インパルス応答を持つフィルタを使用する。

プリミティブには、2 つの関数セットが含まれる。

- 固定フィルタバンク用に設計された変換関数。これらの変換関数は、高いパフォーマンスを発揮する。
- 任意順序のフィルタを使用できる変換関数。これらの関数は、効率的なポリフェーズ・フィルタリング・アルゴリズムを使用する。変換インターフェイスでは、ブロックやリアルタイム・アプリケーション単位でデータを処理できる。

図 7-1　　時間－周波数領域におけるウェーブレット分解係数

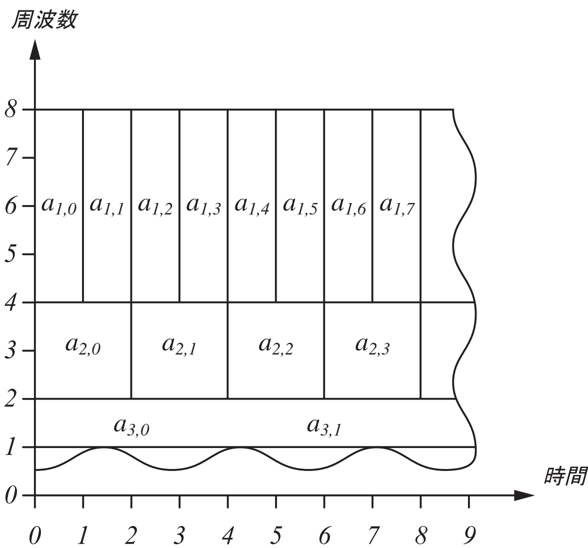


図 7-2　　レベルの離散ウェーブレット分解

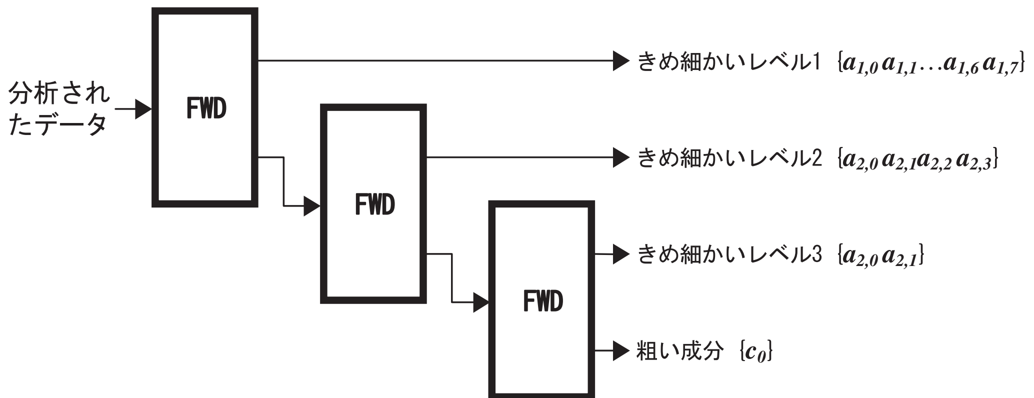
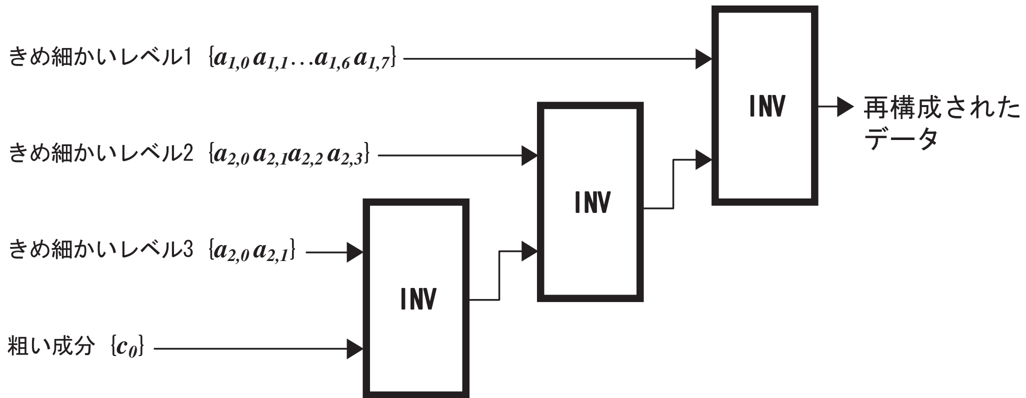


図 7-3　　レベルの離散ウェーブレット再構成



固定フィルタバンクに対する変換

この項では、固定フィルタバンクに対して順方向または逆方向のウェーブレット変換を実行する関数について説明する。

WTHaarFwd, WTHaarInv

順方向または逆方向の単一レベル離散
ウェーブレット Haar 変換を実行する。

```
IppStatus ippsWTHaarFwd_8s(const Ipp8s* pSrc, int lenSrc,
    Ipp8s* pDstLow, Ipp8s* pDstHigh);
IppStatus ippsWTHaarFwd_16s(const Ipp16s* pSrc, int lenSrc,
    Ipp16s* pDstLow, Ipp16s* pDstHigh);
IppStatus ippsWTHaarFwd_32s(const Ipp32s* pSrc, int lenSrc,
    Ipp32s* pDstLow, Ipp32s* pDstHigh);
IppStatus ippsWTHaarFwd_64s(const Ipp64s* pSrc, int lenSrc,
    Ipp64s* pDstLow, Ipp64s* pDstHigh);
IppStatus ippsWTHaarFwd_32f(const Ipp32f* pSrc, int lenSrc,
    Ipp32f* pDstLow, Ipp32f* pDstHigh);
IppStatus ippsWTHaarFwd_64f(const Ipp64f* pSrc, int lenSrc,
    Ipp64f* pDstLow, Ipp64f* pDstHigh);
```

```
IppStatus ippsWTHaarFwd_8s_Sfs(const Ipp8s* pSrc, int lenSrc,
    Ipp8s* pDstLow, Ipp8s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
    Ipp16s* pDstLow, Ipp16s* pDstHigh, int scaleFactor );
IppStatus ippsWTHaarFwd_32s_Sfs(const Ipp32s* pSrc, int lenSrc,
    Ipp32s* pDstLow, Ipp32s* pDstHigh, int scaleFactor);
IppStatus ippsWTHaarFwd_64s_Sfs(const Ipp64s* pSrc, int lenSrc,
    Ipp64s* pDstLow, Ipp64s* pDstHigh, int scaleFactor);
```

```
IppStatus ippsWTHaarInv_8s(const Ipp8s* pSrcLow, const Ipp8s* pSrcHigh,
    Ipp8s* pDst, int lenDst);
IppStatus ippsWTHaarInv_16s(const Ipp16s* pSrcLow, const Ipp16s*
    pSrcHigh, Ipp16s* pDst, int lenDst);
IppStatus ippsWTHaarInv_32s(const Ipp32s* pSrcLow, const Ipp32s*
    pSrcHigh, Ipp32s* pDst, int lenDst);
IppStatus ippsWTHaarInv_64s(const Ipp64s* pSrcLow, const Ipp64s*
    pSrcHigh, Ipp64s* pDst, int lenDst);
IppStatus ippsWTHaarInv_32f(const Ipp32f* pSrcLow, const Ipp32f*
    pSrcHigh, Ipp32f* pDst, int lenDst);
IppStatus ippsWTHaarInv_64f(const Ipp64f* pSrcLow, const Ipp64f*
    pSrcHigh, Ipp64f* pDst, int lenDst);
IppStatus ippsWTHaarInv_8s_Sfs(const Ipp8s* pSrcLow,
    const Ipp8s* pSrcHigh, Ipp8s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_16s_Sfs(const Ipp16s* pSrcLow,
    const Ipp16s* pSrcHigh, Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_32s_Sfs(const Ipp32s* pSrcLow,
    const Ipp32s* pSrcHigh, Ipp32s* pDst, int lenDst, int scaleFactor);
IppStatus ippsWTHaarInv_64s_Sfs(const Ipp64s* pSrcLow,
    const Ipp64s* pSrcHigh, Ipp64s* pDst, int lenDst, int scaleFactor);
```

引数

pSrc	ippsWTHaarFwd 関数への入力信号を格納する配列へのポインタ。
lenSrc	ソース・ベクトル *pSrc* 内の要素の数。
pDstLow	ippsWTHaarFwd 関数の出力に含まれる粗い「低周波数」成分を格納する配列へのポインタ。
pDstHigh	ippsWTHaarFwd 関数の出力に含まれるきめ細かな「高周波数」成分を格納する配列へのポインタ。
pSrcLow	ippsWTHaarInv 関数への入力に含まれる粗い「低周波数」成分を格納する配列へのポインタ。

pSrcHigh	ippsWTHaarInv 関数への入力に含まれるきめ細かな「高周波数」成分を格納する配列へのポインタ。
pDst	ippsWTHaarInv 関数の出力信号を格納する配列へのポインタ。
lenDst	デスティネーション・ベクトル *pDst* 内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsWTHaarFwd と ippsWTHaarInv は、ipps.h ファイルで宣言される。これらの関数は、順方向と逆方向の単一レベル離散 Haar 変換を実行する。これらの変換は直交であり、もとの信号を完全に再構成する。

順方向の変換は、ローパス・デシメーション・フィルタ係数として {1/2, 1/2}、ハイパス・デシメーション・フィルタ係数として {1/2, -1/2} が指定されたウェーブレット信号分解と見なせる。

逆方向の変換は、ローパス補間フィルタ係数として {1, 1}、ハイパス補間フィルタ係数として {-1, 1} が指定されたウェーブレット信号再構成として表現される。

分解フィルタ係数は、入力信号と出力信号に同じ値の範囲を提供するために正規化された周波数応答である。したがって、ゼロ値の周波数に対するローパス・フィルタ周波数応答の大きさは 1 となり、0.5 近傍の周波数値に対するハイパス・フィルタ周波数応答の大きさも 1 となる。

補間フィルタ係数の絶対値は 1 であるため、信号の再構成に必要な演算はごくわずかである。これは、データ圧縮アプリケーションでの使用に大変適している。分解フィルタ係数は 2 の累乗であるため、整数関数は *scaleFactor*=-1 を使用して、損失のない分解を実行する。飽和を回避するには、より精度の高いデータ型を使用する。

フィルタ係数は、パワー・スペクトル応答を正規化できる（詳しくは、[Str96] を参照）。したがって、分解フィルタ係数は $\{2^{-1/2}, 2^{-1/2}\}$ と $\{2^{-1/2}, -2^{-1/2}\}$、再構成フィルタ係数は $\{2^{-1/2}, 2^{-1/2}\}$ と $\{-2^{-1/2}, 2^{-1/2}\}$ になる。

次に示す順方向の単一レベル離散 Haar 変換の定義では、*N=lenSrc* である。粗い「低周波数」成分 *c(k)* は *pDstLow[k]*、きめ細かな「高周波数」成分 *d(k)* は *pDstHigh[k]* である。また、*x(2k)* と x(2*k*+1) はそれぞれ、入力信号 *pSrc* の偶数値と奇数値である。

$$c(k) = (x(2k) + x(2k+1))/2$$

$$d(k) = (x(2k+1) - x(2k))/2$$

逆方向の変換では、*N=lenDst* である。粗い「低周波数」成分 *c(k)* は *pSrcLow[k]*、きめ細かな「高周波数」成分 *d(k)* は *pSrcHigh[k]* である。また *y(2i)* と y(2*i+1*) はそれぞれ、出力信号 *pDst* の偶数値と奇数値である。

$y(2i)=c(i)-d(i)$

$y(2i+1)=c(i)+d(i)$

N の長さが偶数の場合、$0 \le k<N/2$、および $0 \le i<N/2$ である。また、元信号と再構成した信号に対する「低周波数」成分と「高周波数」成分のサイズは、ともに *N*/2 である。成分の長さの合計は、信号の長さ *N* に等しい。

N の長さが奇数の場合、ベクトルは、長さが *N*+1 に拡張されたベクトルと見なされる。このベクトルの最後の 2 つの要素は互いに等しく、$x[N]=x[N-1]$ の関係がある。分解された信号の粗い成分ときめ細かな成分の最後の要素は、次のように定義される。

$c((N+1)/2-1)=x(N-1)$

$d((N+1)/2-1)=0$

同様に、再構成された信号の最後の成分は、次のように定義される。

$y(N)=y(N-1)=c((N+1)/2-1)$

N の長さが奇数の場合は、$0 \le k<(N-1)/2$、および $0 \le i<(N-1)/2$ である。ここでは、$c((N+1)/2-1)=x(N-1)$、および $y(N-1)=c((N+1)/2-1)$ を仮定している。「低周波数」成分のサイズは $(N+1)/2$ である。「高周波数」成分のサイズは $(N-1)/2$ である。これは、最後の要素 $d((N+1)/2-1)$ が常にゼロに等しいからである。また、成分の長さの合計は *N* となる。

このようなアプローチは、対称特性を持つ連続したフィルタ境界に適用される。詳細は、[Bri94] を参照のこと。

ブロック・モード変換を実行する場合、偶数長の信号の分解と再構成には境界での補外は使用されないのに注意する。信号の長さが奇数の場合は、信号境界の対称連続性と最終点のレプリカが適用される。

出力ブロックの連続したセットが必要な場合は、最後の入力ブロックを含めた入力ブロック全体が偶数長でなければならない（最後の入力ブロックは、奇数長、偶数長のどちらにもできる）。したがって、要素の総数が奇数の場合は、最後のブロックだけを奇数長にする。

ippsWTHaarFwd。関数 ippsWTHaarFwd は、長さ *lenSrc* の信号 *pSrc* に対して順方向の単一レベル離散 Haar 変換を実行し、分解された粗い「低周波数」成分を *pDstLow* に、きめ細かな「高周波数」成分を *pDstHigh* に格納する。

ippsWTHaarInv。関数 ippsWTHaarInv は、粗い「低周波数」成分 *pSrcLow* ときめ細かな「高周波数」成分 *pSrcHigh* に対して逆方向の単一レベル離散 Haar 変換を実行し、再構成された信号を長さ *lenDst* のベクトル *pDst* に格納する。

例 7-10 は、関数 ippsWTHaarFwd_32f の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* の値が関数 ippsWinBlackmanOpt に対して 4 未満、ファミリに属するその他すべての関数に対して 3 未満である。

例 7-10 ippsWTHaarFwd 関数の使用例

```
IppStatus wthaar(void) {
      Ipp32f x[8], lo[4], hi[4];
      IppStatus status;
      ippsSet_32f(7, x, 8);  --x[4];
      status = ippsWTHaarFwd_32f(x, 8, lo, hi);
      printf_32f("WT Haar low  =", lo, 4, status);
      printf_32f("WT Haar high =", hi, 4, status);
      return status;
}
Output:
    WT Haar low  =  7.000000 7.000000 6.500000 7.000000
    WT Haar high =  0.000000 0.000000 0.500000 0.000000
```

関連項目

ウェーブレット変換の詳細は、[Str96]『Wavelet and Filter Banks』153 ～ 157 ページ Wellesley-Cambridge Press と、[Bri94]『Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks』Los Alamos Report LA-UR-94-1747/1994 年を参照のこと。これらの参考資料の詳細は、本書の最後にある「参考文献」を参照のこと。

ユーザ・フィルタバンクに対する変換

この項では、ユーザ・フィルタバンクに対して順方向または逆方向のウェーブレット変換を実行する関数について説明する。

WTFwdInitAlloc, WTInvInitAlloc

ウェーブレット変換の構造体を初期化する。

```
IppStatus ippsWTFwdInitAlloc_32f(IppsWTFwdState_32f** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow,
    const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_8s32f(IppsWTFwdState_8s32f** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow,
    const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_8u32f(IppsWTFwdState_8u32f** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f*
    pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_16s32f(IppsWTFwdState_16s32f** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f*
    pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTFwdInitAlloc_16u32f(IppsWTFwdState_16u32f** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f*
    pTapsHigh, int lenHigh, int offsHigh);

IppStatus ippsWTInvInitAlloc_32f(IppsWTInvState_32f** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f*
    pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f8s(IppsWTInvState_32f8s** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f*
    pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f8u(IppsWTInvState_32f8u** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow,
    const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f16s(IppsWTInvState_32f16s** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow,
    const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
IppStatus ippsWTInvInitAlloc_32f16u(IppsWTInvState_32f16u** pState,
    const Ipp32f* pTapsLow, int lenLow, int offsLow,
    const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

引数

pState	割り当てられ、初期化されたステート構造体を格納するロケーションへのポインタ。
pTapsLow	ローパス・フィルタ・タップのベクトルへのポインタ。
lenLow	ローパス・フィルタのタップの数。
offsLow	ローパス・フィルタの追加の遅延（オフセット)。
pTapsHigh	ハイパス・フィルタ・タップのベクトルへのポインタ。
lenHigh	ハイパス・フィルタのタップの数。
offsHigh	ハイパス・フィルタの追加の遅延（オフセット)。

説明

関数 `ippsWTFwdInitAlloc` と `ippsWTInvInitAlloc` は、`ipps.h` ファイルで宣言される。これらの関数は、ローパス・フィルタおよびハイパス・フィルタのタップ *pTapsLow* と *pTapsHigh*、長さ *lenLow* と *lenHigh*、追加の入力遅延 *offsLow* と *offsHigh* の各パラメータを使用して、WT ステート構造体 *pState* を生成し、初期化する。

ippsWTFwdInitAlloc。関数 `ippsWTFwdInitAlloc` は、順方向の WT ステート構造体を初期化する。

ippsWTInvInitAlloc。関数 `ippsWTInvInitAlloc` は、逆方向の WT ステート構造体を初期化する。

アプリケーション・ノート

これらの関数は、ウェーブレット変換ステート構造体の初期化、ステート構造体へのメモリ割り当て、メモリの初期化を行い、ステート構造体に *pState* ポインタを返す。初期化の手順は、順方向変換と逆方向変換でそれぞれ別々に実装される。

順方向と逆方向のウェーブレット変換を両方実行するには、2 つのステート構造体を別々に生成する。一般に、順方向変換と逆方向変換の初期化パラメータの意味は同じである。どちらの関数にも、フィルタの組を記述するパラメータがある。順方向変換では、分析フィルタの組のタップ *pTapsHigh* と *pTapsLow*、長さ *lenHigh* と *lenLow* を使用する。逆方向変換では、合成フィルタの組のタップ *pTapsHigh* と *pTapsLow*、長さ *lenHigh* と *lenLow* が使用される。長さとタップのセットに加えて、関数では、各フィルタごとに追加の遅延 *offsLow* と *offsHigh* を指定できる。遅延の値は調節可能であるため、次のものを同期化できる。

- ハイパス・フィルタとローパス・フィルタのグループ遅延。

- マルチレベルの分解および再構成アルゴリズムにおける異なるレベルのデータ間の遅延。

これらのパラメータの使用方法は、7-63 ページの「WTFwd」、7-69 ページの「WTInv」を参照のこと。順方向変換の場合、追加の遅延に指定可能な最小の値は –1 である。逆方向変換の場合、遅延の値はゼロ以上にする必要がある。追加の遅延の値の選択方法は、7-63 ページの「WTFwd」、7-69 ページの「WTInv」の例を参照のこと。初期化関数は、フィルタ・タップをステート構造体 *pState* にコピーする。そのため、ポインタで指示されるメモリはすべて、関数の演算の完了後に解放または変更できる。メモリが不足している場合、関数は構造体にゼロ・ポインタを設定する。

境界の補間。一般に、境界が定められた信号に対する可逆的ウェーブレット変換では、片側または両側でのデータ補間が必要である。内部の遅延線はすべて、初期化時にゼロに設定される。ゼロ以外の信号プレヒストリを設定するには、7-67 ページの関数 ippsWTFwdSetDlyLine を呼び出す。制限されたデータ・セットがすべて処理されると、データ・ベクトルの開始地点と終了地点の両方で、データ補間を実行できる。ソース・データとその初期補間は遅延線を形成するのに使用するため、信号の残りは、メイン・ブロックと信号端に分割される。信号端データとそれらの補間は、最後のブロックを構築するのに使用する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pState*、*pTapsHigh*、または *pTapsLow* が NULL。
ippStsSizeErr	エラー。*lenLow* または *lenHigh* がゼロ以下。
ippStsWtOffsetErr	エラー。フィルタ遅延 *offsLow* または *offsHigh* が順方向変換に対して –1 未満、逆方向変換に対してゼロ未満である。

WTFwdFree, WTInvFree

ウェーブレット変換の構造体をクローズする。

```
IppStatus ippsWTFwdFree_32f(IppsWTFwdState_32f* pState);
IppStatus ippsWTFwdFree_8s32f(IppsWTFwdState_8s32f* pState);
IppStatus ippsWTFwdFree_8u32f(IppsWTFwdState_8u32f* pState);
IppStatus ippsWTFwdFree_16s32f(IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwdFree_16u32f(IppsWTFwdState_16u32f* pState);
```

```
IppStatus ippsWTInvFree_32f(IppsWTInvState_32f* pState);
IppStatus ippsWTInvFree_32f8s(IppsWTInvState_32f8s* pState);
IppStatus ippsWTInvFree_32f8u(IppsWTInvState_32f8u* pState);
IppStatus ippsWTInvFree_32f16s (IppsWTInvState_32f16s* pState);
IppStatus ippsWTInvFree_32f16u(IppsWTInvState_32f16u* pState);
```

引数

pState	クローズするステート構造体へのポインタ。

説明

関数 ippsWTFwdFree と ippsWTInvFree は、ipps.h ファイルで宣言される。これらの関数は、ippsWTFwdInitAlloc または ippsWTInvInitAlloc により生成されたステートに関連付けられている内部メモリをすべて解放して、WT ステート構造体ステート構造体 *pState* をクローズする。変換が完了したら、ippsWTFwdFree または ippsWTInvFree を呼び出す。*pState* ポインタが NULL の場合、演算は実行されず、ippStsNullPtrErr ステータスが返される。

ippsWTFwdFree。関数 ippsWTFwdFree は、順方向の WT ステート構造体をクローズする。

ippsWTInvFree。関数 ippsWTInvFree は、逆方向の WT ステート構造体をクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pState* が NULL。
ippStsStateMatchErr	エラー。ステート識別子 *pState* が正しくない。

WTFwd

順方向ウェーブレット変換を計算する。

```
IppStatus ippsWTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDstLow,
    Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_32f* pState);
IppStatus ippsWTFwd_8s32f(const Ipp8s* pSrc, Ipp32f* pDstLow,
    Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_8s32f* pState);
IppStatus ippsWTFwd_8u32f(const Ipp8u* pSrc, Ipp32f* pDstLow,
    Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_8u32f* pState);
```

```
IppStatus ippsWTFwd_16s32f(const Ipp16s* pSrc, Ipp32f* pDstLow,
    Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwd_16u32f(const Ipp16u* pSrc, Ipp32f* pDstLow,
    Ipp32f* pDstHigh, int dstLen, IppsWTFwdState_16u32f* pState);
```

引数

pSrc	分解する入力信号を格納するベクトルへのポインタ。
pDstLow	出力に含まれる粗い「低周波数」成分を格納するベクトルへのポインタ。
pDstHigh	出力に含まれるきめ細かな「高周波数」成分を格納するベクトルへのポインタ。
dstLen	ベクトル *pDstHigh* および *pDstLow* 内の要素の数。
pState	ステート構造体へのポインタ。

説明

関数 `ippsWTFwd` は、`ipps.h` ファイルで宣言される。この関数は、順方向ウェーブレット変換を計算する。この関数は、長さ（2 * *dstLen*）のソース・データ・ブロック *pSrc* を変換し、「低周波数」成分 *pDstLow* と「高周波数」成分 *pDstLow* を生成する。変換パラメータは、ステート構造体 *pState* で指定される。

アプリケーション・ノート

これらの関数は、1 レベルの順方向離散マルチスケール変換を実行する。図 7-4 は、対応する変換ダイアグラムを示す。入力信号は、「低周波数」成分と「高周波数」成分に分解される。フィルタの変換特性は、初期化時に設定される係数で決まる。これらの関数はデータをブロック処理するように設計しており、変換ステート構造体 *pState* には必要なフィルタ遅延線がすべて含まれる。これらの主要な遅延線に加え、各関数は、各フィルタごとに他の遅延線を持っている。調節可能な特別の遅延線を使用すると、ハイパス・フィルタとローパス・フィルタのグループ遅延時間を容易に同期化できる。また、信号分解のマルチレベル・システムで、異なる分解レベル間の遅延も同期化できる。

入力データ・ブロックと出力データ・ブロックの長さ。関数は偶数長の信号ブロックを分解するように設計されているため、入力成分の長さを指定する 1 つのパラメータしか用意されていない。入力ブロックの長さは、各成分のサイズの 2 倍でなければならない。

フィルタ・グループ遅延の同期化。アプリケーションによっては、ハイパス・フィルタとローパス・フィルタの時間応答を同期させなければならないものがある。このような同期の典型的な例としては、異なる長さを持つ対称フィルタの同期化がある。

長さがそれぞれ 6 と 2 であるスプライン・フィルタの双直交セットの例を次に示す。

```
static const float decLow[6] =
{
    -6.25000000e-002f,
     6.25000000e-002f,
     5.00000000e-001f,
     5.00000000e-001f,
     6.25000000e-002f,
     -6.25000000e-002f
};


static const float decHigh[2] =
{
    -5.00000000e-001f,
     5.00000000e-001f
};
```

この場合、ローパス・フィルタは、ハイパス・フィルタより 2 サンプル長い遅延を提供する。これは、追加の初期化関数の遅延間の差として設定すべき値に等しい。共通の信号遅延を最小にするには、*offsLow*=−1、*offsHigh*=−1+2=1 を選択する必要がある。この場合、フィルタ遅延のグループ時間は、追加の遅延によって平衡化される。遅延時間の合計は、分解ステージで、元信号の時間フレームで 2 サンプルの値を持つローパス・フィルタ・グループ遅延に等しい。

注： 双直交フィルタバンクと直交フィルタバンクは、1 つの固有の特異性によって区別される。すなわち、順方向変換の追加の遅延は、信号を完全に再構成するために、一様に偶数でなければならない。

マルチレベル分解アルゴリズム。マルチレベル分解アルゴリズムを実装するには、各種のレベルの成分に対して信号の遅延を同期化する必要がある。

これについては、図 7-2 の 3 レベルの分解の例で説明する。変換では、長さがそれぞれ 6 と 2 のスプライン・フィルタの双直交セットを使用していると仮定する。グループ遅延は明確に同期化する必要があるため、最後のレベルに対する追加のフィルタ遅延として *offsLow3*=−1、*offsHigh3*=1 を選択する。分解の最終ステージにおけ

るこのフィルタ・セットの遅延の合計は 2 サンプルである。この値は、分解の最終ステージの入力時間スケールに対応する。2 番目のレベルの「きめ細かな」部分の遅延を等しくするため、遅延は 2*2 サンプルで増加する必要がある。

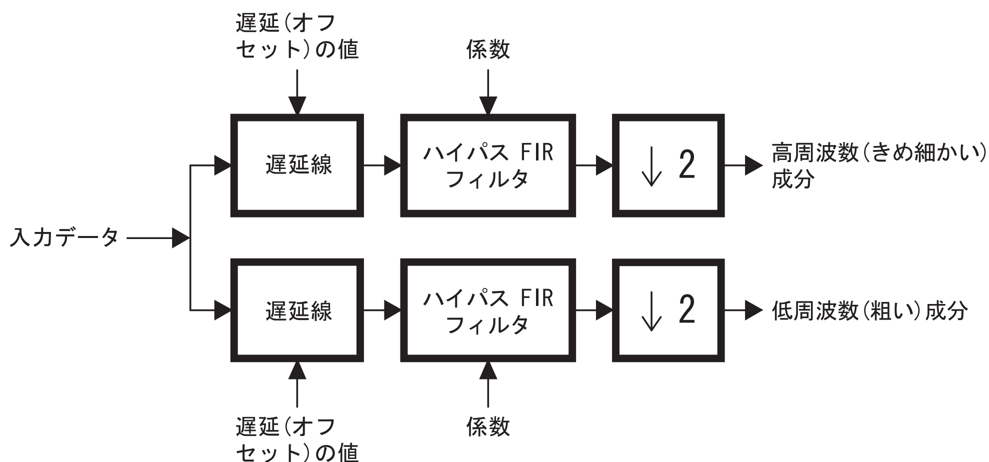
2 番目のレベルの追加の遅延の値は、それぞれ *offsLow2*=-1、*offsHigh2*=*offsHigh3*+4=5 である。分解の最初のレベル *offsLow1*=-1、*offsHigh1*=*offsHigh2*+2*4=13 には、より大きな値の「高周波数」成分の遅延を選択する必要がある。

3 レベルの分解の遅延の合計は 12 サンプルである。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。データ・ブロックへのポインタが NULL。
`ippStsContextMatchErr`	エラー。ステート識別子 *pState* が正しくない。
`ippStsSizeErr`	エラー。*dstLen* または *srcLen* がゼロ以下。

図 7-4 レベルの順方向ウェーブレット変換



WTFwdSetDlyLine, WTFwdGetDlyLine

順方向ウェーブレット変換の遅延線を設定
および取得する。

```
IppStatus ippsWTFwdSetDlyLine_32f(IppsWTFwdState_32f* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8s32f(IppsWTFwdState_8s32f* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_8u32f(IppsWTFwdState_8u32f* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_16s32f(IppsWTFwdState_16s32f* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTFwdSetDlyLine_16u32f(IppsWTFwdState_16u32f* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTFwdGetDlyLine_32f(IppsWTFwdState_32f* pState,
    Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_8s32f(IppsWTFwdState_8s32f* pState,
    Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_8u32f(IppsWTFwdState_8u32f* pState,
    Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_16s32f(IppsWTFwdState_16s32f* pState,
    Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTFwdGetDlyLine_16u32f(IppsWTFwdState_16u32f* pState,
    Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
```

引数

pState	ステート構造体へのポインタ。
pDlyLow	「低周波数」成分の遅延線を格納するベクトルへのポインタ。
pDlyHigh	「高周波数」成分の遅延線を格納するベクトルへのポインタ。

説明

関数 ippsWTFwdSetDlyLine と ippsWTFwdSetDlyLine は、ipps.h ファイルで宣言される。これらの関数は、*pDlyHigh* と *pDlyLow* から遅延線の値をコピーし、それらをステート構造体 *pState* に格納する。

ippsWTFwdSetDlyLine。関数 ippsWTFwdSetDlyLine は、順方向 WT ステートの遅延線の値を設定する。

ippsWTFwdGetDlyLine。関数 ippsWTFwdSetDlyLine は、逆方向 WT ステートの遅延線の値を設定する。

アプリケーション・ノート

これらの関数は、信号プレヒストリを形成し、遅延線を格納および再構成するように設計されている。遅延線は、ハイパス・フィルタとローパス・フィルタにそれぞれ実装される。そのため、各フィルタの独立した信号プレヒストリも取得できる。

遅延線のデータ形式。変換内ではどのような形式の遅延線でも使用できるが、関数は、最も簡潔な形式の受け取りベクトルと返りベクトルを提供する。遅延線に転送されるデータ、または遅延線から返されるデータは、順方向変換関数に入力される初期信号と同じ形式を持つ。すなわち、遅延線ベクトルは、初期信号と同じ時間フレーム内の連続した信号プレヒストリ・カウントで構成される必要がある。

遅延線の長さ。遅延線の実装や関数の読み取りで送られたり、受け取られたりするベクトルの長さは、フィルタの長さと追加のフィルタ遅延の値によって一意に定義される。

「低周波数」成分フィルタの遅延線ベクトルの長さは、次の式で定義される。

dlyLowLen = *lenLow* + *offsLow* - 1,

lenLow と *offsLow* は、それぞれ「低周波数」成分フィルタの長さと追加の遅延である。

「高周波数」成分フィルタの遅延線ベクトルの長さは、次の式で定義される。

dlyHighLen = *lenHigh* + *offsHigh* - 1,

lenHigh と *offsHigh* は、それぞれ「高周波数」成分フィルタの長さと追加の遅延である。

lenLow、*offsLow*、*lenHigh*、*offsHigh* の各パラメータは、変換の初期化に関する項でも説明している。詳しくは、7-60 ページの「WTFwdInitAlloc、WTInvInitAlloc」を参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDlyLow* または *pDlyHigh* が NULL。
ippStsStateMatchErr	エラー。ステート識別子 *pState* が正しくない。

intel®

WTInv

逆方向ウェーブレット変換を計算する。

```
IppStatus ippsWTInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
    int srcLen, Ipp32f* pDst, IppsWTInvState_32f* pState);
IppStatus ippsWTInv_32f8s(const Ipp32f* pSrcLow, const Ipp32f*
    pSrcHigh, int srcLen, Ipp8s* pDst, IppsWTInvState_32f8s* pState);
IppStatus ippsWTInv_32f8u(const Ipp32f* pSrcLow, const Ipp32f*
    pSrcHigh, int srcLen, Ipp8u* pDst, IppsWTInvState_32f8u* pState);
IppStatus ippsWTInv_32f16s(const Ipp32f* pSrcLow, const Ipp32f*
    pSrcHigh, int srcLen, Ipp16s* pDst, IppsWTInvState_32f16s* pState);
IppStatus ippsWTInv_32f16u(const Ipp32f* pSrcLow, const Ipp32f*
    pSrcHigh, int srcLen, Ipp16u* pDst, IppsWTInvState_32f16u* pState);
```

引数

pSrcLow	入力に含まれる粗い「低周波数」成分を格納するベクトルへのポインタ。
pSrcHigh	きめ細かな「高周波数」成分を格納するベクトルへのポインタ。
srcLen	ベクトル *pSrcHigh* と *pSrcLow* 内の要素の数。
pDst	再構成した出力信号を格納するベクトルへのポインタ。
pState	ステート構造体へのポインタ。

説明

関数 ippsDCTInv は、ipps.h ファイルで宣言される。この関数は、逆方向ウェーブレット変換を計算する。この関数は、「低周波数」成分 *pSrcLow* と「高周波数」成分 *pSrcHigh* を変換し、長さ (2**srcLen*) のデスティネーション・データ・ブロック *pDst* を生成する。変換パラメータは、ステート構造体 *pState* で指定される。

アプリケーション・ノート

これらの関数は、1 レベルの逆方向マルチスケール変換で使用される。この変換は、「低周波数」と「高周波数」の 2 つの成分から元の信号を再構成する。図 7-5 は、対応する変換アルゴリズムを示す。信号の再構成には 2 つの補間フィルタが使用されるが、これらの係数は初期化時に設定する。順方向変換の実装と同様、逆方向変換

の実装では、フィルタ遅延のグループ時間を同期化するのに必要な追加の遅延線や、データ再構成の各種のレベルに渡る遅延を同期化するのに必要な追加の遅延線が組み込まれる。

入力と出力データのブロック長。これらの機能は、偶数長の信号のブロックを改善するように設計されている。信号の成分の長さは入力されたデータと同じでなければならない。改善された信号の出力ブロック長は、各成分の 2 倍の長さでなければならない。

フィルタ・グループ遅延の同期化。この例では、長さが 2 と 6 のスプライン・フィルタの双直交セットについて検討する。

```
static const float recLow[2] =
{
     1.00000000e+000f,
     1.00000000e+000f
};
static const float recHigh[6] =
{
    -1.25000000e-001f,
    -1.25000000e-001f,
     1.00000000e+000f,
    -1.00000000e+000f,
     1.25000000e-001f,
     1.25000000e-001f
};
```

このフィルタ・セットは、順方向変換の同じタイトルの項で検討したフィルタ・セットに対応する。詳しくは、7-63 ページの「WTFwd」を参照のこと。

前述した例とは異なり、今回はハイパス・フィルタが、低周波数フィルタと比べて 2 サンプルだけ大きな遅延を生成する。2 サンプルの差は、初期化関数の追加の遅延間にも存在する必要がある。遅延の合計を最小限にするには、追加の遅延のパラメータ *offsLow*=2、*offsHigh*=0 を選択する必要がある。この場合、遅延の合計は、ハイパス・フィルタのグループ遅延に等しい。これは、分解ステージでの元信号の時間フレーム内の 2 サンプルに等しい。

1 レベルの分解と再構成の遅延の合計は、分解ステージの遅延を考慮して 4 サンプルに等しい。

注： 双直交フィルタバンクと直交フィルタバンクは、1 つの固有の特異性によって区別される。すなわち、信号を完全に再構成するために、逆方向変換の追加の遅延は、一様に偶数でなければならず、また、分解遅延の偶数性と逆でなければならない。

マルチレベル再構成アルゴリズム。図 7-3 は、3 レベルの信号再構成アルゴリズムの例を示す。この手法は、順方向変換に関する項で説明した分解手法に対応する。詳しくは、7-63 ページの「WTFwd」を参照のこと。したがって、逆方向変換には、長さがそれぞれ 6 と 2 のスプライン・フィルタの双直交セットが使用される。最低レベルのフィルタ遅延は、*offsLow3*=2、*offsHigh3*=0 に設定されている。このステージでの再構成の遅延の合計は 2 サンプルに等しい。中央レベルの「きめ細かな」部分の遅延を等しくするには、遅延を増やす必要がある。

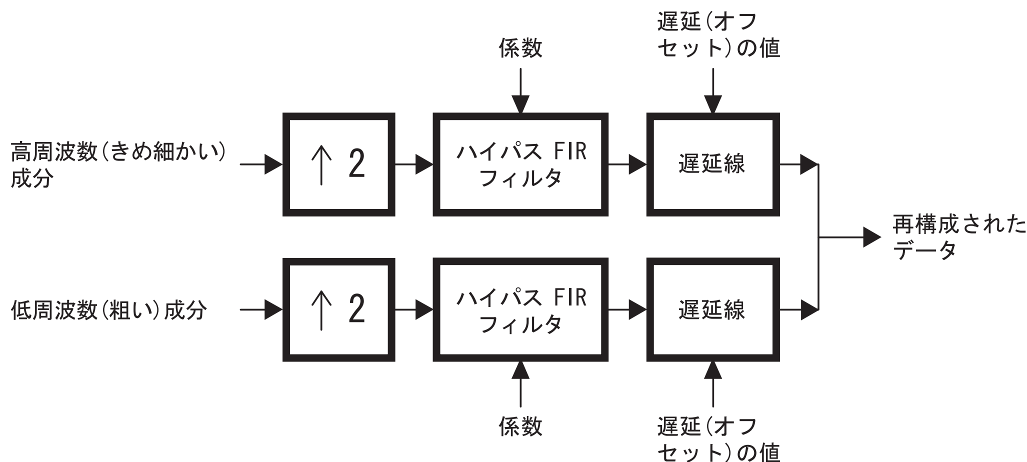
2 番目のレベルの追加の遅延の値は、それぞれ *offsLow2*=2、*offsHigh2*=*offsHigh3*+2*2=4 である。再構成の最後のレベル *offsLow1*=-1、*offsHigh1*=*offsHigh2*+2*4=12 には、より大きな値の高周波数成分の遅延を選択する。

3 レベルの再構成の遅延の合計は 12 サンプルである。3 レベルの分解および再構成サイクルの遅延の合計は 24 サンプルである。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。データ・ブロックへのポインタが NULL。
`ippStsStateMatchErr`	エラー。ステート識別子 *pState* が正しくない。
`ippStsSizeErr`	エラー。*dstLen* または *srcLen* がゼロ以下。

図 7-5　レベルの逆方向ウェーブレット変換



WTInvSetDlyLine, WTInvGetDlyLine

逆方向ウェーブレット変換の遅延線を設定
および取得する。

```
IppStatus ippsWTInvSetDlyLine_32f(IppsWTInvState_32f* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8s(IppsWTInvState_32f8s* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f8u(IppsWTInvState_32f8u* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f16s(IppsWTInvState_32f16s* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);
IppStatus ippsWTInvSetDlyLine_32f16u(IppsWTInvState_32f16u* pState,
    const Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f(IppsWTInvState_32f* pState, Ipp32f*
    pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f8s(IppsWTInvState_32f8s* pState,
    Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
```

```
IppStatus ippsWTInvGetDlyLine_32f8u(IppsWTInvState_32f8u* pState,
    Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f16s(IppsWTInvState_32f16s* pState,
    Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
IppStatus ippsWTInvGetDlyLine_32f16u(IppsWTInvState_32f16u* pState,
    Ipp32f* pDlyLow, Ipp32f* pDlyHigh);
```

引数

pState	ステート構造体へのポインタ。
pDlyLow	「低周波数」成分の遅延線を格納するベクトルへのポインタ。
pDlyHigh	「高周波数」成分の遅延線を格納するベクトルへのポインタ。

説明

関数 ippsWTFwdSetDlyLine と ippsWTFwdSetDlyLine は、ipps.h ファイルで宣言される。これらの関数は、*pDlyHigh* と *pDlyLow* から遅延線の値をコピーし、それらをステート構造体 *pState* に格納する。

ippsWTFwdSetDlyLine。関数 ippsWTFwdSetDlyLine は、順方向 WT ステートの遅延線の値を設定する。

ippsWTInvSetDlyLine。関数 ippsWTFwdSetDlyLine は、逆方向 WT ステートの遅延線の値を設定する。

アプリケーション・ノート

これらの関数は、逆方向マルチスケール変換の遅延線に対する設定と読み取りを行う。これらの関数は、フィルタの低周波数成分と高周波数成分の遅延線ベクトルを受け取ったり、返したりする。また、これらの関数を使用して、各成分の以前のヒストリを形成できる。インストール関数と読み取り関数をあわせて使用すると、各フィルタからの遅延線を格納し、再構成できる。

遅延線のデータ形式。変換内ではどのような形式の遅延線でも使用できるが、関数は、最も簡潔な形式の受け取りベクトルと返りベクトルを提供する。遅延線に転送されるデータと遅延線から返されるデータは、逆方向変換関数の入力での低周波数成分と高周波数成分と同じ形式を持つ。したがって、遅延線ベクトルは、入力成分と同じ時間フレーム内の連続する信号プレヒストリ・カウントで構成されなければならない。

遅延線の長さ。遅延線の実装や関数の読み取りで送られたり、受け取られたりするベクトルの長さは、フィルタの長さと追加のフィルタ遅延の値によって一意に定義される。

「低周波数」成分フィルタの遅延線ベクトルの長さを C 言語で記述すると、次の式になる。（ここでは簡便のため、2 で整数除算を行っている）

dlyLowLen = (*lenLow* + *offsLow* - 1) / 2,

ここで、*lenLow* と *offsLow* は、それぞれ「低周波数」成分フィルタの長さと追加の遅延である。

「高周波数」成分フィルタの遅延線ベクトルの長さを C 言語で記述すると、次の式になる。

dlyHighLen = (*lenHigh* + *offsHigh* - 1) / 2,

ここで、*lenHigh* と *offsHigh* は、それぞれ「高周波数」成分フィルタの長さと追加の遅延である。

lenLow、*offsLow*、*lenHigh*、*offsHigh* の各パラメータについては、変換の初期化に関する項でも説明されている。詳しくは、7-60 ページの「WTFwdInitAlloc、WTInvInitAlloc」を参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDlyLow* または *pDlyHigh* が NULL。
ippStsStateMatchErr	エラー。ステート識別子 *pState* が正しくない。

音声認識関数

8

本章では、音声認識アプリケーションで使用されるインテル® IPP 関数について説明する。

このグループのすべての関数を 表 8-1 に示す。

表 8-1　インテル® IPP 音声認識関数

関数の基本名	操作
基本算術関数	
AddAllRowSum	行列の列ベクトルの和を計算し、1 つのベクトルに加算する。
SumColumn	行列の列ベクトルの和を計算する。
SumRow	行列の列ベクトルの和を計算する。
SubRow	すべての行列の行からベクトルを引く。
CopyColumn_Indirect	列をリダイレクトして、入力行列をコピーする。
BlockDMatrixInitAlloc	対称ブロック対角行列を表す構造体を初期化する。
BlockDMatrixFree	ブロック対角行列の構造体を解放する。
NthMaxElement	ベクトルの N 番目に大きい要素を検索する。
VecMatMul	ベクトルに行列を掛ける。
MatVecMul	行列にベクトルを掛ける。
特徴処理関数	
ZeroMean	入力ベクトルから平均値を引く。
CompensateOffset	入力信号の DC オフセットを除去する。
SignChangeRate	入力信号のゼロ交差レートをカウントする。
LinearPrediction	入力ベクトルの線形予測分析を実行する。
Durbin	自己相関の入力ベクトルに Durbin の再帰を実行する。
Schur	Schur アルゴリズムを使用して反射係数を計算する。
LPToSpectrum	平滑化された大きさスペクトルを計算する。
LPToCepstrum	線形予測係数からケプストラム係数を計算する。
CepstrumToLP	ケプストラム係数から線形予測係数を計算する。
LPToReflection	線形予測係数から線形予測反射係数を計算する。
ReflectionToLP	線形予測反射係数から線形予測係数を計算する。
ReflectionToAR	反射係数を面積比に変換する。
ReflectionToTilt	ライズ / フォール / 接続パラメータのティルトを計算する。

表 8-1　インテル® IPP 音声認識関数 （続き）

関数の基本名	操作
PitchmarkToF0	ティルトに対してライズとフォールの大きさと持続時間を計算する。
UnitCurve	ライズ係数とフォール係数に対してティルトを計算する。
LPToLSP	線形予測係数から線スペクトル対ベクトルを計算する。
LSPToLP	線スペクトル対ベクトルを線形予測係数に変換する。
MelToLinear	メルスケール値をリニアスケール値に変換する。
LinearToMel	リニアスケール値をメルスケール値に変換する。
CopyWithPadding	ゼロのパディングを使用して、入力信号を出力にコピーする。
MelFBankGetSize	メル周波数フィルタ・バンク構造体のサイズを取得する。
MelFBankInit	メル周波数フィルタ・バンク分析を実行するための構造体を初期化する。
MelFBankInitAlloc	メル周波数フィルタ・バンク分析を実行するための構造体を初期化し、メモリを割り当てる。
MelLinFBankInitAlloc	リニア周波数とメル周波数を組み合わせたフィルタ・バンク分析を実行するための構造体を初期化する。
EmptyFBankInitAlloc	空のフィルタ・バンク構造体を初期化する。
FBankFree	フィルタ・バンク分析用の構造体を破壊する。
FBankGetCenters	三角フィルタ・バンクの中心周波数を取得する。
FBankSetCenters	三角フィルタ・バンクの中心周波数を設定する。
FBankGetCoeffs	フィルタ・バンクの重み係数を取得する。
FBankSetCoeffs	フィルタ・バンクの重み係数を設定する。
EvalFBank	フィルタ・バンク分析を実行する。
DCTLifterGetSize_MulC0	DCT 構造体のサイズを取得する。
DCTLifterInit_MulC0	DCT の実行と DCT 係数のリフタリングに使用される構造体を初期化する。
DCTLifterInitAlloc	DCT の実行と DCT 係数のリフタリングに使用される構造体を初期化し、メモリを割り当てる。
DCTLifterFree	DCT およびリフタリングに使用された構造体を破壊する。
DCTLifter	DCT を実行し、DCT 係数をリフタリングする。
NormEnergy	エネルギー値のベクトルを正規化する。
SumMeanVar	ベクトルの和と 2 乗和の両方を計算する。
NewVar	和のアキュムレータと 2 乗和のアキュムレータに基づいて分散値を計算する。
RecSqrt	ベクトルの平方根と、その逆数を計算する。
AccCovarianceMatrix	共分散行列を累算する。
導関数	
CopyColumn	入力シーケンスを、出力シーケンスにコピーする。
EvalDelta	特徴ベクトルの導関数を計算する。

表 8-1 インテル® IPP 音声認識関数 （続き）

関数の基本名	操作
Delta	ベースとなる特徴をコピーし、特徴ベクトルの導関数を計算する。
DeltaDelta	ベースとなる特徴をコピーし、1 次導関数と 2 次導関数を計算する。
ピッチ超解像度関数	
CrossCorrCoeffDecim	デシメーションを使用して、相互相関係数のベクトルを計算する。
CrossCorrCoeff	相互相関係数を計算する。
CrossCorrCoeffInterpolation	補間された相互相関係数を計算する。
モデル評価関数	
AddNRows	N 個のベクトルをベクトル配列から追加する。
ScaleLM	しきい値演算を使用してベクトル要素をスケーリングする。
LogAdd	対数表現の 2 つのベクトルを加算する。
LogSub	対数表現の 2 つのベクトルの差を求める。
LogSum	対数表現のベクトル要素の和を求める。
MahDistSingle	単一の観測ベクトルについてマハラノビス距離を計算する。
MahDist	複数の観測ベクトルについてマハラノビス距離を計算する。
MahDistMultiMix	複数の平均値と分散値についてマハラノビス距離を計算する。
LogGaussSingle	単一のガウスと 1 つの観測ベクトルについて観測確率を計算する。
LogGauss	単一のガウスと複数の観測ベクトルについて観測確率を計算する。
LogGaussMultiMix	複数のガウス混合成分について観測確率を計算する。
LogGaussMax	最大値演算を使用して、複数の観測ベクトルと 1 つのガウス混合成分に基づいて尤度確率を計算する。
LogGaussMaxMultiMix	最大値演算を使用して、複数のガウス混合成分について尤度確率を計算する。
LogGaussAdd	複数の観測ベクトルについて尤度確率を計算する。
LogGaussAddMultiMix	複数のガウス混合成分について尤度確率を計算する。
LogGaussMixture	特定のガウス混合について尤度確率を計算する。
LogGaussMixtureSelect	ガウス分布選択を使用して、ガウス混合について尤度確率を計算する。
BuildSignTable	ガウス混合計算の符号テーブルを埋める。
FillShortlist_Row	ガウス分布選択の行方向ショートリスト・テーブルを埋める。
FillShortlist_Column	ガウス分布選択の列方向ショートリスト・テーブルを埋める。
DTW	動的タイム・ワープ・アルゴリズムを使用して、観測ベクトルのシーケンスと基準ベクトルのシーケンスの間の距離を計算する。

表 8-1 インテル® IPP 音声認識関数 （続き）

関数の基本名	操作
モデル推定関数	
MeanColumn	列要素の平均値を計算する。
VarColumn	列要素の分散値を計算する。
MeanVarColumn	行列の列要素の平均値および分散値を計算する。
WeightedMeanColumn	列要素の加重平均値を計算する。
WeightedVarColumn	列要素の加重分散値を計算する。
WeightedMeanVarColumn	列要素の加重平均値と加重分散値を計算する。
NormalizeColumn	列の平均値と分散値に基づいて、行列の列を正規化する。
NormalizeInRange	入力ベクトルの要素を正規化し、スケーリングする。
MeanVarAcc	平均と分散の再評価のために推定値を累算する。
GaussianDist	2 つのガウス間の距離を計算する。
GaussianSplit	単一のガウス成分を、同じ分散を持つ 2 つの成分に分割する。
GaussianMerge	2 つのガウス確率分布関数を結合する。
Entropy	入力ベクトルのエントロピを計算する。
SinC	正弦を引数で割った値を計算する。
ExpNegSqr	引数を 2 乗して符号を反転した値の指数を計算する。
BhatDist	2 つのガウス間の Bhattacharia 距離を計算する。
UpdateMean	EM トレーニング・アルゴリズムの平均ベクトルを更新する。
UpdateVar	EM トレーニング・アルゴリズムの分散ベクトルを更新する。
UpdateWeight	EM トレーニング･アルゴリズムのガウス混合の重みの値を更新する。
UpdateGConst	ガウスの出力確率密度関数の固定定数を更新する。
OutProbPreCalc	ガウス混合の出力確率のうち、観測ベクトルに無関係な部分を事前に計算する。
DcsClustLAccumulate	決定木クラスタリング・アルゴリズムのステート・クラスタの尤度を計算するためのアキュムレータを更新する。
DcsClustLCompute	決定木ステート･クラスタリング･アルゴリズムの HMM ステート・クラスタの尤度を計算する。
モデル適応関数	
AddMulColumn	重み付きの行列の列を別の列に加算する。
AddMulRow	重み付きのベクトルを別のベクトルに加算する。
QRTransColumn	QR 変換を実行する。
DotProdColumn	2 つの行列の列に対して内積を計算する。
MulColumn	行列の列に値を掛ける。
SumColumnAbs	行列の列要素の和の絶対値を計算する。
SumColumnSqr	重み付けされた行列の列の要素の 2 乗和を計算する。
SumRowAbs	ベクトル要素の和の絶対値を計算する。
SumRowSqr	重み付けされたベクトルの要素の 2 乗和を計算する。

intel®

表 8-1 インテル® IPP 音声認識関数 （続き）

関数の基本名	操作
SVD	行列の特異値分解を実行する
WeightedSum	2 つの入力ベクトルの要素の重み付けされた和を計算する。
ベクトル量子化関数	
FormVector	コードブック・エントリから複数のストリームの出力ベクトルを作成する。
CdbkGetSize	コードブックのサイズをバイト単位で計算する。
CdbkInit	コードブックを格納する構造体を初期化する。
CdbkInitAlloc	コードブック構造体を初期化する。
CdbkFree	コードブック構造体を破壊する。
GetCdbkSize	コードブック内のコードベクトルの数を取得する。
GetCodebook	コードブックからコードベクトルを取得する。
VQ	コードブックに基づいて入力ベクトルを量子化する。
SplitVQ	コードブックに基づいてマルチストリーム・ベクトルを量子化する。
VQSingle_Sort, VQSingle_Thresh	コードブックに基づいて入力ベクトルを量子化し、最も近い複数のクラスタを取得する。
FormVectorVQ	インデックスに基づいて、コードブックからマルチストリーム・ベクトルを作成する。
ポリフェーズ・リサンプリング関数	
ResamplePolyphaseInitAlloc	ポリフェーズ・データ・リサンプリングの構造体を初期化する。
ResamplePolyphaseFree	ポリフェーズ・データ・リサンプリングの構造体を解放する。
ResamplePolyphase	ポリフェーズ・フィルタを使用して入力データを再サンプリングする。
アドバンスト・オーローラ関数	
SmoothedPowerSpectrum_Aurora	FFT 出力の平滑化された大きさを計算する。
NoiseSpectrumUpdate_Aurora	ノイズ・スペクトルを更新する。
WienerFilterDesign_Aurora	適応 Wiener（ウィーナ）フィルタの向上した伝達関数を計算する。
MelFBankInitAlloc_Aurora	メル周波数フィルタ・バンク分析を実行するための構造体を初期化する。
TabsCalculation_Aurora	残留信号フィルタのフィルタ係数を計算する。
ResidualFilter_Aurora	ノイズ除去された波形信号を計算する。
WaveProcessing_Aurora	ノイズ・リダクションの後に波形データを処理する。
LowHighFilter_Aurora	ロー・バンドとハイ・バンドのフィルタを計算する。
HighBandCoding_Aurora	高周波帯域のエネルギー値をコードおよびデコードする。
BlindEqualization_Aurora	ケプストラム係数を均等にする。
DeltaDelta_Aurora	ETSI ES 202 050 規格に基づいて、1 次導関数と 2 次導関数を計算する。
VADGetBufSize_Aurora	VAD 決定のメモリ・サイズをクエリする。

表 8-1 インテル® IPP 音声認識関数 （続き）

関数の基本名	操作
VADInit_Aurora	VAD 構造体のサイズを取得する。
VADDecision_Aurora	VAD 決定を行う。
VADFlush_Aurora	ゼロ入力フレームの VAD 決定を行う。
Ephraim-Malah ノイズ・サプレッサ関数	
FilterUpdateEMNS	ノイズ・サプレッション・フィルタの係数を計算する。
FilterUpdateWiener	Wiener（ウィーナ）フィルタの係数を計算する。
GetSizeMCRA	ステート構造体で必要なサイズをバイト単位で計算する。
InitMCRA	IppMCRAState ステート構造体を初期化する。
InitAllocMCRA	メモリを割り当て、IppMCRAState ステート構造体を初期化する。
UpdateNoisePSDMCRA	ノイズのパワー・スペクトルを再推定する。
音響エコー・キャンセラ関数	
FilterAECNLMS	周波数領域内の適応フィルタ出力を計算する。
CoefUpdateAECNLMS	適応フィルタの係数を更新する。
StepSizeUpdateAECNLMS	適応ステップのサイズを計算する。
ControllerGetSizeAEC	AEC コントローラ・ステート構造体のサイズを返す。
ControllerInitAEC	AEC コントローラ・ステート構造体を初期化する。
ControllerUpdateAEC	エネルギー・ベースの AEC コントローラを実行する。
音声アクティビティ検出（VAD）関数	
FindPeaks	入力ベクトルのピークを識別する。
PeriodicityLSPE	入力音声フレームの周期性を計算する。
Periodicity	入力ブロックの周期性を計算する。

注： 関数の引数に関する以下の説明では、引数がベクトルまたは配列を示す場合、引数の説明の後に示す大カッコ内の式によって、そのベクトルまたは配列の次元を指定する。
本章では、他の章とは異なり、特に断らない限り、ステップ値を示す引数は、対応する配列の要素単位で測られる。

基本算術

この項で説明する関数は、ベクトルと行列の一般的な算術演算を実行する。

intel®

AddAllRowSum

行列の列ベクトルの和を計算し、1 つのベクトルに加算する。

```
IppStatus ippsAddAllRowSum_32f_D2(const Ipp32f* pSrc, int step,
    int height, Ipp32f* pSrcDst, int width);
IppStatus ippsAddAllRowSum_32f_D2L(const Ipp32f** mSrc, int height,
    Ipp32f* pSrcDst, int width);
```

引数

pSrc	入力ベクトル [*height***step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
step	*pSrc* の行のステップ。
height	入力行列 *mSrc* の行数。
pSrcDst	出力ベクトル [*width*] へのポインタ。
width	入力行列 *mSrc* の列数、および出力ベクトル *pSrcDst* の長さ。

説明

関数 ippsAddAllRowSum は、ippsr.h ファイルで宣言される。この関数は、入力行列の列ベクトルの和を計算し、出力ベクトルに加算する。次のように演算する。

D2 サフィックスが付いた関数の場合、

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} pSrc[i \cdot step + j]\ ,\ 0 \le j < width$$

D2L サフィックスが付いた関数の場合、

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} mSrc[i][j],\ 0 \le j < width$$

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*height* または *width* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

SumColumn

行列の列ベクトルの和を計算する。

```
IppStatus ippsSumColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    int height, Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumColumn_16s32f_D2(const Ipp16s* pSrc, int step,
    int height, Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2(const Ipp32f* pSrc, int step,
    int height, Ipp32f* pDst, int width);
IppStatus ippsSumColumn_64f_D2(const Ipp64f* pSrc, int step,
    int height, Ipp64f* pDst, int width);
IppStatus ippsSumColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height,
    Ipp32s* pDst, int width, int scaleFactor);
IppStatus ippsSumColumn_16s32f_D2L(const Ipp16s** mSrc, int height,
    Ipp32f* pDst, int width);
IppStatus ippsSumColumn_32f_D2L(const Ipp32f** mSrc, int height,
    Ipp32f* pDst, int width);
IppStatus ippsSumColumn_64f_D2L(const Ipp64f** mSrc, int height,
    Ipp64f* pDst, int width);
```

引数

pSrc	入力ベクトル [*height***step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
step	入力ベクトル *pSrc* の行のステップ。
height	入力行列 *mSrc* の行数。
pDst	出力ベクトル [*width*] へのポインタ。
width	入力行列 *mSrc* の列数、および出力ベクトル *pDst* の長さ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

intel®

説明

関数 ippsSumColumn は、ippsr.h ファイルで宣言される。この関数は、入力行列の列ベクトルの和を計算し、それらを出力ベクトルに格納する。次のように演算する。

D2 サフィックスが付いた関数の場合、

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i \cdot step + j]\ ,\ 0 \le j < width$$

D2L サフィックスが付いた関数の場合、

$$pDst[j] = \sum_{i=0}^{height-1} mSrc[i][j],\ 0 \le j < width$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、または *pDst* が NULL。
ippStsSizeErr	エラー。*height* または *width* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

SumRow

行列の列ベクトルの和を計算する。

```
IppStatus ippsSumRow_16s32s_D2Sfs(const Ipp16s * pSrc, int width,
    int step, Ipp32s* pDst, int height, int scaleFactor);
IppStatus ippsSumRow_16s32f_D2(const Ipp16s* pSrc, int width,
    int step, Ipp32f* pDst, int height);
IppStatus ippsSumRow_32f_D2(const Ipp32f* pSrc, int width, int step,
    Ipp32f* pDst, int height);
IppStatus ippsSumRow_64f_D2(const Ipp64f* pSrc, int width, int step,
    Ipp64f* pDst, int height);
IppStatus ippsSumRow_16s32s_D2LSfs(const Ipp16s** mSrc, int width,
    Ipp32s* pDst, int height, int scaleFactor);
```

```
IppStatus ippsSumRow_16s32f_D2L(const Ipp16s** mSrc, int width, Ipp32f*
    pDst, int height);
IppStatus ippsSumRow_32f_D2L(const Ipp32f** mSrc, int width,
    Ipp32f* pDst, int height);
IppStatus ippsSumRow_64f_D2L(const Ipp64f** mSrc, int width,
    Ipp64f* pDst, int height);
```

引数

pSrc	入力ベクトル [*height***step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
height	入力行列 *mSrc* の行数、および出力ベクトル *pDst* の長さ。
step	入力ベクトル *pSrc* の行のステップ。
pDst	出力ベクトル [*height*] へのポインタ。
width	入力行列 *mSrc* の列数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsSumRow は、ippsr.h ファイルで宣言される。この関数は、入力行列ベクトル *mSrc* の和を計算し、それらの合計を出力ベクトル *pDst* に格納する。次のように演算する。

D2 サフィックスが付いた関数の場合、

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[i \cdot step + j]\ ,\ \ 0 \le i < height$$

D2L サフィックスが付いた関数の場合、

$$pDst[i] = \sum_{j=0}^{width-1} mSrc[i][j],\ \ 0 \le i < height$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、または *pDst* が NULL。
ippStsSizeErr	エラー。*height* または *width* がゼロ以下。

ippStsStrideErr	エラー。*step* が *width* より小さい。

SubRow

すべての行列の行からベクトルを引く。

```
IppStatus ippsSubRow_16s_D2(const Ipp16s* pSrc, int width, Ipp16s*
    pSrcDst, int dstStep, int height);
IppStatus ippsSubRow_32f_D2(const Ipp32f* pSrc, int width, Ipp32f*
    pSrcDst, int dstStep, int height);
IppStatus ippsSubRow_16s_D2L(const Ipp16s* pSrc, Ipp16s** mSrcDst,
    int width, int height);
IppStatus ippsSubRow_32f_D2L(const Ipp32f* pSrc, Ipp32f** mSrcDst,
    int width, int height);
```

引数

pSrc	入力ベクトル [*width*] へのポインタ。
pSrcDst	ソースおよびデスティネーション行列 [*height***dstStep*] へのポインタ。
mSrcDst	ソースおよびデスティネーション・ベクトル [*height*][*width*] へのポインタ。
width	行列 *mSrcDst* の列数。
dstStep	ベクトル *pSrcDst* の行のステップ。
height	行列 *mSrcDst* の行数。

説明

関数 ippsSubRow は、ippsr.h ファイルで宣言される。この関数は、すべての行列の行から入力ベクトル *pSrc* を引く。次のように演算する。

D2 サフィックスが付いた関数の場合、

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] - pSrc[j] ,$$

$$0 \le i < height,\ 0 \le j < width$$

D2L サフィックスが付いた関数の場合、

$$mSrcDst[i][j] = mSrcDst[i][j] - pSrc[j],$$

$$0 \le i < height,\ 0 \le j < width$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pSrcDst*、または *mSrcDst* が NULL。
ippStsSizeErr	エラー。*height* または *width* がゼロ以下。
ippStsStrideErr	エラー。*dstStep* が *width* より小さい。

CopyColumn_Indirect

列をリダイレクトして、入力行列をコピーする。

```
IppStatus ippsCopyColumn_Indirect_16s_D2(const Ipp16s* pSrc, int
    srcLen, int srcStep, Ipp16s* pDst, const Ipp32s* pIndx, int dstLen,
    int dstStep, int height);
IppStatus ippsCopyColumn_Indirect_32f_D2(const Ipp32f* pSrc, int
    srcLen, int srcStep, Ipp32f* pDst, const Ipp32s* pIndx, int dstLen,
    int dstStep, int height);
IppStatus ippsCopyColumn_Indirect_16s_D2L(const Ipp16s** mSrc, int
    srcLen, Ipp16s** mDst, const Ipp32s* pIndx, int dstLen, int
    height);
IppStatus ippsCopyColumn_Indirect_32f_D2L(const Ipp32f** mSrc, int
    srcLen, Ipp32f** mDst, const Ipp32s* pIndx, int dstLen, int
    height);
```

引数

pSrc	入力ベクトル [*height**srcStep*] へのポインタ。
mSrc	入力行列 [*height*][*srcLen*] へのポインタ。
srcLen	入力行列 *mSrc* の列数。
srcStep	ベクトル *pSrc* の行のステップ。
pDst	出力ベクトル [*height**dstStep*] へのポインタ。
mDst	出力行列 [*height*][*dstLen*] へのポインタ。
pIndx	リダイレクション・ベクトル [*dstLen*] へのポインタ。
dstLen	出力行列 *mDst* の列数。
dstStep	ベクトル *pDst* の行のステップ。

height 入力行列と出力行列の行数。

説明

関数 ippsCopyColumn_Indirect は、ippsr.h ファイルで宣言される。この関数は、*pIndx* によって列をリダイレクトして、入力行列 *mSrc* を出力行列 *mDst* にコピーする。次のように演算する。

D2 サフィックスが付いた関数の場合、

$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcStep + pIndx[j]],\ 0 \le i < height,\ 0 \le j < dstLen$

D2L サフィックスが付いた関数の場合、

$mDst[i][j] = mSrc[i][pIndx[j]],\ 0 \le i < height,\ 0 \le j < dstLen$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pDst*、*mDst* または *pIndx* が NULL。
ippStsSizeErr	エラー。*height*、*srcLen* または *dstLen* がゼロ以下。あるいは *pIndx[j]*≥*srcLen* か *pIndx[j]*<0（0≤*j* <*dstLen* の場合）。
ippStsStrideErr	エラー。*srcstep* が *srclen* より小さい、または *dststep* が *dstlen* より小さい。

BlockDMatrixInitAlloc

対称ブロック対角行列を表す構造体を初期化する。

```
IppStatus ippsBlockDMatrixInitAlloc_16s(IppsBlockDMatrix_16s**
    pMatrix, const Ipp16s** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_32f(IppsBlockDMatrix_32f**
    pMatrix, const Ipp32f** mSrc, const int* bSize, int nBlocks);
IppStatus ippsBlockDMatrixInitAlloc_64f(IppsBlockDMatrix_64f**
    pMatrix, const Ipp64f** mSrc, const int* bSize, int nBlocks);
```

引数

pMatrix	作成されるブロック対角行列へのポインタ。
mSrc	行列の行へのポインタのベクトルへのポインタ。
bSize	ブロック・サイズのベクトル [*nBlocks*] へのポインタ。
nBlocks	行列のブロックの数。

説明

関数 `ippsBlocDMatrixInitAlloc` は、`ippsr.h` ファイルで宣言される。この関数は、対称ブロック対角行列を格納する構造体を作成する。ブロックの外側の行列要素は、ゼロと見なされる。ブロック対角行列 A は、次のように定義される。

$$A[K_l + i][K_l + j] = A[K_l + j][K_l + i] = mSrc[K_l + i]\,[j],$$

($i, j = 0 \ldots bSize[l] - 1$ および $l = 0 \ldots nBlocks - 1$ の場合)

ここで、

$$K_l = \sum_{k < l} bSize[k]$$

$mSrc[K_l + i]$ は、行列 A の行のゼロでない部分を指す。

行列のサイズは $K_{nBlocks} \times K_{nBlocks}$ と等しい。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMatrix* または *mSrc* が NULL。
ippStsSizeErr	エラー。*bSize* または *nBlocks* がゼロ以下。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

BlockDMatrixFree

ブロック対角行列の構造体を解放する。

```
IppStatus ippsBlockDMatrixFree_16s(IppsBlockDMatrix_16s* pMatrix);
IppStatus ippsBlockDMatrixFree_32f(IppsBlockDMatrix_32f* pMatrix);
IppStatus ippsBlockDMatrixFree_64f(IppsBlockDMatrix_64f* pMatrix);
```

引数

pMatrix	ブロック対角行列へのポインタ。

説明

関数 ippsBlockDMatrixFree は、ippsr.h ファイルで宣言される。この関数は、ブロック対角行列の構造体を破壊し、その構造体に割り当てられたすべてのメモリを解放する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMatrix* が NULL。

NthMaxElement

ベクトルの N 番目に大きい要素を検索する。

```
IppStatus ippsNthMaxElement_32s(const Ipp32s* pSrc, int len, int N,
    Ipp32s* pRes);
IppStatus ippsNthMaxElement_32f(const Ipp32f* pSrc, int len, int N,
    Ipp32f* pRes);
IppStatus ippsNthMaxElement_64f(const Ipp64f* pSrc, int len, int N,
    Ipp64f* pRes);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
len	入力ベクトル *pSrc* の要素の数。
N	検索する要素の順位。
pRes	*N* 番目に大きい要素の値へのポインタ。

説明

関数 ippsNthMaxElement は、ippsr.h ファイルで宣言される。この関数は、入力ベクトル *pSrc* で *N* 番目に大きい要素を検索する。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pRes* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsBadArgErr	エラー。*N* がゼロより小さい、または *len* 以上。

VecMatMul

ベクトルに行列を掛ける。

```
IppStatus ippsVecMatMul_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
    pMatr, int step, int height, Ipp16s* pDst, int width, int
    scaleFactor);
IppStatus ippsVecMatMul_16s_D2LSfs(const Ipp16s* pSrc, const Ipp16s**
    mMatr, int height, Ipp16s* pDst, int width, int scaleFactor);
IppStatus ippsVecMatMul_32f_D2(const Ipp32f* pSrc, const Ipp32f* pMatr,
    int step, int height, Ipp32f* pDst, int width);
IppStatus ippsVecMatMul_32f_D2L(const Ipp32f* pSrc, const Ipp32f**
    mMatr, int height, Ipp32f* pDst, int width);
```

引数

pSrc	入力ベクトル [*height*] へのポインタ。
pMatr	入力行列ベクトル [*height***step*] へのポインタ。
mMatr	入力行列 [*height*][*width*] へのポインタ。
step	*pMatr* ベクトルの行のステップ（*pMatr* の要素単位)。
width	結果ベクトルと入力行列の行の長さ。
height	入力行列の行数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsVecMatMul は、ippsr.h ファイルで宣言される。この関数は、次のように入力行ベクトルに行列を掛ける。

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot pMatr[i \cdot step + j]$$

上記は、D2 サフィックスが付いた関数の場合である。

intel®

また、

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot mMatr[i][j]$$

上記は、D2L サフィックスが付いた関数の場合である。ここで、j の値は $0 \le j <$ *width* である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pMatr*、*mMatr* または *pDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

MatVecMul

行列にベクトルを掛ける。

```
IppStatus ippsMatVecMul_16s_D2Sfs(const Ipp16s* pMatr, int step, const
    Ipp16s* pSrc, int width, Ipp16s* pDst, int height, int
    scaleFactor);
IppStatus ippsMatVecMul_16s_D2LSfs(const Ipp16s** mMatr, const Ipp16s*
    pSrc, int width, Ipp16s* pDst, int height, int scaleFactor);
IppStatus ippsMatVecMul_32f_D2(const Ipp32f* pMatr, int step, const
    Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
IppStatus ippsMatVecMul_32f_D2L(const Ipp32f** mMatr, const Ipp32f*
    pSrc, int width, Ipp32f* pDst, int height);
```

引数

pSrc	入力ベクトル [*width*] へのポインタ。
pMatr	入力行列ベクトル [*height***step*] へのポインタ。
mMatr	入力行列 [*height*][*width*] へのポインタ。
step	*pMatr* ベクトルの行のステップ（*pMatr* の要素単位）。
width	入力ベクトルと入力行列の行の長さ。
height	入力行列の行数と結果ベクトルの長さ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsMatVecMul は、ippsr.h ファイルで宣言される。この関数は、次のように行列に入力行ベクトルを掛ける。

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot pMatr[i \cdot step + j]$$

上記は、D2 サフィックスが付いた関数の場合である。

また、

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot mMatr[i][j]$$

上記は、D2L サフィックスが付いた関数の場合である。ここで、$0 \le i < height$ である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pMatr*、*mMatr* または *pDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

特徴処理

この項では、生の音声信号を事前に処理する関数について説明する。特徴抽出は、認識プロセスの最初の手順である。音声特徴は、元の音声信号の圧縮された形式である。特徴を抽出すれば、問題の次元を小さくできる。特徴を抽出すると、話者の違いと環境による歪みがある程度正規化される。
この項では、無音検出と音声検出の両方に必要な関数と、音声信号の一般的な分析手法についても説明する。

ZeroMean

入力ベクトルから平均値を引く。

```
IppStatus ippsZeroMean_16s(Ipp16s* pSrcDst, int len);
```

引数

pSrcDst	ソースおよびデスティネーション・ベクトル [*len*] へのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsZeroMean は、ippsr.h ファイルで宣言される。この関数は、ベクトル *pSrcDst* の平均値を計算し、それをベクトル *pSrcDst* から引く。結果の値は、[-32768..32767] の範囲を超える場合、飽和処理される。
次のように演算する。

$0 \le i < len$ の場合、

$$pSrcDst[i] = \max(-32768, \min(32767, pSrcDst[i] - \frac{1}{len}\sum_{j=0}^{len-1} pSrcDst[j]))$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

CompensateOffset

入力信号の DC オフセットを除去する。

```
IppStatus ippsCompensateOffset_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len, Ipp32f* pSrcDst0, Ipp32f dst0, Ipp32f val);
IppStatus ippsCompensateOffset_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
    pSrcDst0, Ipp16s dst0, Ipp32f val);
IppStatus ippsCompensateOffset_32f_I(Ipp32f* pSrcDst, int len, Ipp32f*
    pSrcDst0, Ipp32f dst0, Ipp32f val);
IppStatus ippsCompensateOffsetQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
IppStatus ippsCompensateOffsetQ15_16s_I(Ipp16s* pSrcDst, int len,
    Ipp16s* pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
```

引数

pSrc	ソース・ベクトル [*len*] へのポインタ。
pDst	デスティネーション・ベクトル [*len*] へのポインタ。
pSrcDst	インプレース演算用のソースおよびデスティネーション・ベクトル [*len*] へのポインタ。
pSrcDst0	直前のソース要素へのポインタ。
len	ベクトル内の要素の数。
dst0	直前のデスティネーション要素。
val	オフセット補償用の定数。
valQ15	オフセット補正用の定数。この値は、Q15 の実数値 ($0.0_{Q31} \le valQ15 < 1.0_{Q31}$) である。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsCompensateOffset は、ippsr.h ファイルで宣言される。この関数は、入力信号のオフセットを除去する。デスティネーション・ベクトルは、次のように計算される。

関数 ippsCompensateOffset の場合、

$$pDst[0] = pSrc[0] - pSrcDst[0] + val \cdot dst0$$

$$pDst[i] = pSrc[i] - pSrc[i-1] + val \cdot pDst[i-1],\ 1 \le i \le len-1$$

$$pSrcDst0[0] = pSrc[len-1],$$

関数 ippsCompensateOffset_I の場合、

$$y = pSrcDst[0] - pSrcDst0[0] + val \cdot dst0, x = pSrcDst[0], pSrcDst[0] = y,$$

$$y = pSrcDst[i] - pSrcDst[i-1] + val \cdot x, x = pSrcDst[i], pSrcDst[i] = y,$$

$$1 \le i \le len-1,$$

$$pSrcDst0[0] = x$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、*pSrcDst*、または *pSrcDst0* が NULL。

ippStsSizeErr　　エラー。*len* がゼロ以下。または *valQ15* が範囲外。

SignChangeRate

入力信号のゼロ交差レートをカウントする。

```
IppStatus ippsSignChangeRate_16s(const Ipp16s* pSrc, int len, Ipp32s*
    pRes);
IppStatus ippsSignChangeRate_32f(const Ipp32f* pSrc, int len, Ipp32f*
    pRes);
IppStatus ippsSignChangeRateCount0_16s(const Ipp16s* pSrc, int len,
    Ipp32s* pRes);
IppStatus ippsSignChangeRateCount0_32f(const Ipp32f* pSrc, int len,
    Ipp32f* pRes);
```

引数

pSrc	入力信号 [*len*] へのポインタ。
len	入力信号 *pSrc* の要素の数。
pRes	結果の変数へのポインタ。

説明

関数 ippsSignChangeRate は、ippsr.h ファイルで宣言される。この関数は、入力信号の符号の変化の回数をカウントする。
この関数を使用して、連続的な音声入力内の音声を検出できる。

次のように演算する。

関数 ippsSignChangeRate の場合、

$$pRes[0] = \sum_{i=1}^{len-1} \begin{cases} 1, & if \;\; pSrc[i] \cdot pSrc[i-1] < 0 \\ 0, & otherwise \end{cases}$$

関数 ippsSignChangeRateCount0 の場合、

$$pRes[0] = \frac{1}{2} \sum_{i=1}^{len-1} |sign(pSrc[i]) - sign(pSrc[i-1])| \;\; , sign(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pRes* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

LinearPrediction

入力ベクトルの線形予測分析を実行する。

```
IppStatus ippsLinearPrediction_Auto_16s_Sfs(const Ipp16s* pSrc, int
    lenSrc, Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsLinearPrediction_Auto_32f(const Ipp32f* pSrc, int lenSrc,
    Ipp32f* pDst, int lenDst);
IppStatus ippsLinearPredictionNeg_Auto_16s_Sfs(const Ipp16s* pSrc, int
    lenSrc, Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsLinearPredictionNeg_Auto_32f(const Ipp32f* pSrc, int
    lenSrc, Ipp32f* pDst, int lenDst);
IppStatus ippsLinearPrediction_Cov_16s_Sfs(const Ipp16s* pSrc, int
    lenSrc, Ipp16s* pDst, int lenDst, int scaleFactor);
IppStatus ippsLinearPrediction_Cov_32f(const Ipp32f* pSrc, int lenSrc,
    Ipp32f* pDst, int lenDst);
```

引数

pSrc	入力ベクトル [*lenSrc*] へのポインタ。
lenSrc	入力ベクトル *pSrc* の長さ。
pDst	出力 LPC 係数ベクトル [*lenDst*] へのポインタ。
lenDst	出力ベクトル *pDst* の長さ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsLinearPrediction は、ippsr.h ファイルで宣言される。この関数は、入力信号の線形予測分析を実行する。
Auto サフィックスが付いた関数の場合、LPC 係数は、次の自己相関手法の式を解くことによって計算される。

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = r[k] \quad , k = 1,\ldots lenDst \, ,$$

ここで、$r[k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k]$ 。

Neg_Auto サフィックスが付いた関数の場合、LPC 係数は、上の式の右辺の符号を反転した式を解けば計算される。

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = -r[k] \quad, k = 1 \ldots lenDst ,$$

Cov サフィックスが付いた関数の場合、LPC 係数は、次の共分散手法の式を解けば計算される。

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot c[i][k] = c[0][k] \quad, k = 1, \ldots lenDst ,$$

ここで、$c[i][k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k-i]$ 。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*lenSrc* または *lenDst* がゼロ以下、あるいは *lenDst* が *lenSrc* 以上。
ippStsNoOperation	LPC 問題の解が存在しない。

Durbin

自己相関の入力ベクトルに Durbin の再帰を実行する。

```
IppStatus ippsDurbin_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp32f* pErr, int scaleFactor);
```

```
IppStatus ippsDurbin_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f* pErr);
```

引数

pSrc	入力ベクトル [*len+1*] へのポインタ。
pDst	出力 LPC 係数ベクトル [*len*] へのポインタ。
len	出力ベクトルの長さ。
pErr	結果の予測エラーへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDurbin は、ippsr.h ファイルで宣言される。この関数は、次のように、入力自己相関ベクトルに Durbin の再帰を実行し、線形予測係数と予測誤差を計算する。

1）初期化

$$m = len,\ R_j = pSrc[j],\ j = 0,\ldots m,\ E^0 = R_0$$

2）反復、$i = 1,\ldots m$ の場合

$$k_i = \left(R_i - \sum_{j=1}^{i-1} y_j^{i-1} \cdot R_{i-j}\right) / E^{i-1}\quad,\quad E^i = (1 - k_i^2) \cdot E^{i-1}\quad,$$

$$y_i^i = k_i\ ,\quad y_j^i = y_j^{i-1} - k_i \cdot y_{i-j}^{i-1}\ ,\ j = 1,\ldots\ i - 1$$

3）結果

$$pDst[i-1] = y_i^m\ ,\ i = 1,\ldots m,\quad pErr[0] = E^m$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pErr* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsNoOperation	LPC 問題の解が存在しない（$E^i \le 0$）。

Schur

Schur アルゴリズムを使用して反射係数を計算する。

```
IppStatus ippsSchur_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
    Ipp32f* pErr, int scaleFactor);
IppStatus ippsSchur_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f* pErr);
```

引数

pSrc	入力自己相関ベクトル [*len+1*] へのポインタ。
pDst	出力反射係数ベクトル [*len*] へのポインタ。
len	出力ベクトルの長さ。
pErr	結果の予測エラーへのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsSchur は、ippsr.h ファイルで宣言される。この関数は、Schur アルゴリズムを使用して、次の手順で反射係数を計算する。

1）初期化

$j = 1,\ldots len-1$、$a_{len}^0 = pSrc[len]$ の場合、

$b_1^0 = pSrc[0]$, $a_j^0 = b_{j+1}^0 = pSrc[j]$,

2）反復、$i = 1,\ldots len$ の場合

$$k_i = -a_i^{i-1}/b_i^{i-1}$$

$$a_j^i = a_j^{i-1} + k_i b_j^{i-1}, \quad b_j^i = b_{j-1}^{i-1} + k_i a_{j-1}^{i-1}, \quad j = i+1,\ldots len$$

3）結果

$$pDst[i-1] = k_i, \quad i = 1,\ldots len$$

$$pErr[0] = b_{len}^{len-1} + k_{len} \cdot a_{len}^{len-1}$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pErr* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsNoOperation	LPC 問題の解が存在しない（$b_i^i = 0$）。

LPToSpectrum

平滑化された大きさスペクトルを計算する。

```
IppStatus ippsLPToSpectrum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s*
    pDst, int order, Ipp32s val, int scaleFactor);
IppStatus ippsLPToSpectrum_32f(const Ipp32f* pSrc, int len, Ipp32f*
    pDst, int order, Ipp32f val);
```

引数

pSrc	入力 LPC 係数ベクトル [*len*] へのポインタ。
pDst	出力 LP スペクトル係数ベクトル [2^{order}] へのポインタ。
len	LPC 係数の数。
order	スペクトル計算用の FFT の次数。
val	スペクトルに加算される値。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsLPToSpectrum は、ippsr.h ファイルで宣言される。この関数は、線形予測の大きさスペクトルの前半を次のように計算する。

$$pDst[k] = \frac{1}{\left| val - \sum_{i=1}^{len} pSrc[i-1] \cdot e^{-\frac{jik\pi}{N}} \right|}, \quad k = 0,...N-1, \quad N = 2^{order}$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。

ippStsSizeErr	エラー。*len* がゼロ以下または $2^{order+1}$ 以上。
ippStsFftOrderErr	エラー。*order* の値が正しくない。
ippStsDivByZero	警告。除数ベクトルの要素の値がゼロ。演算の実行は中止されない。デスティネーション・ベクトルの要素の値は、浮動小数点演算の場合は +Inf に設定され、整数演算の場合は IPP_MAX_16S に設定される。

LPToCepstrum

線形予測係数からケプストラム係数を計算する。

```
IppStatus ippsLPToCepstrum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsLPToCepstrum_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len);
```

引数

pSrc	線形予測係数 [*len*] へのポインタ。
pDst	ケプストラム係数 [*len*] へのポインタ。
len	ソースおよびデスティネーション・ベクトルの要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsLPToCepstrum は、ippsr.h ファイルで宣言される。この関数は、次の式に従って、線形予測係数からケプストラム係数を計算する。

$$pDst[k] = -\left(pSrc[k] + \frac{1}{k+1}\sum_{i=1}^{k}(k-i+1)\cdot pSrc[i-1]\cdot pDst[k-i]\right),$$

$k = 0 \ldots len-1$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。

ippStsSizeErr	エラー。*len* がゼロ以下。

CepstrumToLP

ケプストラム係数から線形予測係数を
計算する。

```
IppStatus ippsCepstrumToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsCepstrumToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len);
```

引数

pSrc	ケプストラム係数 [*len*] へのポインタ。
pDst	線形予測係数 [*len*] へのポインタ。
len	ソースおよびデスティネーション・ベクトルの要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsCepstrumToLP は、ippsr.h ファイルで宣言される。この関数は、次の式に従って、ケプストラム係数から線形予測係数を計算する。

$$pDst[k] = -\left(pSrc[k] + \frac{1}{k+1}\sum_{i=1}^{k}(k-i+1)\cdot pSrc[k-i]\cdot pDst[i-1]\right) \quad , k = 0...len-1$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

LPToReflection

線形予測係数から線形予測反射係数を
計算する。

```
IppStatus ippsLPToReflection_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsLPToReflection_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len);
```

引数

pSrc	線形予測係数 [*len*] へのポインタ。
pDst	線形予測反射係数 [*len*] へのポインタ。
len	ソースおよびデスティネーション・ベクトルの要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 `ippsLPToReflection` は、`ippsr.h` ファイルで宣言される。この関数は、次の式に従って、線形予測反射係数を計算する。

1）初期化

$$n = len,\quad a_i^n = pSrc[i-1],\quad i = 1,\ldots n$$

2) 反復 $i = m,\ldots 1$ の場合

$$k_i = a_i^i,\quad a_j^{i-1} = \frac{a_j^i - a_i^i \cdot a_{i-j}^i}{1 - k_i^2},\quad j = 1,\ldots i\text{-}1$$

3）結果

$$pDst[i-1] = k_i,\quad i = 1,\ldots n$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrc* または *pDst* が NULL。
`ippStsSizeErr`	エラー。*len* がゼロ以下。
`ippStsNoOperation`	反射係数を計算できない（すなわち、いずれかの反復で $\|k_i\| = 1$）。

ReflectionToLP

線形予測反射係数から線形予測係数を
計算する。

```
IppStatus ippsReflectionToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsReflectionToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len);
```

引数

pSrc	線形予測反射係数 [*len*] へのポインタ。
pDst	線形予測係数 [*len*] へのポインタ。
len	ソースおよびデスティネーション・ベクトルの要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsReflectionToLP は、ippsr.h ファイルで宣言される。この関数は、次の式に従って、線形予測係数から線形予測反射係数を計算する。

1）初期化

$$n = len\,,\quad k_i = pSrc[i-1]\,,\quad i = 1,...n$$

2) 反復 $i = 1,...m$ の場合

$$a_i^i = k_i\,,\quad a_j^i = a_j^{i-1} - k_i \cdot a_{i-j}^{i-1}\,,\ j = 1,...k\text{-}1$$

3）結果

$$pDst[i-1] = a_i^n\,,\quad i = 1,...n$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

ReflectionToAR

反射係数を面積比に変換する。

```
IppStatus ippsReflectionToAR_16s_Sfs(const Ipp16s* pSrc, int
    srcShiftVal, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToAR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len);
IppStatus ippsReflectionToLAR_16s_Sfs(const Ipp16s* pSrc, int
    srcShiftVal, Ipp16s* pDst, int len, Ipp32f val, int scaleFactor);
IppStatus ippsReflectionToLAR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len, Ipp32f val);
IppStatus ippsReflectionToTrueAR_16s_Sfs(const Ipp16s* pSrc, int
    srcShiftVal, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToTrueAR_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pDst	デスティネーション・ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。
val	しきい値（1>*val*>0）。
srcShiftVal	ippsReflectionToLAR 関数でのみ使用されるスケール係数。第 2 章の「整数のスケーリング」を参照。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

これらの関数は、ippsr.h ファイルで宣言される。

関数 ippsReflectionToAR は、次の式を使用して、おおよその面積比を計算する。

$$pDst[k] = \frac{1 - pSrc[k]}{1 + pSrc[k]}, \quad k = 0, \ldots len-1$$

関数 ippsReflectionToLAR は、次の式から得られる面積比の対数を計算する。

$$pDst[k] = \begin{cases} \ln\frac{1-val}{1+val}, & if\ pSrc[k] \ge val \\ \ln\frac{1-pSrc[k]}{1+pSrc[k]}, & if\ val > pSrc[k] > -val\ ,\ k = 0,\ldots len-1 \\ \ln\frac{1+val}{1-val}, & if\ -val \ge pSrc[k] \end{cases}$$

関数 ippsReflectionToTrueAR は、次の式を使用して面積比を計算する。

$$pDst[0] = \frac{1-pSrc[0]}{1+pSrc[0]},\quad pDst[k] = pDst[k-1]\cdot\frac{1-pSrc[k]}{1+pSrc[k]}\ ,\ k = 1,\ldots len-1$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsDivByZero	警告。除数ベクトルの要素の値がゼロ。演算の実行は中止されない。浮動小数点演算の場合、デスティネーション・ベクトルの要素の値は、次のように設定される。 NaN　被除数ベクトルの要素の値がゼロの場合。 +Inf　被除数ベクトルの要素の値が正の場合。 -Inf　被除数ベクトルの要素が負の場合。 整数演算の場合、デスティネーション・ベクトルの要素の値は、次のように設定される。 IPP_MAX_16S　被除数ベクトルの要素が正の場合。 IPP_MIN_16S　被除数ベクトルの要素が負の場合。
ippStsRangeErr	エラー。条件 1>*val*>0 を満たしていない。

ReflectionToTilt

ライズ / フォール / 接続パラメータのティルトを計算する。

```
IppStatus ippsReflectionToAbsTilt_16s_Sfs(const Ipp16s* pSrc1, const
    Ipp16s* pSrc2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToAbsTilt_32f(const Ipp32f* pSrc1, const
    Ipp32f* pSrc2, Ipp32f* pDst, int len);
```

```
IppStatus ippsReflectionToTilt_16s_Sfs(const Ipp16s* pSrc1, const
    Ipp16s* pSrc2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsReflectionToTilt_32f(const Ipp32f* pSrc1, const Ipp32f*
    pSrc2, Ipp32f* pDst, int len);
```

引数

pSrc1	1 番目の入力ベクトル [*len*] へのポインタ。
pSrc2	2 番目の入力ベクトル [*len*] へのポインタ。
pDst	デスティネーション・ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

これらの関数は、ippsr.h ファイルで宣言される。

関数 ippsReflectionToAbsTilt は、ライズ係数とフォール係数を、次の式から得られるティルトの絶対値に変換する。

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{|pSrc1[k]| + |pSrc2[k]|},\ k = 0,\ldots len-1$$

関数 ippsReflectionToTilt は、ライズ係数とフォール係数をティルトに変換する。

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{pSrc1[k] + pSrc2[k]},\ k = 0,\ldots len-1$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsDivByZero	警告。除数ベクトルの要素の値がゼロ。演算の実行は中止されない。浮動小数点演算の場合、デスティネーション・ベクトルの要素の値は、次のように設定される。

NaN	被除数ベクトルの要素の値がゼロの場合。
+Inf	被除数ベクトルの要素の値が正の場合。
-Inf	被除数ベクトルの要素が負の場合。

整数演算の場合、デスティネーション・ベクトルの要素の値は、次のように設定される。

IPP_MAX_16S	被除数ベクトルの要素が正の場合。
IPP_MIN_16S	被除数ベクトルの要素が負の場合。

PitchmarkToF0

ティルトに対してライズとフォールの
大きさと持続時間を計算する。

```
IppStatus ippsPitchmarkToF0Cand_16s_Sfs(const Ipp16s* pSrc, Ipp16s*
    pDst, int len, int scaleFactor);
IppStatus ippsPitchmarkToF0Cand_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pDst	デスティネーション・ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 `ippsPitchmarkToF0Cand` は、`ippsr.h` ファイルで宣言される。この関数は、次の式を使用して、ピッチマークから `F0` 候補値を計算する。

$$pDst[0] = \frac{1}{pSrc[0]},\ pDst[k] = \frac{1}{pSrc[k] - pSrc[k-1]},\ k = 1,...len-1$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrc* または *pDst* が NULL。
`ippStsSizeErr`	エラー。*len* がゼロ以下。
`ippStsDivByZero`	警告。除数ベクトルの要素の値がゼロ。演算の実行は中止されない。浮動小数点演算の場合、デスティネーション・ベクトルの要素の値は `+Inf` に設定される。整数演算の場合、デスティネーション・ベクトルの要素の値は `IPP_MAX_16S` に設定される。

UnitCurve

ライズ係数とフォール係数に対して
ティルトを計算する。

```
IppStatus ippsUnitCurve_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsUnitCurve_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsUnitCurve_16s_ISfs(Ipp16s* pSrcDst, int srcShiftVal, int
    len, int scaleFactor);
IppStatus ippsUnitCurve_32f_I(Ipp32f* pSrcDst, int len);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pSrcDst	入力およびデスティネーション・ベクトル [*len*] へのポインタ。
pDst	デスティネーション・ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。
srcShiftVal	第 2 章の「整数のスケーリング」を参照。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsUnitCurve は、ippsr.h ファイルで宣言される。この関数は、次の式に従って、デスティネーション・ベクトルを計算する。

$k = 0,...len-1$ の場合、
$$pDst[k] = \begin{cases} 0\ , & if\ \ pSrc[k] < 0 \\ pSrc[k]^2\ , & if\ \ 0 \le pSrc[k] < 1 \\ 2-(2-pSrc[k])^2\ , & if\ \ 1 \le pSrc[k] \le 2 \\ 2\ , & if\ \ 2 < pSrc[k] \end{cases}$$

インプレース関数型は、ソース・ベクトルをデスティネーション・ベクトルで上書きする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pSrcDst* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

LPToLSP

線形予測係数から線スペクトル対ベクトル
を計算する。

```
IppStatus ippsLPToLSP_16s_Sfs(const Ipp16s* pSrcLP, int srcShiftVal,
    Ipp16s* pDstLSP, int len, int* nRoots, int nInt, int nDiv, int
    scaleFactor);
IppStatus ippsLPToLSP_32f(const Ipp32f* pSrcLP, Ipp32f* pDstLSP, int
    len, int* nRoots, int nInt, int nDiv);
```

引数

pSrcLP	入力 LP 係数ベクトル [*len*] へのポインタ。
pDstLSP	出力 LSP 係数ベクトル [*len*] へのポインタ。
len	LP 係数の数。
nRoots	検出された LSP 値の数へのポインタ。
nInt	ゼロ交差検索の区間の数。
nDiv	根の検索中の間隔二分割の回数。
srcShiftVal	*pSrcLP* 値のスケール係数。
scaleFactor	*pDstLSP* 値のスケール係数。第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsLPToLSP は、ippsr.h ファイルで宣言される。この関数は、線形予測 (LP) 係数を線スペクトル対 (LSP) 表現に変換し、音声符号化でよく使用されるアルゴリズムを実行する。

$a_k = pSrcLP[k]$, $k = 0,...len-1$ (または、ippsLPToLSP_16s_Sfs 関数の場合は $a_k = pSrcLP[k]\cdot 2^{-srcShiftVal}$) を LP 係数とすると (ippsLinearPredictionNeg_Auto 関数で使用される LP 係数の計算式も参照)、

$$A(z) = 1 - \sum_{k=0}^{len-1} a_k z^{-k}$$

LSP 係数は、次の形式の対称多項式および反対称多項式の根として定義される。

$$F_1(z) = A(z) + z^{-len-1}\cdot A(z^{-1}) \ , \quad F_2(z) = A(z) - z^{-len-1}\cdot A(z^{-1}) \ ,$$

ただし、1 および -1 と等しい根は例外である。これらは、次の多項式 $G_1(z)$ および $G_2(z)$ の定義によって除外される。

$G_1(z) = F_1(z)/(1+z^{-1})$ ， $G_2(z) = F_2(z)/(1-z^{-1})$ ， *len* が偶数の場合

$G_1(z) = F_1(z)$ ， $G_2(z) = F_2(z)/(1-z^{-2})$ ， *len* が奇数の場合

これらの多項式は対称多項式であるため、次の式が成り立ち、

$$G_i(z) = g_{i,m_i} \cdot z^{-m_i} + \sum_{k=0}^{m_i-1} g_{i,k} \cdot \left(z^{-k} + z^{-(2m_i-k)}\right) \quad ,$$

次の形式で表現できる。

$$G_i(e^{jw}) = e^{-jwm_i} \cdot G_i'(w) = g_{i,m_i} + 2 \cdot \sum_{k=0}^{m_i-1} g_{i,k} \cdot T_{m_i-k}(x) \quad , i = 1,2 \text{ の場合、}$$

ここで、$x = \cos w$ であり、$T_k(x) = \cos kw$ は m 次の Chebyshev 多項式である（[Kab86] を参照）。

len が偶数の場合、多項式の次数は、$m_1 = m_2 = len/2$ によって得られる。また、

$k = 1,..m_1$ の場合、
$g_{1,0} = g_{2,0} = 1$ ， $g_{1,k} = a_{k-1} + a_{len-k} - g_{1,k-1}$ ，
$g_{2,k} = a_{k-1} - a_{len-k} + g_{2,k-1}$

len が奇数の場合、$m_1 = (len+1)/2$ 、$m_2 = (len-1)/2$ であり、

$k = 1,..m_1$ の場合、$g_{1,0} = 1$ ， $g_{1,k} = -a_{k-1} - a_{len-k}$

$g_{2,-1} = 0$ 、$g_{2,0} = 1$ 、および $m_2 > 0$ の場合は、
$k = 1,..m_2$ の場合、$g_{2,k} = g_{2,k-2} + a_{len-k} - a_{k-1}$

多項式 $G_1(z)$ および $G_2(z)$ は、0 と p の間の等しい間隔の *nInt* 個の点で多項式の符号の変化をチェックすること（ゼロ交差検索）によって余弦領域内で見つかった単位円上に、インタレースされた異なる根を持つ。この符号の変化の間隔は *nDiv* 回だけ二分割され、その後に線形補間によって根が見つけられる。Chebyshev 多項式を使用して、$G_1(z)$ と $G_2(z)$ を評価する。根は *pDstLSP* ベクトルに昇順で書き込まれる。

見つかった根の数は、*nRoots*[0] に書き込まれる。*len* 個の根が見つからない場合は、エラー・ステータスが返される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcLP*、*pDstLSP* または *nRoots* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下、*nInt* が *len* より小さい、または *nDiv* がゼロより小さい。
ippStsNoRootFoundErr	見つかった根の数が *len* より少ない。

LSPToLP

線スペクトル対ベクトルを線形予測係数に変換する。

```
IppStatus ippsLSPToLP_16s_Sfs(const Ipp16s* pSrcLSP, int srcShiftVal,
    Ipp16s* pDstLP, int len, int scaleFactor);
IppStatus ippsLSPToLP_32f(const Ipp32f* pSrcLSP, Ipp32f* pDstLP, int
    len);
```

引数

pSrcLSP	入力 LSP 係数ベクトル [*len*] へのポインタ。
pDstLP	出力 LP 係数ベクトル [*len*] へのポインタ。
len	LSP 係数および LP 係数の数。
srcShiftVal	*pSrcLSP* 値のスケール係数。
scaleFactor	*pDstLP* 値のスケール係数。第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsLSPToLP は、ippsr.h ファイルで宣言される。この関数は、Kabal 法 ([Kab86] を参照) を使用して、線スペクトル対 (LSP) 値を線形予測 (LP) 係数に変換する。

多項式

$$G_1(e^{-jw}) = e^{-jwm_1} \cdot G_1'(w) \quad \text{および} \quad G_2(e^{-jw}) = e^{-jwm_2} \cdot G_2'(w)$$

は、Chebyshev 表現を使用して、多項式の根 *pSrcLSP*[k]、(k =0,...*len*-1) から再構築される。

intel®

次に、多項式 $F_1(z)$ および $F_2(z)$ が、次のように再構築される。

len が偶数の場合、$F_1(z) = G_1(z) \cdot (1 + z^{-1})$ ， $F_2(z) = G_2(z) \cdot (1 - z^{-1})$

len が奇数の場合、$F_1(z) = G_1(z)$ ， $F_2(z) = G_2(z) \cdot (1 - z^{-2})$

最後に、線形予測係数が、次の式から求められる。

$$A(z) = \frac{F_1(z) + F_2(z)}{2} = 1 - \sum_{k=0}^{len-1} a_k z^{-k}$$

また、次の式が使用される。

$pDstLP[k] = a_k, k = 0,...len-1$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcLSP* または *pDstLP* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsIncorrectLSP	警告。*pSrcLSP* の要素が有効な LSP 値でない(したがって、$-1 < ... < pSrcLSP[k+1] < pSrcLSP[k] < ... < 1$ の条件を満たしていない)。

MelToLinear

メルスケール値をリニアスケール値に変換する。

```
IppStatus ippsMelToLinear_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
    len, Ipp32f melMul, Ipp32f melDiv);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pDst	出力ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。
melMul	メルスケール式の乗算係数。

melDiv	メルスケール式の除算係数。

説明

関数 ippsMelToLinear は、ippsr.h ファイルで宣言される。この関数は、メル周波数スケールをリニア周波数スケールに変換する。

$$pDst[i] = melDiv \cdot \left[\exp\left(\frac{pSrc[i]}{melMul}\right) - 1\right] \ , \ i = 0,...,len\text{-}1$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下、あるいは *melMul* または *melDiv* がゼロ。

LinearToMel

リニアスケール値をメルスケール値に変換する。

```
IppStatus ippsLinearToMel_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f melMul, Ipp32f melDiv);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pDst	出力ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。
melMul	メルスケール式の乗算係数。
melDiv	メルスケール式の除算係数。

説明

関数 ippsLinearToMel は、ippsr.h ファイルで宣言される。この関数は、リニア周波数スケールをメル周波数スケールに変換する。

$$pDst[i] = melMul \cdot \ln\left(1 + \frac{pSrc[i]}{melDiv}\right) \ , \ i = 0,...,len\text{-}1$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。または *melMul* あるいは *melDiv* がゼロ。

CopyWithPadding

ゼロのパディングを使用して、入力信号を出力にコピーする。

```
IppStatus ippsCopyWithPadding_16s(const Ipp16s* pSrc, int lenSrc,
    Ipp16s* pDst, int lenDst);
IppStatus ippsCopyWithPadding_32f(const Ipp32f* pSrc, int lenSrc,
    Ipp32f* pDst, int lenDst);
```

引数

pSrc	入力ベクトル [*lenSrc*] へのポインタ。
pDst	出力ベクトル [*lenDst*] へのポインタ。
lenSrc	入力ベクトル *pSrc* の長さ。
lenDst	出力ベクトル *pDst* の長さ。

説明

関数 ippsCopyWithPadding は、ippsr.h ファイルで宣言される。この関数は、入力ベクトルを出力ベクトルにコピーする。出力ベクトルのサイズの方が大きい場合は、後続の要素はゼロで埋められる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*lenSrc* または *lenDst* がゼロ以下、あるいは *lenDst* が *lenSrc* より小さい。

MelFBankGetSize

メル周波数フィルタ・バンク構造体の
サイズを取得する。

```
IppStatus ippsMelFBankGetSize_32s(int winSize, int nFilter, IppMelMode
    mode, int *pSize);
```

引数

winSize	サンプル単位のフレームの長さ（32 ≤ *winSize* ≤ 8192）。
nFilter	メルスケール・フィルタ・バンク K の数（0 < *nFilter* ≤ *winSize*）。
mode	実行モードを指定するフラグ。現在のところ、IPP_FBANK_FREQWGT だけが使用できる。
pSize	フィルタ・バンク構造体のサイズを格納する変数へのポインタ。

説明

関数 ippsMelFBankGetSize は、ippsr.h ファイルで宣言される。この関数は、メル周波数フィルタ・バンク構造体および関連ストレージに必要なサイズを決定する。この関数は、ippMelFBankInit を呼び出す前とメモリを割り当てる前に呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsFBankFlagErr	エラー。*mode* の値が正しくない。
ippStsNullPtrErr	エラー。ポインタ *pSize* が NULL。
ippStsSizeErr	エラー。*nFilter* が範囲外、または *winSize* がゼロ以下。

MelFBankInit

メル周波数フィルタ・バンク分析を
実行するための構造体を初期化する。

```
IppStatus ippsMelFBankInit_32s(IppsFBankState_32s *pFBank, int
    *pFFTLen, int winSize, Ipp32s sampFreq, Ipp32s lowFreq, Ipp32s
    highFreq, int nFilter, Ipp32s melMulQ15, Ipp32s melDivQ15,
    IppMelMode mode);
```

引数

pFBank	初期化されるメルスケール・フィルタ・バンク構造体へのポインタ。
pFFTLen	フィルタ・バンクの評価に使用される FFT（N = *pFFTLen* [0]）のサイズへのポインタ。
winSize	フレームの長さ（サンプル単位）。
sampFreq	入力信号のサンプリング周波数 f_i（Hz 単位）。
lowFreq	最初のバンドパス・フィルタの開始周波数 f_{low}（Hz 単位）。
highFreq	最後のバンドパス・フィルタの終了周波数 f_{high}（Hz 単位）。
nFilter	メルスケール・フィルタ・バンク K の数。
melMulQ15	メルスケール式の乗算係数。この値は、Q15 の実数値である。
melDivQ15	メルスケール式の除数。この値は、Q15 の実数値である。
mode	実行モードを指定するフラグ。次の値を指定できる。

IPP_FBANK_MELWGT - 関数はメルスケールでフィルタ・バンクの重みを計算する。

IPP_FBANK_FREQWGT - 関数は周波数空間でフィルタ・バンクの重みを計算する。

上の 2 つのフラグのうち 1 つをセットする必要がある。

IPP_POWER_SPECTRUM -フィルタ・バンク分析中に FFT パワー・スペクトルを使用する。

説明

関数 ippsMelFBankInit は、ippsr.h ファイルで宣言される。この関数は、メル周波数フィルタ・バンク分析関数用の三角フィルタ・バンクを初期化する。メルのフィルタ係数は、ippsMelFBankInitAlloc 関数と同じ方法で計算される。構造体のメモリは、事前に割り当てる必要がある。このメモリのサイズは、ippsMelFBankGetSize 関数によって決定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* または *pFFTLen* が NULL。
ippStsSizeErr	エラー。*winSize*、*nFilter*、*sampFreq*、または *lowFreq* がゼロ以下。
ippStsFBankFreqErr	エラー。*highFreq* が *lowFreq* より小さい、または *highFreq* が *sampFreq/2* より大きい。
ippStsFBankFlagErr	エラー。*mode* の値が正しくない。

MelFBankInitAlloc

メル周波数フィルタ・バンク分析を実行するための構造体を初期化し、メモリを割り当てる。

```
IppStatus ippsMelFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f
    highFreq, int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode
    mode);
IppStatus ippsMelFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f
    highFreq, int nFilter, Ipp32f melMul, Ipp32f melDiv, IppMelMode
    mode);
```

引数

pFBank	作成されるメルスケール・フィルタ・バンク構造体へのポインタ。
pFFTLen	フィルタ・バンクの評価に使用される FFT（N = *pFFTLen*[0]）のサイズへのポインタ。
winSize	フレームの長さ（サンプル単位）。

sampFreq	入力信号のサンプリング周波数 f_i（Hz 単位）。
lowFreq	最初のバンドパス・フィルタの開始周波数 f_{low}（Hz 単位）。
highFreq	最後のバンドパス・フィルタの終了周波数 f_{high}（Hz 単位）。
nFilter	メルスケール・フィルタ・バンク K の数。
melMul	メルスケール式の乗算係数。
melDiv	メルスケール式の除数。
mode	実行モードを指定するフラグ。次の値を指定できる。

`IPP_FBANK_MELWGT`	関数はメルスケールでフィルタ・バンクの重みを計算する。
`IPP_FBANK_FREQWGT`	関数は周波数空間でフィルタ・バンクの重みを計算する。

上の 2 つのフラグのうち 1 つをセットする必要がある。

`IPP_POWER_SPECTRUM`	フィルタ・バンク分析中に FFT パワー・スペクトルを使用する。

説明

関数 `ippsMelFBankInitAlloc` は、`ippsr.h` ファイルで宣言される。この関数は、メル周波数フィルタ・バンク分析用の三角フィルタ・バンクを初期化する。メル周波数フィルタ・バンク分析は、メル周波数ケプストラム係数（MFCC）特徴計算の主な手順の 1 つである。

メル周波数変換は、次の式に従って実行される。

$$mel(f) = melMul \cdot \ln\left(1 + \frac{f}{melDiv}\right)$$

各フィルタ・バンクの中心（メルスケールでは c_k、FFT 領域では y_k）は、次のように計算される。

$k = 0...K+1$ の場合、

$$c_k = mel(f_{low}) + \frac{k}{K+1} \cdot [mel(f_{high}) - mel(f_{low})] ,$$

$$y_k = round\left\{\frac{N}{f_s} mel^{-1}(c_k)\right\}$$

round(*x*) は、*x* の値を最も近い整数に丸める（*mode* が IPP_FBANK_FREQWGT の場合）か、*x* 以下の最大の整数に丸める（*mode* が IPP_FBANK_MELWGT の場合）。*N* は、FFT の長さである。

mode が IPP_FBANK_FREQWGT の場合、フィルタ出力は、次の式に従って（関数 ippsEvalFBank によって）計算される。

$$fFank_{k-1} = \sum_{i=y_{k-1}}^{y_k} \frac{i - y_{k-1} + 1}{y_k - y_{k-1} + 1} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{y_{k+1} - i + 1}{y_{k+1} - y_k + 1} \cdot x_i \text{ , } k = 1 \ldots K \quad (8.1)$$

x_i は対応する FFT スペクトルの大きさである。

mode が IPP_FBANK_MELWGT の場合、フィルタ出力は、次の式に従って（関数 ippsEvalFBank によって）計算される。

$$fFank_{k-1} = \sum_{i=y_{k-1}+1}^{y_k} \frac{mel\left(i\frac{f_s}{N}\right) - c_{k-1}}{c_k - c_{k-1}} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{c_{k+1} - mel\left(i\frac{f_s}{N}\right)}{c_{k+1} - c_k} \cdot x_i \text{ , } k = 1 \ldots K \quad (8.2)$$

関数型 ippsMelFBankInitAlloc_32s では、現在のところ IPP_FBANK_MELWGT モードだけを使用できる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* または *pFFTLen* が NULL。
ippStsSizeErr	エラー。*winSize*、*nFilter*、*sampFreq*、または *lowFreq* がゼロ以下。
ippStsFBankFreqErr	エラー。ポインタ *highFreq* が *lowFreq* より小さい、または *highFreq* が *sampFreq/2* より大きい。
ippStsFBankFlagErr	エラー。*mode* の値が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

MelLinFBankInitAlloc

リニア周波数とメル周波数を組み合わせた
フィルタ・バンク分析を実行するための
構造体を初期化する。

```
IppStatus ippsMelLinFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
    Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
    Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
IppStatus ippsMelLinFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
    int* pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq,
    Ipp32f highFreq, int nFilter, Ipp32f highLinFreq, int nLinFilter,
    Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

引数

pFBank	作成されるフィルタ・バンク構造体へのポインタ。
pFFTLen	フィルタ・バンクの評価に使用される FFT（*N* = *pFFTLen*[0]）のサイズへのポインタ。
winSize	フレームの長さ（サンプル単位）。
sampFreq	入力信号のサンプリング周波数 f_i（Hz 単位）。
lowFreq	最初のバンドパス・フィルタの開始周波数 f_{low}（Hz 単位）。
highLinFreq	リニアスケールの最後のバンドパス・フィルタの終了周波数 f_{lin}（Hz 単位）。
highFreq	最後のバンドパス・フィルタの終了周波数 f_{high}（Hz 単位）。
nFilter	メルスケール・フィルタ・バンク *K* の数。
nLinFilter	リニアスケール・フィルタ・バンク *L* の数。
melMul	メルスケール式の乗算係数。
melDiv	メルスケール式の除数。
mode	実行モードを指定するフラグ。次の値を指定できる。

IPP_FBANK_MELWGT	関数はメルスケールでフィルタ・バンクの重みを計算する。
IPP_FBANK_FREQWGT	関数は周波数空間でフィルタ・バンクの重みを計算する。

上の 2 つのフラグのうち 1 つをセットする必要がある。

`IPP_POWER_SPECTRUM`　フィルタ・バンク分析中に FFT パワー・スペクトルを使用する。

説明

関数 ippsMelLinFBankInitAlloc は、ippsr.h ファイルで宣言される。この関数は、リニア周波数およびメル周波数の三角フィルタ・バンクを初期化する。最初の L 個のフィルタは、f_{low} から f_{lin} までの周波数帯にリニアに配置される。各フィルタ・バンクの中心（メルスケールでは c_k、FFT 領域では y_k）は、次のように計算される。

$L = K$ の場合

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L+1} \cdot k \ , \quad y_k = round\left\{\frac{N}{f_s} \cdot z_k\right\} \ , \quad c_k = mel(z_k) \quad , k = 0,...L+1$$

$L < K$ の場合

$$z_k = f_{low} + \frac{f_{lin} - f_{low}}{L} \cdot k \ , \quad y_k = round\left\{\frac{N}{f_s} \cdot z_k\right\} \ , \quad c_k = mel(z_k) \quad , k = 0,...L$$

および

$$c_k = mel(f_{lin}) + \frac{k-L}{K-L+1} \cdot [mel(f_{high}) - mel(f_{lin})] \ ,$$

$$y_k = round\left\{\frac{N}{f_s} \cdot mel^{-1}(c_k)\right\} \ , \ k = L,...K+1$$

$round(x)$ は、x の値を最も近い整数に丸める（*mode* が IPP_FBANK_FREQWGT の場合）か、x 以下の最大の整数に丸める（*mode* が IPP_FBANK_MELWGT の場合）。N は FFT の長さである。

mode が IPP_FBANK_FREQWGT の場合、フィルタ出力は、ippsMelFBankInitAlloc 関数について指定された式 (8.1) に従って（関数 ippsEvalFBank によって）計算される。

mode が IPP_FBANK_MELWGT の場合、フィルタ出力は、式 (8.2) に従って（関数 ippsEvalFBank によって）計算される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* または *pFFTLen* が NULL。

ippStsSizeErr	エラー。*winSize*、*nFilter*、*sampFreq* または *lowFreq* がゼロ以下、*nLinFilter* が 1 以下、あるいは *nLinFilter* が *nFilter* より大きい。
ippStsFBankFreqErr	エラー。*highLinFreq* が *lowFreq* より小さいか、*highFreq* が *highLinFreq* より小さいか、*highFreq* が *sampFreq/2* より大きい。あるいは *(highLinFreq > lowFreq)&&(nLinFilter==0)*、または *(highLinFreq <highFreq)&&(nLinFilter==nFilter)* である。
ippStsFBankFlagErr	エラー。*mode* の値が正しくない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

EmptyFBankInitAlloc

空のフィルタ・バンク構造体を初期化する。

```
IppStatus ippsEmptyFBankInitAlloc_16s(IppsFBankState_16s** pFBank,
    int* pFFTLen, int winSize, int nFilter, IppMelMode mode);
IppStatus ippsEmptyFBankInitAlloc_32f(IppsFBankState_32f** pFBank,
    int* pFFTLen, int winSize, int nFilter, IppMelMode mode);
```

引数

pFBank	作成されるフィルタ・バンク構造体へのポインタ。
pFFTLen	フィルタ・バンクの評価に使用される FFT（*N* = *pFFTLen*[0]）のサイズへのポインタ。
winSize	フレームの長さ（サンプル単位)。
nFilter	フィルタ・バンク *K* の数。
mode	実行モードを指定するフラグ。次の値を指定できる。 IPP_POWER_SPECTRUM　フィルタ・バンク分析中に FFT パワー・スペクトルを使用する。

説明

関数 ippsEmptyFBankInitAlloc は、ippsr.h ファイルで宣言される。この関数は、*nFilter* 個のフィルタのフィルタ・バンク構造体を初期化する。フィルタ・バンクの中心周波数は、関数 ippsFBankSetCenters で設定できる。フィルタ・バンクの中心周波数は、関数 ippsFBankSetCoeffs で設定できる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* または *pFFTLen* が NULL。
ippStsSizeErr	エラー。*winSize*、*nFilter* または *pFBank* がゼロ以下。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

FBankFree

フィルタ・バンク分析用の構造体を破壊する。

```
IppStatus ippsFBankFree_16s(IppsFBankState_16s* pFBank);
IppStatus ippsFBankFree_32f(IppsFBankState_32f* pFBank);
```

引数

pFBank	フィルタ・バンク構造体へのポインタ。

説明

関数 ippsFBankFree は、ippsr.h ファイルで宣言される。この関数は、フィルタ・バンク構造体を破壊し、その構造体に割り当てられたすべてのメモリを解放する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* が NULL。

FBankGetCenters

三角フィルタ・バンクの中心周波数を設定する。

```
IppStatus ippsFBankGetCenters_16s(const IppsFBankState_16s* pFBank,
    int* pCenters);
IppStatus ippsFBankGetCenters_32f(const IppsFBankState_32f* pFBank,
    int* pCenters);
```

引数

pFBank	フィルタ・バンク構造体へのポインタ。
pCenters	中心周波数を格納する出力ベクトルへのポインタ。

説明

関数 ippsGetCenters は、ippsr.h ファイルで宣言される。この関数は、フィルタ・バンクの中心の（FFT 領域点内の）インデックス y_k を取得する。結果の配列 *pCenters* のサイズは *nFilter*+1 である。フィルタ・バンク構造体 *pFBank* は、ippsMelFBankInitAlloc または ippsMelLinFBankInitAlloc 関数によって初期化されていなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* または *pCenters* が NULL。
ippStsFBankErr	エラー。ippsEmptyFBankInitAlloc 関数によるフィルタ・バンクの初期化後のフィルタの中心が有効でない。

FBankSetCenters

三角フィルタ・バンクの中心周波数を設定する。

```
IppStatus ippsFBankSetCenters_16s(IppsFBankState_16s* pFBank, const
    int* pCenters);
IppStatus ippsFBankSetCenters_32s(IppsFBankState_32s* pFBank, const
    int* pCenters);
```

引数

pFBank	フィルタ・バンク構造体へのポインタ。
pCenters	中心周波数を格納する出力ベクトルへのポインタ。

説明

関数 ippsSetCenters は、ippsr.h ファイルで宣言される。この関数は、*pFBank* 内のフィルタ中心インデックス y_k を設定する。
このインデックスは、次の条件を満たしていなければならない。

$0 \le ... \le y_k \le y_{k+1} \le ... \le N/2 , \quad i = 0,...K$

k 番目の中心を変更する場合は、ippsFBankSetCoeffs 関数を使用して、（k-1）番目、k 番目、および（k+1）番目のフィルタ・バンクのフィルタ重み係数も変更しなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* または *pCenters* が NULL。

FBankGetCoeffs

フィルタ・バンクの重み係数を取得する。

```
IppStatus ippsFBankGetCoeffs_16s(const IppsFBankState_16s* pFBank, int
    fIdx, Ipp32f* pCoeffs);
IppStatus ippsFBankGetCoeffs_32f(const IppsFBankState_32f* pFBank, int
    fIdx, Ipp32f* pCoeffs);
```

引数

pFBank	フィルタ・バンク構造体へのポインタ。
fIdx	フィルタのインデックス。
pCoeffs	フィルタ係数を格納する出力ベクトルへのポインタ。

説明

関数 ippsFBankGetCoeffs は、ippsr.h ファイルで宣言される。この関数は、フィルタ・バンク *fIdx* の重み係数を、配列 *pCoeffs* にコピーする。この配列のサイズは（$y_{k+1} - y_{k-1} + 1$）である。

ただし、ippsFBankGetCoeffs_16s 関数型の場合は、フィルタの重み係数の出力は浮動小数点形式で表現される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* または *pCoeffs* が NULL。
ippStsSizeErr	エラー。*fIdx* が 1 より小さい、または *nFilter* より大きい。

ippStsFBankErr	エラー。*fIdx* フィルタ係数が利用できないか、有効でない。

FBankSetCoeffs

フィルタ・バンクの重み係数を設定する。

```
IppStatus ippsFBankSetCoeffs_16s(IppsFBankState_16s* pFBank, int fIdx,
    const Ipp32f* pCoeffs);
IppStatus ippsFBankSetCoeffs_32f(IppsFBankState_32f* pFBank, int fIdx,
    const Ipp32f* pCoeffs);
```

引数

pFBank	フィルタ・バンク構造体へのポインタ。
fIdx	フィルタのインデックス。
pCoeffs	出力係数ベクトルへのポインタ。

説明

関数 ippsFBankGetCoeffs は、ippsr.h ファイルで宣言される。この関数は、フィルタ・バンク *fIdx* の重み係数を設定する。ベクトル *pCoeffs* は、y_{k-1} から y_{k+1} までの重み係数を格納する。

ただし、ippsFBankSetCoeffs_16s 関数型の場合は、重み係数は浮動小数点形式で表現され、[-1,1] の区間に飽和される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* または *pCoeffs* が NULL。
ippStsSizeErr	エラー。*fIdx* が 1 より小さい、または *nFilter* より大きい。
ippStsFBankErr	エラー。重み係数が利用できないか、有効でない。

EvalFBank

フィルタ・バンク分析を実行する。

```
IppStatus ippsEvalFBank_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pDst,
    const IppsFBankState_16s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
    IppsFBankState_32s* pFBank, int scaleFactor);
IppStatus ippsEvalFBank_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFBankState_32f* pFBank);
```

引数

pSrc	ソース ベクトル $[2^{pFFTOrder[0]}]$ へのポインタ。
pDst	フィルタ・バンク係数ベクトル [*nFilter*] へのポインタ。
pFBank	フィルタ・バンク構造体へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 `ippsEvalFBank` は、`ippsr.h` ファイルで宣言される。この関数は、入力ベクトル *pSrc* のフィルタ・バンク分析を実行する。フィルタ・バンク構造体の初期化時に設定された実行モードによって、次のように、入力ベクトルの使用方法が決まる。

`IPP_POWER_SPECTRUM` フラグがセットされている場合は、ソース・ベクトルは波形信号である。フィルタ・バンク分析に使用される入力信号スペクトルの大きさは、次のように計算される。

$$x_k = \left|\left(\sum_{i=0}^{N-1} pSrc[i] \cdot \exp\left(-j \cdot 2\pi \frac{ik}{N}\right)\right)\right| \quad , 0 \le k \le N/2$$

`IPP_POWER_SPECTRUM` フラグがセットされていない場合は、入力ベクトルがフィルタ・バンク分析に直接使用される。

$$x_k = pSrc[i], \quad 0 \le k \le N/2$$

実行モードに基づいて、式 (8.1) または (8.2) を使用してフィルタ・バンク係数を取得する。入力ベクトル *pSrc* は、分析の終了後に削除される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsFBankErr	エラー。*pFBank* 構造体を計算に使用できない。

DCTLifterGetSize_MulC0

DCT 構造体のサイズを取得する。

```
IppStatus ippsDCTLifterGetSize_MulC0_16s(int lenDCT, int lenCeps,
    int *pSize);
```

引数

lenDCT	DCT のサイズ（$0 < lenDCT \leq 8192$）。
lenCeps	C0 を除く出力係数の数（$0 < lenCeps \leq lenDCT$）。
pSize	バイト単位のサイズを格納する変数へのポインタ。

説明

関数 ippsDCTLifterGetSize_MulC0 は、ippsr.h ファイルで宣言される。この関数は、DCT 構造体および関連したストレージに必要なサイズをバイト単位で計算する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSize* が NULL。
ippStsSizeErr	エラー。*lenDCT* または *lenCeps* が範囲外。

DCTLifterInit_MulC0

DCT の実行と DCT 係数のリフタリングに
使用される構造体を初期化する。

```
IppStatus ippsDCTLifterInit_MulC0_16s(IppsDCTLifterState_16s*
    pDCTLifter, int lenDCT, const Ipp32s* pLifterQ15, int lenCeps);
```

引数

pDCTLifter	DCT 計算およびリフタリング用に初期化される構造体へのポインタ。
lenDCT	DCT のサイズ（$0 < lenDCT \le 8192$）。
pLifterQ15	Q15 で表現されたリフタリング係数ベクトルへのポインタ（$0.0_{Q15} \le pLifterQ15[k] < 512.0_{Q15}$）。
lenCeps	C_0 を除く出力係数の数（$0 < lenCeps \le lenDCT$）。

説明

関数 `ippsDCTLifterInit_MulC0` は、`ippsr.h` ファイルで宣言される。この関数は、`ippsDCTLifterInitAlloc_MulC0` 関数と同様に DCT 計算およびリフタリング用の構造体を初期化する。ただし、構造体のメモリは、事前に割り当てる必要がある。このメモリのサイズは `ippsDCTLifterGetSize_MulC0` 関数によって決定される。

C_0 は、出力ベクトルの最後の要素に格納される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pLifterQ15* または *pDCTLifter* が NULL。
`ippStsSizeErr`	エラー。*lenDCT*、*lenCeps*、または *pLifterQ15*[k] が範囲外。

intel®

DCTLifterInitAlloc

DCT の実行と DCT 係数のリフタリングに使用される構造体を初期化し、メモリを割り当てる。

```
IppStatus ippsDCTLifterInitAlloc_16s(IppsDCTLifterState_16s**
    pDCTLifter, int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_16s(IppsDCTLifterState_16s**
    pDCTLifter, int lenDCT, int lenCeps, int nLifter, Ipp32f val,
    Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_16s(IppsDCTLifterState_16s**
    pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_16s(IppsDCTLifterState_16s**
    pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_32f(IppsDCTLifterState_32f**
    pDCTLifter, int lenDCT, int lenCeps, int nLifter, Ipp32f val);
IppStatus ippsDCTLifterInitAlloc_C0_32f(IppsDCTLifterState_32f**
    pDCTLifter, int lenDCT, int lenCeps, int nLifter, Ipp32f val,
    Ipp32f val0);
IppStatus ippsDCTLifterInitAlloc_Mul_32f(IppsDCTLifterState_32f**
    pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);
IppStatus ippsDCTLifterInitAlloc_MulC0_32f(IppsDCTLifterState_32f**
    pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);
```

引数

pDCTLifter	DCT 計算およびリフタリング用に作成される構造体へのポインタ。
lenDCT	DCT のサイズ。
lenCeps	出力係数の数（C_0 を除く）。
nLifter	リフタリング係数。
PLifter	リフタリング係数ベクトルへのポインタ。
val	出力係数（C_0 を除く）のスケール係数
val0	C_0 のスケール係数。

説明

関数 `ippsDCTLifterInitAlloc` は、`ippsr.h` ファイルで宣言される。この関数は、DCT 計算およびリフタリング用の構造体を初期化する。

最初の DCT 係数 C_0 は、通常は無視される。したがって、`ippsDCTLifter` 関数の出力には、C_1 から $C_{lenCeps}$ までの係数だけが含まれる。ただし、C_0 サフィックスを指定した場合は、C_0 は出力ベクトルの最後の要素として格納される。

リフタリング係数は、以下の式に従って計算される。
`Mul` サフィックスも `C0` サフィックスも付かない関数の場合、

$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val \ , \ i = 1,...lenCeps$$

`C0` サフィックスが付いた関数の場合、

$$l_0 = val0$$

$$l_i = \left(1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right)\right) \cdot val \ , \ i = 1,...lenCeps$$

`Mul` サフィックスが付いた関数の場合、

$$l_i = pLifter[i-1] \ , \ i = 1,...lenCeps$$

`Mul` サフィックスと `C0` サフィックスが付いた関数の場合、

$$l_0 = pLifter[lenCeps-1]$$

$$l_i = pLifter[i-1] \ , \ i = 1,...lenCeps$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pLifter* が NULL。
`ippStsSizeErr`	エラー。*lenDCT*、*lenCeps* または *nLifter* がゼロ以下、あるいは *lenDCT* が *lenCeps* より小さい。
`ippStsMemAllocErr`	エラー。メモリが割り当てられていない。

DCTLifterFree

DCT およびリフタリングに使用された構造体を破壊する。

```
IppStatus ippsDCTLifterFree_16s(IppsDCTLifterState_16s* pDCTLifter);
IppStatus ippsDCTLifterFree_32f(IppsDCTLifterState_32f* pDCTLifter);
```

引数

pDCTLifter	DCT/ リフタリング構造体へのポインタ。

説明

関数 ippsDCTLifterFree は、ippsr.h ファイルで宣言される。この関数は、DCT/ リフタリング構造体に割り当てられたすべてのメモリを解放して、DCT/ リフタリング構造体をクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDCTLifter* が NULL。

DCTLifter

DCT を実行し、DCT 係数をリフタリングする。

```
IppStatus ippsDCTLifter_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
    IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
IppStatus ippsDCTLifter_32s16s_Sfs (const Ipp32s* pSrc, Ipp16s* pDst,
    const IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
IppStatus ippsDCTLifter_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDCTLifterState_32f* pDCTLifter);
```

引数

pSrc	ソース・ベクトル [*lenDCT*] へのポインタ。
pDst	出力ベクトル [*lenCeps*] または [*lenCeps*+1] へのポインタ。
pDCTLifter	DCT/ リフタリング構造体へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDCTLifter は、ippsr.h ファイルで宣言される。この関数は、最初に DCT を実行し、次に DCT 係数をリフタリングする。

DCT 係数は、次の式に従って計算される。

$$y_i = \sum_{j=1}^{lenDCT} pSrc[j-1] \cdot \cos\left(\frac{\pi \cdot i \cdot (j-0.5)}{lenDCT}\right) \quad , \quad i = 0,...lenCeps$$

出力係数は、次のように重み付けされる。

$$pDst[i-1] = l_i \cdot y_i \; , \quad i = 1,...lenCeps$$

C_0 係数が必要な場合（C0 サフィックスが付いた関数によって *pDCTLifter* が初期化されている場合）は、次のように出力ベクトルの最後の要素として格納される。

$$pDst[lenCeps] = l_0 \cdot y_0$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pFBank* が NULL。

次の例は、MFCC 特徴計算での特徴処理関数の使用方法を示している。

例 8-1　MFCC 特徴計算

```
/* Input: samples[] Input samples
sample_number Number of samples
Output: mfccs[12*(sample_num-240)/160] The resulting MFCC coefficients
*/

void Calc_MFCC (Ipp32f *samples, int sample_num, Ipp32f *mfccs) {
Ipp32f* frame_buffer,fbank_buffer,mfcc_cur;
IppsFBankState_32f *fbank;
IppsDCTLifterState_32f *dctl;
int fft_len,fft_order;
int i,j;
float LogEnergy;

/* Initialize the structures */
ippsMelFBankInitAlloc_32f(&fbank, /* return the structure pointer */
                  &fft_order, /* return the FFT length */
                  400, /* 25ms window/512 point FFT */
                  16000, /* sample rate */
                  64, /* lowest frequency of interest */
```

```
                  8000, /* highest frequency of interest */
                  24, /* number of filter banks */
                  1127.0, /* mel-scale factor 1 */
                  700.0, /* mel-scale factor 2 */
                  IPP_FBANK_MELWGT | IPP_POWER_SPECTRUM);
ippsDCTLifterInitAlloc_32f(&dctl, /* return the structure pointer */
                  24, /* filter bank channels */
                  12, /* number of MFCC coefficients */
                  22, /* liftering */
                  1.0); /* no scaling */
fft_len=1<<fft_order;
frame_buffer=ippsMalloc_32f(fft_len);
fbank_buffer=ippsMalloc_32f(24);
mfcc_cur=mfccs;

/* Calculate MFCC features */
for (i=j=0; i+400<sample_num; i+=160,mfcc_cur+=12,j++) {
       /* Organize the input wave data into a frame */
       ippsCopyWithPadding_32f(&samples[i],400,frame_buffer,fft_len);
       ippsDotProd_32f(frame_buffer,frame_buffer,fft_len,&LogEnergy);
       /* Pre-emphasize the input signal with factor 0.97 */
       ippsPreemphasize_32f(frame_buffer,400,0.97);
       frame_buffer[0]*=(1.0-0.97);
       /* Add the hamming window to the input signal */
       ippsWinHamming_32f_I(frame_buffer,400);
       /* Perform the filter bank analysis */
       ippsEvalFBank_32f(frame_buffer,fbank_buffer,fbank);
       ippsThreshold_LTVal_32f_I(fbank_buffer, 24, 1.0, 1.0);
       ippsLn_32f_I(fbank_buffer, 24);
       /* Perform the DCT analysis and liftering */
       ippsDCTLifter_32f(fbank_buffer,mfcc_cur,dctl);
       mfcc_cur[12] = (float) log ((double) LogEnergy);

}
/* Normalize log energy */
ippsNormEnergy_32f(mfccs+11,12,(sample_num-240)/160,50.0,1.0);
/* Destroy the structures after calculation */
ippsFree(fbank_buffer);
```

```
ippsFree(frame_buffer);
ippsFBankFree_32f(fbank);
ippsDCTLifterFree_32f(dctl);
}
```

NormEnergy

エネルギー値のベクトルを正規化する。

```
IppStatus ippsNormEnergy_32f(Ipp32f* pSrcDst, int step, int height,
    Ipp32f silFloor, Ipp32f enScale);
IppStatus ippsNormEnergy_16s(Ipp16s* pSrcDst, int step, int height,
    Ipp16s silFloor, Ipp16s val, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_32f(Ipp32f* pSrcDst, int step, int height,
    Ipp32f silFloor, Ipp32f maxE, Ipp32f enScale);
IppStatus ippsNormEnergy_RT_16s(Ipp16s* pSrcDst, int step, int height,
    Ipp16s silFloor, Ipp16s maxE, Ipp16s val, Ipp32f enScale);
```

引数

pSrcDst	入力および出力ベクトル [*height***step*] へのポインタ。
step	ベクトル *pSrcDst* のサンプル・ステップ。
height	正規化に使用するサンプル数。
silFloor	無音のフロア値。
val	係数値。
maxE	最大エネルギー値。
enScale	エネルギー・スケール。

説明

関数 `ippsNormEnergy` は、`ippsr.h` ファイルで宣言される。この関数は、エネルギー値を格納した入力ベクトルを正規化する。
正規化は、次のように実行される。

RT サフィックスが付かない関数の場合は、最大エネルギー値は次のように計算される。

$$maxE = \max_{0 < i < height-1} pSrcDst[i \cdot step]$$

すべての関数について、*val*=1 とすると（値が指定されていない場合）、

$minE = maxE - (silFloor \cdot \ln 10)/10$

$0 \le i < height$ の場合、

$pSrcDst[i \cdot step] = val - (maxE - \max(pSrcDst[i \cdot step], minE)) \cdot enScale,$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcDst* ポインタが NULL。
ippStsSizeErr	エラー。*step* または *height* がゼロ以下。

SumMeanVar

ベクトルの和と 2 乗和の両方を計算する。

```
IppStatus ippsSumMeanVar_32f(const Ipp32f* pSrc, int srcStep, int
    height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsSumMeanVar_32f_I(const Ipp32f* pSrc, int srcStep, int
    height, Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
IppStatus ippsSumMeanVar_16s32f(const Ipp16s* pSrc, int srcStep, int
    height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsSumMeanVar_16s32f_I(const Ipp16s* pSrc, int srcStep, int
    height, Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);
IppStatus ippsSumMeanVar_16s32s_Sfs(const Ipp16s* pSrc, int srcStep,
    int height, Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int
    scaleFactor);
IppStatus ippsSumMeanVar_16s32s_ISfs(const Ipp16s* pSrc, int srcStep,
    int height, Ipp32s* pSrcDstMean, Ipp32s* pSrcDstVar, int width, int
    scaleFactor);
```

引数

pSrc	ソース・ベクトル [*height***srcStep*] へのポインタ。
srcStep	ベクトル *pSrc* の行のステップ。
height	*pSrc* の行数。
pDstMean	和を格納するデスティネーション・ベクトル [*width*] へのポインタ。

pDstVar	2 乗和を格納するデスティネーション・ベクトル [*width*] へのポインタ。
pSrcDstMean	和を格納するソース / デスティネーション・ベクトル [*width*] へのポインタ。
pSrcDstVar	2 乗和を格納するソース/デスティネーション･ベクトル [*width*] へのポインタ。
width	ソース・ベクトル *pSrc* の列数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsSumMeanVar は、ippsr.h ファイルで宣言される。この関数は、ソース・ベクトルの和と 2 乗和の両方を次のように計算する。

$j = 0,...width - 1$ の場合、

$$pDstVar[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pDstMean[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j]$$

関数 ippsSumMeanVar_I は、次の式に従ってインプレース計算を実行する。

$j = 0,...width - 1$ の場合、

$$pSrcDstVar[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pSrcDstMean[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j]$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDstMean*、*pDstVar*、*pSrcDstMean*、または *pSrcDstVar* が NULL。

ippStsSizeErr	エラー。*srcStep*、*width*、または *height* がゼロ以下、あるいは *width* が *srcStep* より大きい。

NewVar

和のアキュムレータと 2 乗和のアキュムレータに基づいて分散値を計算する。

```
IppStatus ippsNewVar_32f(const Ipp32f* pSrcMean, const Ipp32f* pSrcVar,
    Ipp32f* pDstVar, int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32f_I(const Ipp32f* pSrcMean, Ipp32f* pSrcDstVar,
    int width, Ipp32f val1, Ipp32f val2);
IppStatus ippsNewVar_32s_Sfs(const Ipp32s* pSrcMean, const Ipp32s*
    pSrcVar, Ipp32s* pDstVar, int width, Ipp32f val1, Ipp32f val2, int
    scaleFactor);
IppStatus ippsNewVar_32s_ISfs(const Ipp32s* pSrcMean, Ipp32s*
    pSrcDstVar, int width, Ipp32f val1, Ipp32f val2, int scaleFactor);
```

引数

pSrcMean	和を計算するベクトル [*width*] へのポインタ。
pSrcVar	2 乗和を計算するベクトル [*width*] へのポインタ。
pSrcDstVar	2 乗和のアキュムレータと結果分散ベクトル [*width*] へのポインタ。
pDstVar	結果分散ベクトル [*width*] へのポインタ。
width	分散ベクトル *pDstVar* の長さ。
val1, *val2*	一定の係数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsNewVar は、ippsr.h ファイルで宣言される。この関数は、和のアキュムレータと 2 乗和のアキュムレータに基づいて、次のように分散値を計算する。

i=0...*width*-1 の場合、

$$pDstVar[i] = (pSrcVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2 ,$$

インプレース関数 ippsNewVar_I は、次の式を使用する。

$i = 0,...width - 1$ の場合、

$$pSrcDstVar[i] = (pSrcDstVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2,$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcMean*、*pSrcVar*、*pDstVar*、または *pSrcDstVar* が NULL。
ippStsSizeErr	エラー。*width* がゼロ以下。

RecSqrt

ベクトルの平方根と、その逆数を計算する。

```
IppStatus ippsRecSqrt_32s_Sfs(Ipp32s* pSrcDst, int len, Ipp32s val, int
    scaleFactor);
IppStatus ippsRecSqrt_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
IppStatus ippsRecSqrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int
    len, Ipp32s val, int scaleFactor);
```

引数

pSrcDst	ソースおよびデスティネーション・ベクトル [*len*] へのポインタ。
len	ベクトル *pSrcDst* の長さ。
val	ソース値を処理するためのしきい値。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsRecSqrt は、ippsr.h ファイルで宣言される。この関数は、ベクトルの平方根を計算し、その逆数を求める。次のように演算する。

$$pSrcDst[j] = \begin{cases} val & , \ if\ pSrcDst[j] < val \\ \dfrac{1}{\sqrt{pSrcDst[j]}} & , \ otherwise \end{cases}$$

ここでは、$j = 0,...,len - 1$ である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下、または *val* がゼロ以下。
ippStsInvZero	警告。すべての *pSrcDst*[i] が *val* より小さい。

AccCovarianceMatrix

共分散行列を累算する。

```
IppStatus ippsAccCovarianceMatrix_16s64f_D2L(const Ipp16s** mSrc, int
    height, const Ipp16s* pMean, Ipp64f** mSrcDst, int width, Ipp64f
    val);
IppStatus ippsAccCovarianceMatrix_32f64f_D2L(const Ipp32f** mSrc, int
    height, const Ipp32f* pMean, Ipp64f** mSrcDst, int width, Ipp64f
    val);
IppStatus ippsAccCovarianceMatrix_16s64f_D2(const Ipp16s* pSrc, int
    srcStep, int height, const Ipp16s* pMean, Ipp64f* pSrcDst, int
    width, int dstStep, Ipp64f val);
IppStatus ippsAccCovarianceMatrix_32f64f_D2(const Ipp32f* pSrc, int
    srcStep, int height, const Ipp32f* pMean, Ipp64f* pSrcDst, int
    width, int dstStep, Ipp64f val);
```

引数

pSrc	入力ベクトル [*height***srcStep*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
srcStep	*pSrc* の行のステップ。
pMean	平均ベクトル [*width*] へのポインタ。
width	入力行列の行の長さ、および平均ベクトルと分散ベクトルの長さ。
pSrcDst	結果ベクトル [*width***dstStep*] へのポインタ。
mSrcDst	結果行列 [*width*][*width*] へのポインタ。
dstStep	*pSrcDst* の行のステップ。
height	入力行列の行数。
val	各距離に掛ける値。

説明

関数 ippsAccCovarianceMatrix は、ippsr.h ファイルで宣言される。この関数は、以下の式に従って、デスティネーション共分散行列の要素を累算する。

D2 サフィックスが付いた関数の場合、

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] +$$

$$+ val \cdot \sum_{k=0}^{height-1} (pSrc[k \cdot srcStep + i] - pMean[j]) \cdot (pSrc[k \cdot srcStep + j] - pMean[j])$$

$$pSrcDst[j \cdot dstStep + i] = pSrcDst[i \cdot dstStep + j] \quad ,$$

($i = 0,...width - 1,\ j = i,...width - 1$ の場合)

D2L サフィックスが付いた関数の場合、

$$mSrcDst[i][j] = mSrcDst[i][j] + val \cdot \sum_{k=0}^{height-1} (mSrc[k][i] - pMean[j]) \cdot (mSrc[k][j] - pMean[j])$$

$$mSrcDst[j][i] = mSrcDst[i][j] \ ,$$

($i = 0,...width - 1,\ j = i,...width - 1$ の場合)

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pMean*、*pSrcDst*、または *mSrcDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*srcStep* または *dstStep* が *width* より小さい。

導関数

この項で説明する関数は、特徴ベクトルの 1 次導関数と 2 次導関数を計算する。これらの関数は、一連の入力特徴ベクトルを処理し、導関数を格納する出力特徴ベクトルを生成する。

入力シーケンス a_0 ,..., a_{N-1} は、サイズ $N \cdot M$ の 1 次元配列として格納される（N は特徴ベクトルの数、M は各特徴 a_i の次元である）。同様に、出力シーケンス b_0 ,..., b_{N-1} も、サイズ $N \cdot K$ の 1 次元配列として格納される（K は各特徴 b_j の次元である）。M と K の値は、以下の制約条件に従っていなければならない。

$K \geq M$

$K \geq 2M$ （1 次導関数を生成する場合）

$K \geq 3M$ （1 次導関数と 2 次導関数を生成する場合）

`ippsCopyColumn` 関数は、入力シーケンスを出力シーケンスにコピーする（つまり、ベースとなる特徴を出力シーケンス内に配置する）。ベースとなる特徴は、各出力ベクトルの最初の M 個の要素に格納される。次に、関数 `ippsEvalDelta` を使用して導関数を計算する。

これらの 2 つの関数は、一般的な導関数計算に使用される。関数 `ippsDelta` と `ippsDeltaDelta` は、関数 `ippCopyColumn` と `ippsEvalDelta` を組み合わせた機能を持つ。

導関数演算は、履歴内の特徴ベクトルと将来の特徴ベクトルの両方を利用する。したがって、最初の `winSize`（デルタ・ウィンドウのサイズ）特徴と最後の `winSize` 特徴には、特殊な取り扱いが必要である。最初の `winSize` 特徴については、通常は最初の特徴ベクトルを繰り返して履歴を提供する。最後の `winSize` 特徴については、最後の特徴ベクトルを繰り返して将来の情報を提供する。

CopyColumn

入力シーケンスを、出力シーケンスにコピーする。

```
IppStatus ippsCopyColumn_16s_D2(const Ipp16s* pSrc, int srcWidth,
    Ipp16s* pDst, int dstWidth, int height);
IppStatus ippsCopyColumn_32f_D2(const Ipp32f* pSrc, int srcWidth,
    Ipp32f* pDst, int dstWidth, int height);
```

引数

pSrc	入力特徴シーケンス [*height***srcWidth*] へのポインタ。
srcWidth	各入力特徴ベクトルの長さ。

pDst	出力特徴シーケンス [*height*dstWidth*] へのポインタ。
dstWidth	各出力特徴ベクトルの長さ。
height	シーケンスの特徴の数。

説明

関数 ippsCopyColumn は、ippsr.h ファイルで宣言される。この関数は、入力特徴シーケンスを出力シーケンスにコピーする。

$$pDst[j \cdot dstWidth + i] = pSrc[j \cdot srcWidth + i] ,$$

$0 \le i < srcWidth,\ 0 \le j < height$

出力シーケンスの未指定の要素は変更されない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*height* または *srcWidth* がゼロ以下。
ippStsStrideErr	エラー。*dstWidth* が *srcWidth* より小さい。

EvalDelta

特徴ベクトルの導関数を計算する。

```
IppStatus ippsEvalDelta_16s_D2Sfs(Ipp16s* pSrcDst, int height, int
    step, int width, int offset, int winSize, Ipp16s val, int
    scaleFactor);
IppStatus ippsEvalDelta_32f_D2(Ipp32f* pSrcDst, int height, int step,
    int width, int offset, int winSize, Ipp32f val);
IppStatus ippsEvalDeltaMul_16s_D2Sfs(Ipp16s* pSrcDst, int height, int
    step, const Ipp16s* pVal, int width, int offset, int winSize, int
    scaleFactor);
IppStatus ippsEvalDeltaMul_32f_D2(Ipp32f* pSrcDst, int height, int
    step, const Ipp32f* pVal, int width, int offset, int winSize);
```


intel®

引数

pSrcDst	入力および出力シーケンス [*height***step*] へのポインタ。
height	*pSrcDst* 内の特徴の数。
step	*pSrcDst* 内の各特徴の長さ。
width	各特徴ごとに計算される導関数の数。
offset	導関数の値を配置する際のオフセット。
winSize	デルタ・ウィンドウのサイズ。
val	デルタ係数。
pVal	デルタ係数ベクトル [*width*] へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsEvalDelta は、ippsr.h ファイルで宣言される。この関数は、入力特徴シーケンスの導関数を計算する。ベースとなる特徴の範囲は、各特徴ベクトル内の *offset* から (*offset* + *width* - 1) までである。出力導関数は、各特徴ベクトル内でベースとなる特徴に隣接する、(*offset* + *width*) から (*offset* + 2**width* - 1) までの範囲に格納される。演算は次のように行われる。

関数 ippsEvalDelta の場合、

$$pSrcDst[i \cdot step + width + j] = val \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$$0 \le i < height, \quad offset \le j < offset + width$$

関数 ippsEvalDeltaMul の場合、

$$pSrcDst[i \cdot step + width + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pSrcDst[\min(i+k, height-1) \cdot step + j] - pSrcDst[\max(i-k, 0) \cdot step + j]\},$$

$$0 \le i < height, \quad offset \le j < offset + width$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcDst* ポインタが NULL。
ippStsSizeErr	エラー。*height*、*width*、または *winSize* がゼロ以下、*offset* がゼロより小さい、あるいは *height* が 2**winSize* より小さいのいずれかである。
ippStsStrideErr	エラー。step が *offset*+2**width* より小さい。

Delta

ベースとなる特徴をコピーし、特徴ベクトルの導関数を計算する。

```
IppStatus ippsDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
    Ipp16s* pDst, int dstStep, int height, Ipp16s val, int deltaMode,
    int scaleFactor);
IppStatus ippsDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
    Ipp16s* pDst, int dstStep, int height, Ipp16s val, int deltaMode,
    int scaleFactor);
IppStatus ippsDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth,
    Ipp32f* pDst, int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippsDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth,
    Ipp32f* pDst, int dstStep, int height, Ipp32f val, int deltaMode);
IppStatus ippsDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
    pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int
    deltaMode, int scaleFactor);
IppStatus ippsDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
    pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int
    deltaMode, int scaleFactor);
IppStatus ippsDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f*
    pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height, int
    deltaMode);
IppStatus ippsDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f*
    pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height, int
    deltaMode);
```

引数

pSrc	入力機能シーケンス [*height*srcWidth*] へのポインタ。

srcWidth	入力シーケンス *pSrc* の特徴ベクトルの長さ。
pDst	出力特徴のシーケンスへのポインタ。
dstStep	出力シーケンス *pDst* の特徴ベクトルの長さ。
height	特徴ベクトルの数。
val	デルタ係数。
deltaMode	実行モード。
pVal	デルタ係数ベクトル [*width*] へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDelta と ippsDeltaMul は、ippsr.h ファイルで宣言される。これらの関数は、ippsCopyColumn と ippsEvalDelta を組み合わせた機能を持つ。最初に、入力特徴ベクトルが出力シーケンスにコピーされる。次に、導関数が計算される。

以下の説明では、次の条件が適用される。

関数サフィックス Win1 または Win2 は、それぞれデルタ・ウィンドウのサイズ *winSize*=1 または 2 を指定する。

関数 ippsDelta は、デルタ係数に関する次の制約条件を前提とする。

$pVal[j] \equiv val, \ \forall j$

実行モード *deltaMode* は、ベースとなる特徴のコピーと導関数の計算プロセスを制御する。*deltaMode* の許容される値と、各値に対応する関数実行ロジックは、次のとおりである。

1. *deltaMode* が IPP_DELTA_BEGIN|IPP_DELTA_END の場合

オフラインのデルタ特徴計算を実行する。計算の実行時にすべてのベースとなる特徴が利用可能になっていると見なされる。最初に、次の式に従って、入力ストリーム *pSrc* から出力ストリーム *pDst* にベースとなる特徴がコピーされる。

$0 \le i < height$、$0 \le j < srcWidth$ の場合、

$$pDst[i \cdot dstStep + j] = pSrc[i \cdot srcWidth + j] , \quad (8.4)$$

続いて、次の式に従って導関数が計算される。

$0 \le i < height$、$0 \le j < srcWidth$ の場合、

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[\min(i+k, height-1) \cdot dstStep + j] -$$

$$- pDst[\max(i-k, 0) \cdot dstStep + j]\}, \quad (8.5)$$

2. *deltaMode* が 0 の場合

オンラインのデルタ特徴計算を実行する。入力特徴は、連続的なストリームの現在のセグメントである。関数 ippsDelta は、現在の入力に従って部分的なデルタ特徴を計算する。最初に、次の式に従ってベースとなる特徴がコピーされる。

$0 \le i < height$,　$0 \le j < srcWidth$ の場合、

$$pDst[(i + 2 \cdot winSize) \cdot dstStep + j] = pSrc[i \cdot srcWidth + j], \quad (8.6)$$

続いて、次の式に従って導関数が計算される。

$0 \le i - winSize < height$, $0 \le j < srcWidth$ の場合、

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[(i+k) \cdot dstStep + j] -$$

$$- pDst[(i-k) \cdot dstStep + j]\}, \quad (8.7)$$

この実行モードでは、$pDst[i \cdot dstStep + j]$ 内のベースとなる特徴と $pDst[k \cdot dstStep + srcWidth + j]$ 内の導関数は、$0 \le j < srcWidth$、$0 \le i < 2 \cdot winSize$、および $0 \le k < winSize$ の場合における前回の導関数計算で利用可能とみなされる。

3. *deltaMode* が IPP_DELTA_BEGIN の場合

入力ストリームの開始点がわかっている、部分的なオンライン・デルタ特徴計算を実行する。最初に、式（8.4）に従ってベースとなる特徴がコピーされる。続いて、次の式に従って導関数が計算される。

$0 \le i < height - winSize$, $0 \le j < srcWidth$ の場合、

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{pDst[(i+k) \cdot dstStep + j] -$$

$$- pDst[\max(i-k, 0) \cdot dstStep + j]\}, \quad (8.8)$$

4. *deltaMode* が IPP_DELTA_END の場合

入力ストリームの終了点がわかっている、部分的なオンライン・デルタ特徴計算を実行する。最初に、式（8.6）に従ってベースとなる特徴がコピーされる。続いて、次の式に従って導関数が計算される。

$0 \le i - winSize < height + winSize$, $0 \le j < srcWidth$ の場合、

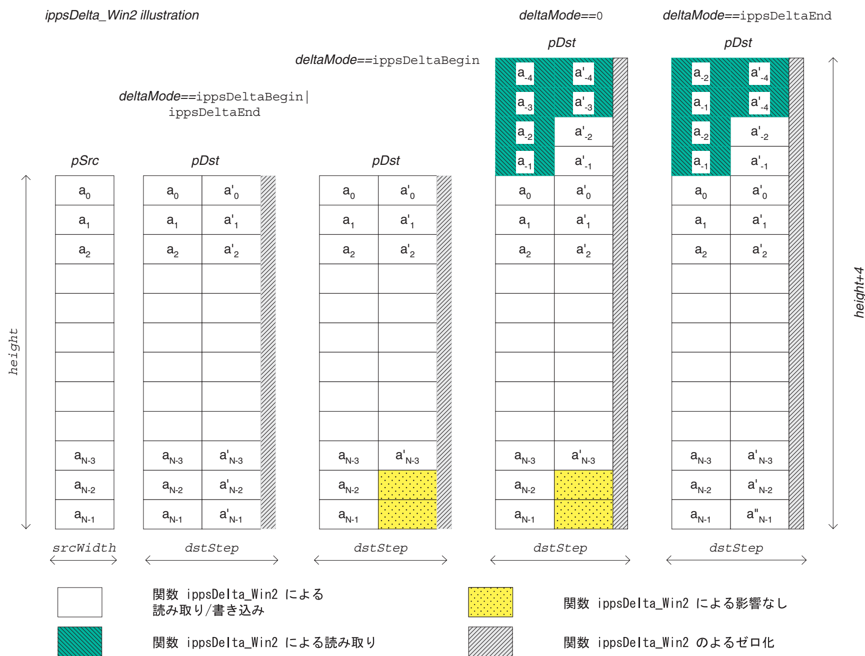
$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{$$

$$pDst[\min(i+k, height + 2 \cdot winSize - 1) \cdot dstStep + j] - pDst[(i-k) \cdot dstStep + j]\}$$

この実行モードでは、$pDst[i \cdot dstStep + j]$ 内のベースとなる特徴と $pDst[k \cdot dstStep + srcWidth + j]$ 内の導関数は、$0 \le j < srcWidth$、$0 \le i < 2 \cdot winSize$、および $0 \le k < winSize$ の場合における前回の導関数計算で利用可能とみなされる。

次の図は、上の 4 つのデルタ計算モードを示している。

図 8-1　ippsDelta_Win2 関数の実行モード



戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pVal* が NULL。
ippStsSizeErr	エラー。*srcWidth* がゼロ以下、*height* がゼロ以下、IPP_DELTA_BEGIN が設定されており、*height* が *winSize*+1 以下、IPP_DELTA_BEGIN が設定されてなく、*height* がゼロより小さい、のいずれか。
ippStsStrideErr	エラー。*dstStep* が 2**srcWidth* より小さい。

DeltaDelta

ベースとなる特徴をコピーし、1 次導関数
と 2 次導関数を計算する。

```
IppStatus ippsDeltaDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int
    srcWidth, Ipp16s* pDst, int dstStep, int height, Ipp16s val1,
    Ipp16s val2, int deltaMode, int scaleFactor);
IppStatus ippsDeltaDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int
    srcWidth, Ipp16s* pDst, int dstStep, int height, Ipp16s val1,
    Ipp16s val2, int deltaMode, int scaleFactor);
IppStatus ippsDeltaDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth,
    Ipp32f* pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2,
    int deltaMode);
IppStatus ippsDeltaDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth,
    Ipp32f* pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2,
    int deltaMode);
IppStatus ippsDeltaDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const
    Ipp16s* pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height,
    int deltaMode, int scaleFactor);
IppStatus ippsDeltaDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const
    Ipp16s* pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height,
    int deltaMode, int scaleFactor);
IppStatus ippsDeltaDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const
    Ipp32f* pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height,
    int deltaMode);
IppStatus ippsDeltaDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const
    Ipp32f* pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height,
    int deltaMode);
```

引数

pSrc	入力機能シーケンス [*height*srcWidth*] へのポインタ。
srcWidth	入力シーケンス *pSrc* の特徴ベクトルの長さ。
pDst	出力特徴のシーケンスへのポインタ。
dstStep	出力シーケンス *pDst* の特徴ベクトルの長さ。
height	特徴ベクトルの数。
val1, val2	デルタ係数。
deltaMode	実行モード。
pVal	デルタ係数ベクトル [*width*] へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDeltaDelta および関数 ippsDeltaDeltaMul は、ippsr.h ファイルで宣言される。これらの関数は、ippsCopyColumn と、ippsEvalDelta の二重演算を組み合わせた機能を持つ。最初に、入力特徴ベクトルが出力シーケンスにコピーされる。次に、1 次導関数と 2 次導関数が計算される。

以下の説明では、次の条件が適用される。

関数サフィックス Win1 または Win2 は、それぞれデルタ・ウィンドウのサイズ *winSize*=1 または 2 を指定する。

関数 ippsDeltaDelta は、デルタ係数に関する次の制約条件を前提とする。

pVal[*j*] ≡ *val*1， ∀*j*、1 次導関数計算の場合

pVal[*j*] ≡ *val*2， ∀*j*、2 次導関数計算の場合

実行モード *deltaMode* は、ベースとなる特徴のコピーと導関数の計算プロセスを制御する。*deltaMode* の許容される値と、各値に対応する関数実行ロジックは、次のとおりである。

1. *deltaMode* が IPP_DELTA_BEGIN|IPP_DELTA_END の場合

   オフラインのデルタデルタ特徴計算を実行する。計算の実行時にすべてのベースとなる特徴が利用可能になっていると見なされる。最初に、(8.4) に従って、入力ストリーム *pSrc* から出力ストリーム *pDst* にベースとなる特徴がコピーされる。

次に、(8.5)($0 \le i < height$, $0 \le j < srcWidth$ の場合)に従って 1 次導関数が計算される。最後に、式(8.5)($0 \le i < height$, $0 \le j - srcWidth < srcWidth$ の場合)に従って 2 次導関数が計算される。

2. *deltaMode* が 0 の場合

オンラインのデルタデルタ特徴計算を実行する。入力特徴は、連続的なストリームの現在のセグメントである。関数 ippsDeltaDelta は、現在の入力に従って部分的なデルタデルタ特徴を計算する。

最初に、次の式に従ってベースとなる特徴がコピーされる。

$0 \le i < height$, $0 \le j < srcWidth$ の場合、

$$pDst[(i + 3 \cdot winSize) \cdot dstStep + j] = pSrc[i \cdot srcWidth + j] \quad , \qquad (8.9)$$

次に、式(8.7)($0 \le i - 2 \cdot winSize < height$ および $0 \le j < srcWidth$ の場合)に従って 1 次導関数が計算される。

最後に、式(8.7)($0 \le i - winSize < height$, $0 \le j - srcWidth < srcWidth$ の場合)に従って 2 次導関数が計算される。

この実行モードでは、$pDst[i \cdot dstStep + j]$ 内のベースとなる特徴、$pDst[k \cdot dstStep + srcWidth + j]$ 内の 1 次導関数、および $pDst[l \cdot dstStep + 2 \cdot srcWidth + j]$ 内の 2 次導関数は、$0 \le j < srcWidth$、$0 \le i < 3 \cdot winSize$、$0 \le k < 2 \cdot winSize$、および $0 \le l < winSize$ の場合における前回のデルタデルタ計算で利用可能とみなされる。

3. *deltaMode* が IPP_DELTA_BEGIN の場合

入力ストリームの開始点がわかっている、部分的なオンライン・デルタ特徴計算を実行する。最初に、式(8.4)に従ってベースとなる特徴がコピーされる。次に、式(8.8)($0 \le i < height - winSize$ および $0 \le j < srcWidth$ の場合)に従って 1 次導関数が計算される。

最後に、式(8.8)($0 \le i < height - 2 \cdot winSize$ および $0 \le j - srcWidth < srcWidth$ の場合)によって 2 次導関数が計算される。

4. `deltaMode` が `IPP_DELTA_END` の場合

入力ストリームの終了点がわかっている、部分的なオンライン・デルタデルタ特徴計算を実行する。最初に、式（8.9）に従ってベースとなる特徴がコピーされる。次に、次の式に従って 1 次導関数が計算される。

$0 \le i - 2 \cdot winSize < height + winSize,\ 0 \le j < srcWidth$ の場合、

$$pDst[i \cdot dstStep + srcWidth + j] = pVal[j] \cdot \sum_{k=1}^{winSize} k \cdot \{ \quad (8.10)$$

$$pDst[\min(i+k, height + 3 \cdot winSize - 1) \cdot dstStep + j] - pDst[(i-k) \cdot dstStep + j]\}$$

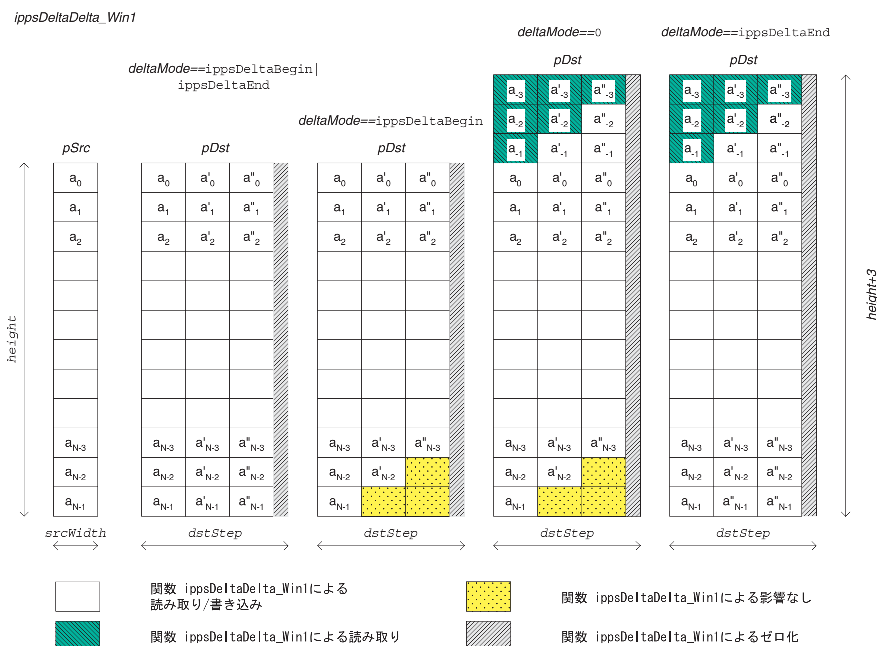
最後に、式（8.10）（添字 i および j は次の範囲内で変化する）に従って 2 次導関数が計算される。

$0 \le i - winSize < height + 2 \cdot winSize,\ 0 \le j - srcWidth < srcWidth$

この実行モードでは、$pDst[i \cdot dstStep + j]$ 内のベースとなる特徴、$pDst[k \cdot dstStep + srcWidth + j]$ 内の 1 次導関数、および $pDst[l \cdot dstStep + 2 \cdot srcWidth + j]$ 内の 2 次導関数は、$0 \le j < srcWidth$、$0 \le i < 3 \cdot winSize$、$0 \le k < 2 \cdot winSize$、および $0 \le l < winSize$ の場合における前回のデルタデルタ計算で利用可能とみなされる。

次の図は、上の 4 つのデルタ計算モードを示している。

図 8-2　ippsDeltaDelta_Win1 関数の実行モード



戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pVal* が NULL。
ippStsSizeErr	エラー。*srcWidth* がゼロ以下、*height* がゼロ以下、IPP_DELTA_BEGIN が設定されていて *height* が 2*(*winSize*+1) 以下、IPP_DELTA_BEGIN が設定されてなく、*height* がゼロより小さい、のいずれか。
ippStsStrideErr	エラー。*dstStep* が 3**srcWidth* より小さい。

ピッチ超解像度

この項では、ピッチ解像度アルゴリズム（[Med91] を参照）に使用される関数について説明する。

CrossCorrCoeffDecim

デシメーションを使用して、相互相関係数のベクトルを計算する。

```
IppStatus ippsCrossCorrCoeffDecim_16s32f(const Ipp16s* pSrc1, const
    Ipp16s* pSrc2, int maxLen, int minLen, Ipp32f* pDst, int dec);
```

引数

pSrc1	1 番目の入力ベクトル [*maxLen*] へのポインタ。
pSrc2	2 番目の入力ベクトル [*maxLen*] へのポインタ。
pDst	出力ベクトル [(*maxLen*-*minLen*)/*dec*+1] へのポインタ。
maxlen	相互相関の最大長。
minlen	相互相関の最小長。
dec	デシメーション・ステップ。

説明

関数 ippsCrossCorrCoeffDecim は、ippsr.h ファイルで宣言される。この関数は、デシメーション・ステップ *dec* を使用して、長さが *minlen* から *maxlen* までの相互相関係数のベクトルを計算する。

計算は、次の式に従って実行される。

$$pDst[k] = \begin{cases} 0, & if\ (\mathbf{x}^k, \mathbf{x}^k)_{dec} \cdot (\mathbf{y}^k, \mathbf{y}^k)_{dec} = 0 \\ \dfrac{(\mathbf{x}^k, \mathbf{y}^k)_{dec}}{\sqrt{(\mathbf{x}^k, \mathbf{x}^k)_{dec}} \cdot \sqrt{(\mathbf{y}^k, \mathbf{y}^k)_{dec}}}, & otherwise \end{cases}$$

ここで、

$$\mathbf{x}_i^k = pSrc1[maxLen - minLen - dec \cdot k + i] \ , \quad \mathbf{y}_i^k = pSrc2[i] \ ,$$

$$i = 0, \ldots minLen + dec \cdot k - 1 \ , \quad k = 0, \ldots (maxLen - minLen)/dec ,$$

$(\mathbf{a}, \mathbf{b})_{dec}$ は、2 つのベクトル **a** と **b** のデシメーションされた内積を示す。

$$(\mathbf{a}, \mathbf{b})_{dec} = \sum_{i=0}^{(len-1)/dec} a[i \cdot dec] \times b[i \cdot dec]$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pDst* が NULL。
ippStsSizeErr	エラー。*minlen* または *dec* がゼロ以下、あるいは *maxlen* が *minlen* より小さい。

CrossCorrCoeff

相互相関係数を計算する。

```
IppStatus ippsCrossCorrCoeff_16s32f(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32f* pResult);
IppStatus ippsCrossCorrCoeffPartial_16s32f(const Ipp16s* pSrc1, const
    Ipp16s* pSrc2, int len, Ipp32f val, Ipp32f* pResult);
```

引数

pSrc1	1 番目の入力ベクトル [*len*] へのポインタ。
pSrc2	2 番目の入力ベクトル [*len*] へのポインタ。
len	入力ベクトルの長さ。
pResult	結果の相互相関係数値へのポインタ。
val	1 番目のベクトルの大きさの 2 乗として使用される値。

説明

関数 ippsCrossCorrCoeff と ippsCrossCorrCoeffPartial は、ippsr.h ファイルで宣言される。これらの関数は、以下に示す式に従って、2 つのベクトルの相互相関係数を計算する。

関数 ippsCrossCorrCoeff の場合、

$$pResult[0] = \begin{cases} 0, & if\ (\mathbf{x}, \mathbf{x}) \cdot (\mathbf{y}, \mathbf{y}) = 0 \\ \dfrac{(\mathbf{x}, \mathbf{y})}{\sqrt{(\mathbf{x}, \mathbf{x})} \cdot \sqrt{(\mathbf{y}, \mathbf{y})}}, & otherwise \end{cases}$$

関数 ippsCrossCorrCoeffPartial の場合、

$$pResult[0] = \begin{cases} 0, & if\ val \cdot (\mathbf{y}, \mathbf{y}) = 0 \\ \dfrac{(\mathbf{x}, \mathbf{y})}{\sqrt{val} \cdot \sqrt{(\mathbf{y}, \mathbf{y})}}, & otherwise \end{cases}$$

ここで、

$\mathbf{x}_i = pSrc1[i-1]$, $\mathbf{y}_i = pSrc2[i-1]$ （$i = 1, ...len$ の場合）

(*a*, *b*) は、2 つのベクトル ***a*** と ***b*** の内積を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pResult* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsBadArgErr	エラー。*val* がゼロより小さい。

CrossCorrCoeffInterpolation

補間された相互相関係数を計算する。

```
IppStatus ippsCrossCorrCoeffInterpolation_16s32f(const Ipp16s* pSrc1,
    const Ipp16s* pSrc2, int len, Ipp32f* pBeta, Ipp32f* pResult);
```

引数

pSrc1	1 番目の入力ベクトル [*len*] へのポインタ。
pSrc2	2 番目の入力ベクトル [*len*+1] へのポインタ。
len	入力ベクトルの長さ。
pResult	結果の相互相関係数値へのポインタ。
pBeta	結果のピッチ周期の小数点以下の部分へのポインタ。

説明

関数 `ippsCrossCorrCoeffInterpolation` は、`ippsr.h` ファイルで宣言される。この関数は、2 つのベクトルの補間された相互相関係数 *pResult* と、ピッチ周期の小数点以下の部分 *pBeta* を、次の式に従って計算する。

$$pBeta[0] = \beta = \frac{(\boldsymbol{x}, \boldsymbol{y}') \cdot (\boldsymbol{y}, \boldsymbol{y}) - (\boldsymbol{x}, \boldsymbol{y}) \cdot (\boldsymbol{y}, \boldsymbol{y}')}{(\boldsymbol{x}, \boldsymbol{y}') \cdot [(\boldsymbol{y}, \boldsymbol{y}) - (\boldsymbol{y}, \boldsymbol{y}')] + (\boldsymbol{x}, \boldsymbol{y}) \cdot [(\boldsymbol{y}', \boldsymbol{y}') - (\boldsymbol{y}, \boldsymbol{y}')]},$$

また、$0 \le \beta < 1$ の場合、

$$pResult[0] = \frac{(1-\beta) \cdot (\boldsymbol{x}, \boldsymbol{y}) + \beta \cdot (\boldsymbol{x}, \boldsymbol{y}')}{\sqrt{(\boldsymbol{x}, \boldsymbol{x}) \cdot ((1-\beta)^2 \cdot (\boldsymbol{y}, \boldsymbol{y}) + 2\beta \cdot (1-\beta) \cdot (\boldsymbol{y}, \boldsymbol{y}') + \beta^2 \cdot (\boldsymbol{y}', \boldsymbol{y}'))}},$$

ここで、

$\boldsymbol{x}_i = pSrc1[i-1]$, $\boldsymbol{y}_i = pSrc2[i-1]$, $\boldsymbol{y}_i' = pSrc2[i]$, $i = 1, ...len$,

(*a*, *b*) は、2 つのベクトル ***a*** と ***b*** の内積を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、*pBeta* または *pResult* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsStrideErr	エラー。*srcStep* または *dstStep* が *width* より小さい。
ippStsDivByZero	警告。*pBeta*[0] の式の分母の値がゼロである。演算の実行は中止されない。

モデル評価

この項では、音響モデルと言語モデルを評価する関数について説明する。

AddNRows

N 個のベクトルをベクトル配列から
追加する。

```
IppStatus ippsAddNRows_32f_D2(Ipp32f* pSrc, int height, int offset,
    int step, Ipp32s* pInd, Ipp16u* pAddInd, int rows, Ipp32f* pDst,
    int width, Ipp32f weight);
```

引数

pSrc	ベクトル配列 [*height***step*] へのポインタ。
height	ベクトル配列 *pSrc* の行数。
offset	ベクトル配列 *pSrc* 内の対象ベクトルのオフセット。
step	ベクトル配列 *pSrc* の行のステップ。
pInd	インデックス・ベクトル [*rows*] へのポインタ。
pAddInd	追加のインデックス・ベクトル [*rows*] へのポインタ。
rows	追加するベクトルの数。
pDst	出力ベクトル [*width*] へのポインタ。
width	出力ベクトル *pDst* の長さ。
weight	出力に加算される重みの値。

説明

関数 ippsAddNRows は、ippsr.h ファイルで宣言される。この関数は、次のように、インデックス・ベクトル *pInd* および *pAddInd* の和に従って、ベクトル配列 *pSrc* から *rows* 個のベクトルの和の合計を計算する。

$$pDst[k] = weight + \sum_{i=0}^{rows} pSrc[k + offset + step \cdot (pInd[i] + pAddInd[i])] \ ,$$

$0 \le k < width$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pInd*、*pAddInd*、または *pDst* が NULL。

ippStsSizeErr	エラー。*height*、*width*、*rows* がゼロ以下、(*pInd*[*i*]+*pAddInd*[*i*]) か *offset* がゼロより小さい、または (*pInd*[*i*]+*pAddInd*[*i*]) が *height* 以上。
ippStsStrideErr	エラー。*step* が *width* + *offset* より小さい。

ScaleLM

しきい値演算を使用してベクトル要素を
スケーリングする。

```
IppStatus ippsScaleLM_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len,
    Ipp16s floor, Ipp16s scale, Ipp32s base);
IppStatus ippsScaleLM_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
    Ipp32f floor, Ipp32f scale, Ipp32f base);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pDst	出力ベクトル [*len*] へのポインタ。
len	ベクトル *pSrc* または *pDst* の長さ。
floor	しきい値。
scale	スケール係数。
base	追加の係数。

説明

関数 ippsScaleLM は、ippsr.h ファイルで宣言される。この関数は、次のように、入力ベクトル *pSrc* にしきい値を設定し、ベクトル要素をスケーリングする。

$$pDst[n] = scale \cdot \max(pSrc[n],\ floor) + base\ ,\ 0 \le n < len$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

LogAdd

対数表現の 2 つのベクトルを加算する。

```
IppStatus ippsLogAdd_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len,
    IppHintAlgorithm hint);
IppStatus ippsLogAdd_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
    IppHintAlgorithm hint);
IppStatus ippsLogAdd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int
    len, int scaleFactor, IppHintAlgorithm hint);
IppStatus ippsLogAdd_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int
    len, int scaleFactor, IppHintAlgorithm hint);
```

引数

pSrc	1 番目の入力ベクトル [*len*] へのポインタ。
pSrcDst	2 番目の入力ベクトルと出力ベクトル [*len*] へのポインタ。
len	入力ベクトルと出力ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。
hint	対数による加算で、特別のコードの使用を指定する。

説明

関数 ippsLogAdd は、ippsr.h ファイルで宣言される。この関数は、対数表現の要素を持つベクトル *pSrc* と *pSrcDst* を加算する。浮動小数点の引数を処理する関数の場合、対数表現の出力ベクトル *pSrcDst* は次のように計算される。

$$pSrcDst[i] = \ln(e^{pSrc[i]} + e^{pSrcDst[i]}),\quad 0 \le i < len$$

整数の引数を処理する関数の場合、出力ベクトル *pSrcDst* は次のように計算される。

$$pSrcDst[i] = 2^{scaleFactor} \cdot \ln\left(e^{pSrc[i] \cdot 2^{-scaleFactor}} + e^{pSrcDst[i] \cdot 2^{-scaleFactor}}\right),$$

$$0 \le i < len$$

hint 引数は、特別なコード（高速だが精度が低い計算、または精度は高いが低速な計算を実行するコード）の使用を指定する。*hint* 引数に指定可能な値は、表 7-3 の「flag 引数と hint 引数」に記載されている。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

LogSub

対数表現の 2 つのベクトルの差を求める。

```
IppStatus ippsLogSub_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsLogSub_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsLogSub_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int
    len, int scaleFactor);
IppStatus ippsLogSub_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int
    len, int scaleFactor);
```

引数

pSrc	1 番目の入力ベクトル [*len*] へのポインタ。
pSrcDst	2 番目の入力ベクトルおよび出力ベクトル [*len*] へのポインタ。
len	入力ベクトルと出力ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsLogSub は、ippsr.h ファイルで宣言される。この関数は、対数表現のベクトル *pSrc* から対数表現のベクトル *pSrcDst* を引く。浮動小数点の引数を処理する関数の場合、出力ベクトル *pSrcDst* は次のように計算される。

$$pSrcDst[i] = \ln(e^{pSrc[i]} - e^{pSrcDst[i]}),\ 0 \le i < len$$

整数の引数を処理する関数の場合、出力ベクトル *pSrcDst* は次のように計算される。

$$pSrcDst[i] = 2^{scaleFactor} \cdot \ln\left(e^{pSrc[i] \cdot 2^{-scaleFactor}} - e^{pSrcDst[i] \cdot 2^{-scaleFactor}}\right),$$

$$0 \le i < len$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsRangeErr	エラー。*pSrc*[*i*] が *pSrcDst*[*i*] 以下。

LogSum

対数表現のベクトル要素の和を求める。

```
IppStatus ippsLogSum_32f(const Ipp32f* pSrc, Ipp32f* pResult, int len,
    IppHintAlgorithm hint);
IppStatus ippsLogSum_64f(const Ipp64f* pSrc, Ipp64f* pResult, int len,
    IppHintAlgorithm hint);
IppStatus ippsLogSum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pResult, int
    len, int scaleFactor, IppHintAlgorithm hint);
IppStatus ippsLogSum_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pResult, int
    len, int scaleFactor, IppHintAlgorithm hint);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pResult	結果の値へのポインタ。
len	入力ベクトル内の要素の数。
scaleFactor	第 2 章の「整数のスケーリング」を参照。
hint	対数による加算で、特別のコードの使用を指定する。

説明

関数 ippsLogSum は、ippsr.h ファイルで宣言される。この関数は、対数表現のソース・ベクトル要素の和を求める。浮動小数点の引数を処理する関数の場合、出力ベクトルも対数表現になり、次のように計算される。

$$pResult[0] = \ln \sum_{i=0}^{len-1} e^{pSrc[i]}$$

整数の引数を処理する関数の場合、出力ベクトルは次のように計算される。

$$pResult[0] = 2^{scaleFactor} \cdot \ln\left(\sum_{i=0}^{len-1} e^{pSrc[i] \cdot 2^{-scaleFactor}}\right)$$

hint 引数は、特別なコード（高速だが精度が低い計算、または精度は高いが低速な計算を実行するコード）の使用を指定する。*hint* 引数に指定可能な値は、表 7-3 の「flag 引数と hint 引数」に記載されている。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pResult* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

MahDistSingle

単一の観測ベクトルについてマハラノビス距離を計算する。

```
IppStatus ippsMahDistSingle_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, int
    scaleFactor);
IppStatus ippsMahDistSingle_16s32f(const Ipp16s* pSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_32f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int len, Ipp32f* pResult);
IppStatus ippsMahDistSingle_64f(const Ipp64f* pSrc, const Ipp64f*
    pMean, const Ipp64f* pVar, int len, Ipp64f* pResult);
IppStatus ippsMahDistSingle_32f64f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int len, Ipp64f* pResult);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pMean	平均ベクトル [*len*] へのポインタ。
pVar	分散ベクトル [*len*] へのポインタ。

len	入力ベクトル、平均ベクトル、分散ベクトル内の要素の数。
pResult	結果へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsMahDistSingle は、ippsr.h ファイルで宣言される。この関数は、平均ベクトル *hint* と分散ベクトル *hint* に基づき、入力ベクトル *hint* についてマハラノビス距離を計算する。次のようにする。

$$pResult[0] = \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pMean*、*pVar*、または *pResult* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

MahDist

複数の平均値と分散値について 距離を計算する。

```
IppStatus ippsMahDist_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int
    height);
IppStatus ippsMahDist_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
    const Ipp32f* pVar, int width, Ipp32f* pDst, int height);
IppStatus ippsMahDist_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int
    height);
IppStatus ippsMahDist_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
    const Ipp64f* pVar, int width, Ipp64f* pDst, int height);
```

引数

pSrc	入力ベクトル [*height***step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
step	入力ベクトル *pSrc* の行のステップ。
pMean	平均ベクトル [*width*] へのポインタ。
pVar	分散ベクトル [*width*] へのポインタ。
width	平均ベクトルと分散ベクトルの長さ。
pDst	結果のベクトル [*height*] へのポインタ。
height	入力行列 *mSrc* の行数。

説明

関数 ippsMahDist は、ippsr.h ファイルで宣言される。この関数は、平均ベクトル *pMean* と分散ベクトル *pVar* に基づいて、複数の入力ベクトルについてマハラノビス距離を計算する。次のようにする。

D2 サフィックスが付いた関数の場合、

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 \quad ,0 \le i < height$$

D2L サフィックスが付いた関数の場合、

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 \quad ,\ 0 \le i < height$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pMean*、*pVar*、または *pDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

intel®

MahDistMultiMix

複数の平均値と分散値についてマハラノビス
距離を計算する。

```
IppStatus ippsMahDistMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
    pVar, int step, const Ipp32f* pSrc, int width, Ipp32f* pDst, int
    height);
IppStatus ippsMahDistMultiMix_32f_D2L(const Ipp32f** mMean, const
    Ipp32f** mVar, const Ipp32f* pSrc, int width, Ipp32f* pDst, int
    height);
IppStatus ippsMahDistMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
    pVar, int step, const Ipp64f* pSrc, int width, Ipp64f* pDst, int
    height);
IppStatus ippsMahDistMultiMix_64f_D2L(const Ipp64f** mMean, const
    Ipp64f** mVar, const Ipp64f* pSrc, int width, Ipp64f* pDst, int
    height);
```

引数

pMean	平均ベクトル [*height***step*] へのポインタ。
pVar	分散ベクトル [*height***step*] へのポインタ。
mMean	平均行列 [*height*][*width*] へのポインタ。
mVar	分散行列 [*height*][*width*] へのポインタ。
step	平均および分散ベクトルの行のステップ。
pSrc	入力ベクトル [*width*] へのポインタ。
width	入力ベクトル *pSrc* の長さ。
pDst	結果のベクトル [*height*] へのポインタ。
height	結果ベクトル *pDst* の長さ。

説明

関数 ippsMahDistMultiMix は、ippsr.h ファイルで宣言される。この関数は、次のように、複数の平均値と分散値のペアに基づいて、単一の観測ベクトルについてマハラノビス距離を計算する。

D2 サフィックスが付いた関数の場合、

$$pDst[i] = \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2,$$

$0 \le i < height$

D2L サフィックスが付いた関数の場合、

$$pDst[i] = \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 \; , 0 \le i < height$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mMean*、*mVar*、*pMean*、*pVar*、または *pDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

LogGaussSingle

単一のガウスと 1 つの観測ベクトルについて
観測確率を計算する。

事例 1: 逆対角共分散行列の演算

```
IppStatus ippsLogGaussSingle_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s
    val, int scaleFactor);
IppStatus ippsLogGaussSingle_Scaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f
    val, int scaleFactor);
IppStatus ippsLogGaussSingle_32f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_32f64f(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f val);
```


intel®

```
IppStatus ippsLogGaussSingle_64f(const Ipp64f* pSrc, const Ipp64f*
    pMean, const Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_Low_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s
    val, int scaleFactor);
IppStatus ippsLogGaussSingle_LowScaled_16s32f(const Ipp16s* pSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f
    val, int scaleFactor);
```

事例 2: 対角共分散行列の演算

```
IppStatus ippsLogGaussSingle_DirectVar_16s32s_Sfs(const Ipp16s* pSrc,
    const Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32s* pResult,
    Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussSingle_DirectVarScaled_16s32f(const Ipp16s*
    pSrc, const Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32f*
    pResult, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussSingle_DirectVar_32f(const Ipp32f* pSrc, const
    Ipp32f* pMean, const Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f
    val);
IppStatus ippsLogGaussSingle_DirectVar_32f64f(const Ipp32f* pSrc, const
    Ipp32f* pMean, const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f
    val);
IppStatus ippsLogGaussSingle_DirectVar_64f(const Ipp64f* pSrc, const
    Ipp64f* pMean, const Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f
    val);
```

事例 3: 恒等共分散行列の演算

```
IppStatus ippsLogGaussSingle_IdVar_16s32s_Sfs(const Ipp16s* pSrc, const
    Ipp16s* pMean, int len, Ipp32s* pResult, Ipp32s val, int
    scaleFactor);
IppStatus ippsLogGaussSingle_IdVarScaled_16s32f(const Ipp16s* pSrc,
    const Ipp16s* pMean, int len, Ipp32f* pResult, Ipp32f val, int
    scaleFactor);
IppStatus ippsLogGaussSingle_IdVar_32f(const Ipp32f* pSrc, const
    Ipp32f* pMean, int len, Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_IdVar_32f64f(const Ipp32f* pSrc, const
    Ipp32f* pMean, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_IdVar_64f(const Ipp64f* pSrc, const
    Ipp64f* pMean, int len, Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_IdVarLow_16s32s_Sfs(const Ipp16s* pSrc,
    const Ipp16s* pMean, int len, Ipp32s* pResult, Ipp32s val, int
    scaleFactor);
IppStatus ippsLogGaussSingle_IdVarLowScaled_16s32f(const Ipp16s* pSrc,
    const Ipp16s* pMean, int len, Ipp32f* pResult, Ipp32f val, int
    scaleFactor);
```

事例 4: ブロック対角共分散行列の演算

```
IppStatus ippsLogGaussSingle_BlockDVar_16s32s_Sfs(const Ipp16s* pSrc,
    const Ipp16s* pMean, const IppsBlockDMatrix_16s * pBlockVar, int
    len, Ipp32s* pResult, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussSingle_BlockDVarScaled_16s32f(const Ipp16s*
    pSrc, const Ipp16s* pMean, const IppsBlockDMatrix_16s* pBlockVar,
    int len, Ipp32f* pResult, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussSingle_BlockDVar_32f(const Ipp32f* pSrc, const
    Ipp32f* pMean, const IppsBlockDMatrix_32f* pBlockVar, int len,
    Ipp32f* pResult, Ipp32f val);
IppStatus ippsLogGaussSingle_BlockDVar_32f64f(const Ipp32f* pSrc, const
    Ipp32f* pMean, const IppsBlockDMatrix_32f * pBlockVar, int len,
    Ipp64f* pResult, Ipp64f val);
IppStatus ippsLogGaussSingle_BlockDVar_64f(const Ipp64f* pSrc, const
    Ipp64f* pMean, const IppsBlockDMatrix_64f * pBlockVar, int len,
    Ipp64f* pResult, Ipp64f val);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pMean	平均ベクトル [*len*] へのポインタ。
pVar	分散ベクトル [*len*] へのポインタ。
pBlockVar	ブロック対角分散行列へのポインタ。
len	入力ベクトル、平均ベクトル、分散ベクトル内の要素の数。
pResult	結果へのポインタ。
val	ガウス定数。
scaleFactor	中間和のスケール係数。

説明

関数 ippsLogGaussSingle は、ippsr.h ファイルで宣言される。この関数は、単一の観測ベクトルと 1 つのガウス混合成分について観測確率を計算する。結果 *pResult* は対数表現になる。

DirectVar、IdVar、BlockDVar サフィックスが付かない関数の場合、共分散行列は逆対角行列と見なされる。

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

DirectVar サフィックスが付いた関数の場合、共分散行列は対角行列と見なされる。

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2 / pVar[i]$$

IdVar サフィックスが付いた関数の場合、共分散行列は恒等行列と見なされる。

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} (pSrc[i] - pMean[i])^2$$

BlockDVar サフィックスが付いた関数の場合、共分散行列はブロック対角行列と見なされる。

$$pResult[0] = val - \frac{1}{2} \cdot \sum_{i=0}^{len-1} \sum_{j=0}^{len-1} (pSrc[i] - pMean[i]) \cdot V[i][j] \cdot (pSrc[j] - pMean[j])$$

$V[i][j]$ はブロック対角行列 *pVar* の要素である。

整数スケーリングを実行する関数型（引数リストに *scaleFactor* 引数が含まれていることで区別される）は、出力される結果に対してではなく、上記の式の中間和に対して乗数 $2^{-scaleFactor}$ を適用する。

Low サフィックスが付いた関数は、下位の入力値のための関数であり、高速の計算を実行できる。16 ビット入力データが 12 ビットで表現され、入力ベクトルの長さが 128 より小さい場合は、これらの関数によって正しい結果が得られる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pMean*、*pVar*、*pBlockVar*、または *pResult* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

LogGauss

単一のガウスと複数の観測ベクトルについて
観測確率を計算する。

事例 1: 逆対角共分散行列の演算

```
IppStatus ippsLogGauss_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int
    height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGauss_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
    pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height,
    Ipp32s val, int scaleFactor);
IppStatus ippsLogGauss_Scaled_16s32f_D2(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst,
    int height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGauss_Scaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int
    height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGauss_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int
    height, Ipp32f val);
IppStatus ippsLogGauss_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int height,
    Ipp32f val);
IppStatus ippsLogGauss_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int
    height, Ipp64f val);
IppStatus ippsLogGauss_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int height,
    Ipp64f val);
IppStatus ippsLogGauss_Low_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pDst,
    int height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGauss_Low_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int
    height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGauss_LowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f*
    pDst, int height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGauss_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int
    height, Ipp32f val, int scaleFactor);
```


intel®

事例 2: 恒等共分散行列の演算

```
IppStatus ippsLogGauss_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s
    val, int scaleFactor);
IppStatus ippsLogGauss_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
    scaleFactor);
IppStatus ippsLogGauss_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, int width, Ipp32f* pDst, int height,
    Ipp32f val, int scaleFactor);
IppStatus ippsLogGauss_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f
    val, int scaleFactor);
IppStatus ippsLogGauss_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pMean, int width, Ipp32f* pDst, int height, Ipp32f val);
IppStatus ippsLogGauss_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pMean, int width, Ipp32f* pDst, int height, Ipp32f val);
IppStatus ippsLogGauss_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pMean, int width, Ipp64f* pDst, int height, Ipp64f val);
IppStatus ippsLogGauss_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pMean, int width, Ipp64f* pDst, int height, Ipp64f val);
IppStatus ippsLogGauss_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, int width, Ipp32s* pDst, int height,
    Ipp32s val, int scaleFactor);
IppStatus ippsLogGauss_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc,
    const Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s
    val, int scaleFactor);
IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, int width, Ipp32f* pDst, int height,
    Ipp32f val, int scaleFactor);
IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f
    val, int scaleFactor);
```

引数

pSrc	入力ベクトル [*height***step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
step	入力ベクトル *pSrc* の行のステップ。
pMean	平均ベクトル [*width*] へのポインタ。
pVar	分散ベクトル [*width*] へのポインタ。
width	平均ベクトルと分散ベクトルの長さ。

pDst	結果のベクトル [*height*] へのポインタ。
height	入力行列の行数と結果ベクトル *pDst* の長さ。
val	ガウス定数。
scaleFactor	中間和のスケール係数。

説明

関数 ippsLogGauss は、ippsr.h ファイルで宣言される。この関数は、1 つのガウス混合成分と複数の観測ベクトルについて観測確率を計算する。結果 *pResult* は対数表現になる。

IdVar サフィックスが付かない関数の場合、共分散行列は逆対角行列と見なされる。

D2 サフィックスが付いた関数の場合、

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 ,$$

$0 \le i < height$

D2L サフィックスが付いた関数の場合、

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 ,$$

$0 \le i < height$

IdVar サフィックスが付いた関数の場合、共分散行列は恒等行列と見なされる。

D2 サフィックスが付いた関数の場合、

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2 ,$$

$0 \le i < height$

D2L サフィックスが付いた関数の場合、

$$pDst[i] = val - 0.5 \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2 ,\ 0 \le i < height$$

整数スケーリングを実行する関数型（引数リストに *scaleFactor* 引数が含まれていることで区別される）は、出力される結果に対してではなく、上記の式の中間和に対して乗数 $2^{-scaleFactor}$ を適用する。

Low サフィックスが付いた関数は、下位の入力値のための関数であり、高速の計算を実行できる。16 ビット入力データが 12 ビットで表現され、入力ベクトルの長さが 128 より小さい場合は、これらの関数によって正しい結果が得られる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pMean*、*pVar*、または *pDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

LogGaussMultiMix

複数のガウス混合成分について観測確率を計算する。

```
IppStatus ippsLogGaussMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const
    Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32s*
    pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_16s32s_D2LSfs(const Ipp16s** mMean,
    const Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s*
    pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean,
    const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width,
    Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean,
    const Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32f*
    pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_32f_D2(const Ipp32f* pMean, const
    Ipp32f* pVar, int step, const Ipp32f* pSrc, int width, Ipp32f*
    pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_32f_D2L(const Ipp32f** mMean, const
    Ipp32f** mVar, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int
    height);
```

```
IppStatus ippsLogGaussMultiMix_64f_D2(const Ipp64f* pMean, const
    Ipp64f* pVar, int step, const Ipp64f* pSrc, int width, Ipp64f*
    pSrcDst, int height);
IppStatus ippsLogGaussMultiMix_64f_D2L(const Ipp64f** mMean, const
    Ipp64f** mVar, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int
    height);
IppStatus ippsLogGaussMultiMix_Low_16s32s_D2Sfs(const Ipp16s* pMean,
    const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width,
    Ipp32s* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_Low_16s32s_D2LSfs(const Ipp16s** mMean,
    const Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s*
    pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2(const Ipp16s* pSrc,
    int step, const Ipp16s* pMean, const Ipp16s* pVar, int width,
    Ipp32f* pDst, int height, int scaleFactor);
IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2L(const Ipp16s**
    mSrc, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f*
    pDst, int height, int scaleFactor);
```

引数

pMean	平均ベクトル [*height***step*] へのポインタ。
pVar	分散ベクトル [*height***step*] へのポインタ。
mMean	平均行列 [*height*][*width*] へのポインタ。
mVar	分散行列 [*height*][*width*] へのポインタ。
step	平均ベクトルおよび分散ベクトルの業のステップ。
pSrc	入力ベクトル [*width*] へのポインタ。
pSrcDst	入力およびデスティネーション・ベクトル [*height*] へのポインタ。
width	平均行列 *mMean* の列数。
height	平均行列 *mMean* の行数。
scaleFactor	中間和のスケール係数。

説明

関数 ippsLogGaussMultiMix は、ippsr.h ファイルで宣言される。この関数は、複数のガウス混合成分について観測確率を計算する。演算の結果は対数表現になる。計算は次のように行われる。

D2 サフィックスが付いた関数の場合、

$$pSrcDst[i] = pSrcDst[i] - 0.5 \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2$$

$0 \le i < height$

D2L サフィックスが付いた関数の場合、

$$pSrcDst[i] = pSrcDst[i] - 0.5 \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2 ,$$

$0 \le i < height$

整数スケーリングを実行する関数型（引数リストに *scaleFactor* 引数が含まれていることで区別される）は、出力される結果に対してではなく、上記の式の中間和に対して乗数 $2^{-scaleFactor}$ を適用する。

Low サフィックスが付いた関数は、下位の入力値のための関数であり、高速の計算を実行できる。16 ビット入力データが 12 ビットで表現され、入力ベクトルの長さが 128 より小さい場合は、これらの関数によって正しい結果が得られる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mMean*、*mVar*、*pMean*、*pVar*、または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

LogGaussMax

最大値演算を使用して、複数の観測ベクトルと 1 つのガウス混合成分に基づいて尤度確率を計算する。

事例 1: 逆対角共分散行列の演算

```
IppStatus ippsLogGaussMax_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s*
    pSrcDst, int height, Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLogGaussMax_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int
    height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_Scaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f*
    pDst, int height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussMax_Scaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
    height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussMax_32f_D2(const Ipp32f* pSrc, int step, const
    Ipp32f* pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int
    height, Ipp32f val);
IppStatus ippsLogGaussMax_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val);
IppStatus ippsLogGaussMax_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int
    height, Ipp64f val);
IppStatus ippsLogGaussMax_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height,
    Ipp64f val);
IppStatus ippsLogGaussMax_Low_16s32s_D2Sfs(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s*
    pSrcDst, int height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_Low_16s32s_D2LSfs(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int
    height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_LowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f*
    pSrcDst, int height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussMax_LowScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f*
    pSrcDst, int height, Ipp32f val, int scaleFactor);
```

事例 2: 恒等共分散行列の演算

```
IppStatus ippsLogGaussMax_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height,
    Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc,
    const Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height, Ipp32s
    val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);
```

```
IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f
    val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVar_32f_D2(const Ipp32f* pSrc, int step,
    const Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f
    val);
IppStatus ippsLogGaussMax_IdVar_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussMax_IdVar_64f_D2(const Ipp64f* pSrc, int step,
    const Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f
    val);
IppStatus ippsLogGaussMax_IdVar_64f_D2L(const Ipp64f** mSrc, const
    Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s*
    pSrcDst, int height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s*
    pSrcDst, int height, Ipp32s val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc,
    int step, const Ipp16s* pMean, const Ipp16s* pVar, int width,
    Ipp32f* pSrcDst, int height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2L(const Ipp16s**
    mSrc, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f*
    pSrcDst, int height, Ipp32f val, int scaleFactor);
```

引数

pSrc	観測ベクトル [*height**step*] へのポインタ。
mSrc	観測行列 [*height*][*width*] へのポインタ。
step	観測ベクトル *pSrc* の行のステップ。
pMean	平均ベクトル [*width*] へのポインタ。
pVar	分散ベクトル [*width*] へのポインタ。
width	平均ベクトルと分散ベクトルの長さ。
pSrcDst	尤度ベクトル [*height*] へのポインタ。
height	ベクトル *pSrcDst* の長さ。
val	ガウス定数。
scaleFactor	中間和のスケール係数。

説明

関数 ippsLogGaussMax は、ippsr.h ファイルで宣言される。この関数は、「最大値」演算を使用して、複数の観測ベクトルについて観測確率を計算し、得られた確率をベクトル *pSrcDst* に累算する。
IdVar サフィックスが付かない関数の場合、共分散行列は逆対角行列と見なされる。

D2 サフィックスが付いた関数の場合、

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) , \; 0 \le i < height$$

D2L サフィックスが付いた関数の場合、

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) , \; 0 \le i < height$$

IdVar サフィックスが付いた関数の場合、共分散行列は恒等行列と見なされる。

D2 サフィックスが付いた関数の場合、

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2 ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) , \; 0 \le i < height$$

D2L サフィックスが付いた関数の場合、

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2 ,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]) , \; 0 \le i < height$$

整数スケーリングを実行する関数型（引数リストに *scaleFactor* 引数が含まれていることで区別される）は、出力される結果に対してではなく、上記の式の中間和に対して乗数 $2^{-scaleFactor}$ を適用する。

Low サフィックスが付いた関数は、下位の入力値のための関数であり、高速の計算を実行できる。16 ビット入力データが 12 ビットで表現され、入力ベクトルの長さが 128 より小さい場合は、これらの関数によって正しい結果が得られる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pMean*、*pVar*、または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

LogGaussMaxMultiMix

最大値演算を使用して、複数のガウス混合成分について尤度確率を計算する。

```
IppStatus ippsLogGaussMaxMultiMix_16s32s_D2Sfs(const Ipp16s* pMean,
    const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const
    Ipp32s* pVal, Ipp32s* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_16s32s_D2LSfs(const Ipp16s** mMean,
    const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s*
    pVal, Ipp32s* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_Scaled_16s32f_D2(const Ipp16s*
    pMean, const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width,
    const Ipp32f* pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_Scaled_16s32f_D2L(const Ipp16s**
    mMean, const Ipp16s** mVar, const Ipp16s* pSrc, int width, const
    Ipp32f* pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_32f_D2(const Ipp32f* pMean, const
    Ipp32f* pVar, int step, const Ipp32f* pSrc, int width, const
    Ipp32f* pVal, Ipp32f* pSrcDst, int height);
IppStatus ippsLogGaussMaxMultiMix_32f_D2L(const Ipp32f** mMean, const
    Ipp32f** mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal,
    Ipp32f* pSrcDst, int height);
```

```
IppStatus ippsLogGaussMaxMultiMix_64f_D2(const Ipp64f* pMean, const
    Ipp64f* pVar, int step, const Ipp64f* pSrc, int width, const
    Ipp64f* pVal, Ipp64f* pSrcDst, int height);
IppStatus ippsLogGaussMaxMultiMix_64f_D2L(const Ipp64f** mMean, const
    Ipp64f** mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal,
    Ipp64f* pSrcDst, int height);
IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2Sfs(const Ipp16s*
    pMean, const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width,
    const Ipp32s* pVal, Ipp32s* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2LSfs(const Ipp16s**
    mMean, const Ipp16s** mVar, const Ipp16s* pSrc, int width, const
    Ipp32s* pVal, Ipp32s* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2(const Ipp16s*
    pMean, const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width,
    const Ipp32f* pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2L(const Ipp16s**
    mMean, const Ipp16s** mVar, const Ipp16s* pSrc, int width, const
    Ipp32f* pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
```

引数

pMean	平均ベクトル [*height***step*] へのポインタ。
pVar	分散ベクトル [*height***step*] へのポインタ。
mMean	平均行列 [*height*][*width*] へのポインタ。
mVar	分散行列 [*height*][*width*] へのポインタ。
step	平均ベクトルおよび分散ベクトルの業のステップ。
pSrc	入力ベクトル [*width*] へのポインタ。
width	平均ベクトルおよび分散行列の列数。
pVal	重み定数ベクトル [*height*] へのポインタ。
pSrcDst	尤度ベクトル [*height*] へのポインタ。
height	平均および分散行列の行数。
scaleFactor	中間和のスケール係数。

説明

関数 `ippsLogGaussMaxMultiMix` は、`ippsr.h` ファイルで宣言される。この関数は、「最大値」演算を使用して、複数のガウス混合成分について観測確率を計算し、得られた確率をベクトル *pSrcDst* に累算する。共分散行列は逆対角行列と見なされる。

D2 サフィックスが付いた関数の場合、

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]), \quad 0 \le i < height$$

D2L サフィックスが付いた関数の場合、

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2,$$

$$pSrcDst[i] = \max(pSrcDst[i], V[i]), \quad 0 \le i < height$$

整数スケーリングを実行する関数型（引数リストに *scaleFactor* 引数が含まれていることで区別される）は、出力される結果に対してではなく、上記の式の中間和に対して乗数 $2^{-scaleFactor}$ を適用する。

Low サフィックスが付いた関数は、下位の入力値のための関数であり、高速の計算を実行できる。16 ビット入力データが 12 ビットで表現され、入力ベクトルの長さが 128 より小さい場合は、これらの関数によって正しい結果が得られる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mMean*、*mVar*、*pMean*、*pVar*、*pVal* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

LogGaussAdd

複数の観測ベクトルについて尤度確率を計算する。

事例 1: 逆対角共分散行列の演算

```
IppStatus ippsLogGaussAdd_Scaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f*
    pSrcDst, int height, Ipp32f val, int scaleFactor);
```

```
IppStatus ippsLogGaussAdd_Scaled_16s32f_D2L(const Ipp16s** mSrc, const
    Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
    height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussAdd_32f_D2(const Ipp32f** pSrc, int step, const
    Ipp32f* pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int
    height, Ipp32f val);
IppStatus ippsLogGaussAdd_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val);
IppStatus ippsLogGaussAdd_64f_D2(const Ipp64f* pSrc, int step, const
    Ipp64f* pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int
    height, Ipp64f val);
IppStatus ippsLogGaussAdd_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
    pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height,
    Ipp64f val);
IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f*
    pSrcDst, int height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f*
    pSrcDst, int height, Ipp32f val, int scaleFactor);
```

事例 2: 恒等共分散行列の演算

```
IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int
    step, const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc,
    const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f
    val, int scaleFactor);
IppStatus ippsLogGaussAdd_IdVar_32f_D2(const Ipp32f* pSrc, int step,
    const Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f
    val);
IppStatus ippsLogGaussAdd_IdVar_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2(const Ipp64f* pSrc, int step,
    const Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f
    val);
IppStatus ippsLogGaussAdd_IdVar_64f_D2L(const Ipp64f** mSrc, const
    Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc,
    int step, const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int
    height, Ipp32f val, int scaleFactor);
IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2L(const Ipp16s**
    mSrc, const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height,
    Ipp32f val, int scaleFactor);
```

引数

pSrc	観測ベクトル [*height***step*] へのポインタ。
mSrc	観測行列 [*height*][*width*] へのポインタ。
step	観測ベクトル *pSrc* の行のステップ。
pMean	平均ベクトル [*width*] へのポインタ。
pVar	分散ベクトル [*width*] へのポインタ。
width	平均ベクトルと分散ベクトルの長さ。
pSrcDst	尤度ベクトル [*height*] へのポインタ。
height	観測ベクトルの数。
val	重み定数。
scaleFactor	中間和のスケール係数。

説明

関数 `ippsLogGaussAdd` は、`ippsr.h` ファイルで宣言される。この関数は、複数の観測ベクトルについて観測確率を計算し、得られた確率をベクトル *pSrcDst* に累算する。

`IdVar` サフィックスが付かない関数の場合、共分散行列は逆対角行列と見なされる。

`D2` サフィックスが付いた関数の場合、

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2 ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) , \; 0 \le i < height$$

`D2L` サフィックスが付いた関数の場合、

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2 ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) , \; 0 \le i < height$$

IdVar サフィックスが付いた関数の場合、共分散行列は恒等行列と見なされる。

D2 サフィックスが付いた関数の場合、

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[i \cdot step + j] - pMean[j])^2 ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) , 0 \le i < height$$

D2L サフィックスが付いた関数の場合、

$$V[i] = val - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[i][j] - pMean[j])^2 ,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}) , 0 \le i < height$$

整数スケーリングを実行する関数型（引数リストに *scaleFactor* 引数が含まれていることで区別される）は、出力される結果に対してではなく、上記の式の中間和に対して乗数 $2^{-scaleFactor}$ を適用する。

Low サフィックスが付いた関数は、下位の入力値のための関数であり、高速の計算を実行できる。16 ビット入力データが 12 ビットで表現され、入力ベクトルの長さが 128 より小さい場合は、これらの関数によって正しい結果が得られる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pMean*、*pVar*、または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

LogGaussAddMultiMix

特定のガウス混合について尤度確率を
計算する。

```
IppStatus ippsLogGaussAddMultiMix_Scaled_16s32f_D2(const Ipp16s*
    pMean, const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width,
    const Ipp32f pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_Scaled_16s32f_D2L(const Ipp16s**
    mMean, const Ipp16s** mVar, const Ipp16s* pSrc, int width, const
    Ipp32f* pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_32f_D2(const Ipp32f* pMean, const
    Ipp32f* pVar, int step, const Ipp32f* pSrc, int width, const
    Ipp32f* pVal, Ipp32f* pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_32f_D2L(const Ipp32f** mMean, const
    Ipp32f** mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal,
    Ipp32f* pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_64f_D2(const Ipp64f* pMean, const
    Ipp64f* pVar, int step, const Ipp64f* pSrc, int width, const
    Ipp64f* pVal, Ipp64f* pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_64f_D2L(const Ipp64f** mMean, const
    Ipp64f** mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal,
    Ipp64f* pSrcDst, int height);
IppStatus ippsLogGaussAddMultiMix_LowScaled_16s32f_D2(const Ipp16s*
    pMean, const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width,
    const Ipp32f* pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
IppStatus ippsLogGaussAddMultiMix_LowScaled_16s32f_D2L(const Ipp16s**
    mMean, const Ipp16s** mVar, const Ipp16s* pSrc, int width, const
    Ipp32f* pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
```

引数

pMean	平均ベクトル [*height***step*] へのポインタ。
pVar	分散ベクトル [*height***step*] へのポインタ。
mMean	平均行列 [*height*][*width*] へのポインタ。
mVar	分散行列 [*height*][*width*] へのポインタ。
step	平均ベクトルおよび分散ベクトルの行のステップ (*pMean* の要素単位)。
pSrc	入力ベクトル [*width*] へのポインタ。
width	平均および分散行列の長さ。

pVal	重み定数ベクトル [*height*] へのポインタ。
pSrcDst	尤度ベクトル [*height*] へのポインタ。
height	ガウス混合成分の数。
scaleFactor	中間和のスケール係数。

説明

関数 ippsLogGaussAddMultiMix は、ippsr.h ファイルで宣言される。この関数は、複数のガウス混合成分について観測確率を計算し、得られた確率をベクトル *pSrcDst* に累算する。共分散行列は逆対角行列と見なされる。

D2 サフィックスが付いた関数の場合、

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}),\ 0 \le i < height$$

D2L サフィックスが付いた関数の場合、

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2,$$

$$pSrcDst[i] = \ln(e^{pSrcDst[i]} + e^{V[i]}),\ 0 \le i < height$$

整数スケーリングを実行する関数型（引数リストに *scaleFactor* 引数が含まれていることで区別される）は、出力される結果に対してではなく、上記の式の中間和に対して乗数 $2^{-scaleFactor}$ を適用する。

Low サフィックスが付いた関数は、下位の入力値のための関数であり、高速の計算を実行できる。16 ビット入力データが 12 ビットで表現され、入力ベクトルの長さが 128 より小さい場合は、これらの関数によって正しい結果が得られる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mMean*、*mVar*、*pMean*、*pVar*、*pVal* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

LogGaussMixture

特定のガウス混合について尤度確率を
計算する。

事例 1: 逆対角共分散行列の演算

```
IppStatus ippsLogGaussMixture_Scaled_16s32f_D2(const Ipp16s* pSrc,
    const Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int
    width, const Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_Scaled_16s32f_D2L(const Ipp16s* pSrc,
    const Ipp16s** mMean, const Ipp16s** mVar, int height, int width,
    const Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_LowScaled_16s32f_D2(const Ipp16s* pSrc,
    const Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int
    width, const Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_LowScaled_16s32f_D2L(const Ipp16s* pSrc,
    const Ipp16s** mMean, const Ipp16s** mVar, int height, int width,
    const Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_32f_D2(const Ipp32f* pSrc, const Ipp32f*
    pMean, const Ipp32f* pVar, int height, int step, int width, const
    Ipp32f* pVal, Ipp32f* pResult);
IppStatus ippsLogGaussMixture_32f_D2L(const Ipp32f* pSrc, const
    Ipp32f** mMean, const Ipp32f** mVar, int height, int width, const
    Ipp32f* pVal, Ipp32f* pResult);
IppStatus ippsLogGaussMixture_64f_D2(const Ipp64f* pSrc, const Ipp64f*
    pMean, const Ipp64f* pVar, int height, int step, int width, const
    Ipp64f* pVal, Ipp64f* pResult);
IppStatus ippsLogGaussMixture_64f_D2L(const Ipp64f* pSrc, const
    Ipp64f** mMean, const Ipp64f** mVar, int height, int width, const
    Ipp64f* pVal, Ipp64f* pResult);
```

事例 2: 恒等共分散行列の演算

```
IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2(const Ipp16s*
    pSrc, const Ipp16s* pMean, int height, int step, int width, const
    Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2L(const Ipp16s*
    pSrc, const Ipp16s** mMean, int height, int width, const Ipp32f*
    pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2(const Ipp16s*
    pSrc, const Ipp16s* pMean, int height, int step, int width, const
    Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
```

```
IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2L(const Ipp16s*
    pSrc, const Ipp16s** mMean, int height, int width, const Ipp32f*
    pVal, Ipp32f* pResult, int scaleFactor);
IppStatus ippsLogGaussMixture_IdVar_32f_D2(const Ipp32f* pSrc, const
    Ipp32f* pMean, int height, int step, int width, const Ipp32f* pVal,
    Ipp32f* pResult);
IppStatus ippsLogGaussMixture_IdVar_32f_D2L(const Ipp32f* pSrc, const
    Ipp32f** mMean, int height, int width, const Ipp32f* pVal, Ipp32f*
    pResult);
IppStatus ippsLogGaussMixture_IdVar_64f_D2(const Ipp64f* pSrc, const
    Ipp64f* pMean,  int height, int step, int width, const Ipp64f*
    pVal, Ipp64f* pResult);
IppStatus ippsLogGaussMixture_IdVar_64f_D2L(const Ipp64f* pSrc, const
    Ipp64f** mMean, int height, int width, const Ipp64f* pVal, Ipp64f*
    pResult);
```

引数

pSrc	入力ベクトル [width] へのポインタ。
pMean	平均ベクトル [*height***step*] へのポインタ。
pVar	分散ベクトル [*height***step*] へのポインタ。
mMean	平均行列 [*height*][*width*] へのポインタ。
mVar	分散行列 [*height*][*width*] へのポインタ。
height	ガウス混合成分の数。
step	平均ベクトルおよび分散ベクトルの行のステップ (*pMean* の要素単位)。
width	平均ベクトルおよび分散行列の行の長さ、およびベクトル *pSrc* の長さ。
pVal	重み定数のベクトル [*height*] へのポインタ。
pResult	出力混合値へのポインタ。
scaleFactor	中間和のスケール係数。

説明

関数 ippsLogGaussMixture は、ippsr.h ファイルで宣言される。この関数は、特定のガウス混合について観測確率を計算し、その結果を *pResult* に返す。

IdVar サフィックスが付かない関数の場合、共分散行列は逆対角行列と見なされる。

intel®

D2 サフィックスが付いた関数の場合、

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} pVar[i \cdot step + j] \cdot (pSrc[j] - pMean[i \cdot step + j])^2,$$

($i = 0,... height - 1$ の場合)、および

$$pResult[0] = \ln \sum_{i=0}^{height-1} e^{V[i]}$$

D2L サフィックスが付いた関数の場合、

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (pSrc[j] - mMean[i][j])^2,$$

$i = 0,... height - 1$

IdVar サフィックスが付いた関数の場合、共分散行列は恒等行列と見なされる。

D2 サフィックスが付いた関数の場合、

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[j] - pMean[i \cdot step + j])^2,$$

($i = 0,... height - 1$ の場合)、および

$$pResult[0] = \ln \sum_{i=0}^{height-1} e^{V[i]}$$

D2L サフィックスが付いた関数の場合、

$$V[i] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[j] - mMean[i][j])^2, \quad i = 0,... height - 1$$

整数スケーリングを実行する関数型（引数リストに *scaleFactor* 引数が含まれていることで区別される）は、出力される結果に対してではなく、上記の式の中間和に対して乗数 $2^{-scaleFactor}$ を適用する。

Low サフィックスが付いた関数は、下位の入力値のための関数であり、高速の計算を実行できる。16 ビット入力データが 12 ビットで表現され、入力ベクトルの長さが 128 より小さい場合は、これらの関数によって正しい結果が得られる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pMean*、*mMean*、*pVar*、*mVar*、*pVal* または *pResult* が NULL。
ippStsSizeErr	エラー。*width* または *height* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

LogGaussMixtureSelect

ガウス分布選択を使用して、ガウス混合について尤度確率を計算する。

事例 1: 逆対角共分散行列の演算

```
IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2(const Ipp16s*
    pSrc, const Ipp16s* pMean, const Ipp16s* pVar, int step, int width,
    const Ipp32s* pVal, const Ipp8u* pSign, int height, Ipp32s*
    pResult, int frames, Ipp32f none, int scaleFactor);
IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2L(const Ipp16s**
    mSrc, const Ipp16s** mMean, const Ipp16s** mVar, int width, const
    Ipp32s* pVal, const Ipp8u* pSign, int height, Ipp32s* pResult, int
    frames, Ipp32f none, int scaleFactor);
IppStatus ippsLogGaussMixture_Select_32f_D2(const Ipp32f* pSrc, const
    Ipp32f* pMean, const Ipp32f* pVar, int step, int width, const
    Ipp32f* pVal, const Ipp8u* pSign, int height, Ipp32f* pResult, int
    frames, Ipp32f none);
IppStatus ippsLogGaussMixture_Select_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f** mMean, const Ipp32f** mVar, int width, const Ipp32f* pVal,
    const Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f
    none);
```

事例 2: 恒等共分散行列の演算

```
IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2(const
    Ipp16s* pSrc, const Ipp16s* pMean, int step, int width, const
    Ipp32s* pVal, const Ipp8u* pSign, int height, Ipp32s* pResult, int
    frames, Ipp32f none, int scaleFactor);
IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2L(const
    Ipp16s** mSrc, const Ipp16s** mMean, int width, const Ipp32s* pVal,
    const Ipp8u* pSign, int height, Ipp32s* pResult, int frames, Ipp32f
    none, int scaleFactor);
IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2(const Ipp32f* pSrc,
    const Ipp32f* pMean, int step, int width, const Ipp32f* pVal, const
    Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f
    none);
IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2L(const Ipp32f** mSrc,
    const Ipp32f** mMean, int width, const Ipp32f* pVal, const Ipp8u*
    pSign, int height, Ipp32f* pResult, int frames, Ipp32f none);
```

引数

pSrc	入力ベクトル [*frames**step*] へのポインタ。
pMean	平均ベクトル [*height**step*] へのポインタ。
pVar	分散ベクトル [*height**step*] へのポインタ。
mSrc	入力行列 [*frames*][*width*] へのポインタ。
mMean	平均行列 [*height*][*width*] へのポインタ。
mVar	分散行列 [*height*][*width*] へのポインタ。
pVal	重み定数のベクトル [*height*] へのポインタ。
pSign	ガウス計算符号のベクトル [*frames**(*height*+7)/8] へのポインタ（バイト単位）。
step	平均、分散、および入力ベクトルの行のステップ（*pMean* の要素単位）。
width	平均、分散、および入力行列の長さ。
height	ガウス混合成分の数。
pResult	出力混合値のベクトル [*frames*] へのポインタ。
frames	入力ベクトルの数。*pSign* 行の長さでもある。
none	ガウスが計算されなかった場合の結果値。
scaleFactor	中間和のスケール係数。

説明

関数 `ippsLogGaussMixture_Select` は、`ippsr.h` ファイルで宣言される。この関数は、特定のガウス混合とフレーム入力ベクトルについて観測確率を計算し、その結果を *`pResult`* ベクトルに返す。対応する符号の付いたガウスがゼロと等しい場合は計算されない。入力ベ クトルのガウスが計算されない場合、*`none`* 値が返される。

たとえば、$s(i,k)$ を *`pSign`* の i 行目の k 番目のビットとする。

`IdVar` サフィックスが付かない関数の場合、共分散行列は逆対角行列と見なされる。

`D2` サフィックスが付いた関数の場合、

$k = 0,\ldots$ *`frames`* - 1, $i = 0,\ldots$ *`height`* - 1、$s(i,k) \neq 0$、

および次の場合

$$pResult[k] = \begin{cases} none, & \text{if } s(i,k) = 0 \text{ for } i = 0,\ldots height-1 \\ \ln \sum_{\substack{i=0 \\ s(i,k)\neq 0}}^{height-1} e^{V[i,k]}, & \text{otherwise} \end{cases}$$

`D2L` サフィックスが付いた関数の場合、

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} mVar[i][j] \cdot (mSrc[k][j] - mMean[i][j])^2$$

$k = 0,\ldots$ *`frames`* - 1, $i = 0,\ldots$ *`height`* - 1 および $s(i,k) \neq 0$ の場合
`pResult` ベクトルは上記と同じ式で計算される。

`IdVar` サフィックスが付いた関数の場合、共分散行列は恒等行列と見なされる。

`D2` サフィックスが付いた関数の場合、

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (pSrc[k \cdot step + j] - pMean[i \cdot step + j])^2$$

$k = 0,\ldots$ *`frames`* - 1, $i = 0,\ldots$ *`height`* - 1 および $s(i,k) \neq 0$ の場合
`pResult` ベクトルは上記と同じ式で計算される。

`D2L` サフィックスが付いた関数の場合、

$$V[i,k] = pVal[i] - 0.5 \cdot \sum_{j=0}^{width-1} (mSrc[k][j] - mMean[i][j])^2$$

$k = 0,...$ *frames* - 1, $i = 0,...$ *height* - 1 および $s(i,k) \neq 0$ の場合 *pResult* ベクトルは上記と同じ式で計算される。

整数スケーリングを実行する関数型（引数リストに *scaleFactor* 引数が含まれていることで区別される）は、出力される結果に対してではなく、上記の式の中間和に対して乗数 2-*scaleFactor* を適用する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pMean*、*pVar*、*mSrc*、*mMean*、*mVar*、*pSign*、*pVal* または *pResult* が NULL。
ippStsSizeErr	エラー。*width*、*height*、または *frames* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。
ippStsNoGaussian	警告。入力ベクトルの 1 つでガウスが計算されなかった。

BuildSignTable

ガウス混合計算の符号テーブルを埋める。

```
IppStatus ippsBuildSignTable_8u1u(const Ipp32s* pIndx, int num, const
    Ipp8u** mShortlist, int clust, int width, int shift, Ipp8u* pSign,
    int frames, int comps);
IppStatus ippsBuildSignTable_Var_8u1u(const Ipp32s* pIndx, const int*
    pNum, const Ipp8u** mShortlist, int clust, int width, int shift,
    Ipp8u* pSign, int frames, int comps);
```

引数

pIndx	クラスタのインデックス・ベクトル（[*frames***num*] または *pNum* 要素の和）へのポインタ。
num	各入力ベクトルのクラスタの数。
pNum	クラスタ・ベクトル [*frames*] の数。

mShortlist	ショートリスト行列 [*clust***width*] へのポインタ(バイト単位)。
clust	ショートリスト・ベクトル内の行数(コードブック・サイズと等しい)。
width	ショートリスト行列の行の長さ(バイト単位)。
shift	ショートリスト・ベクトル行内の最初の要素変位(ビット単位)。
pSign	ガウス計算符号の出力ベクトル [*frames**(*comps*+7)/8] へのポインタ(バイト単位)。
frames	出力ベクトルの数(*pSign* ベクトルの行数)。
comps	ガウス混合成分の数(*pSign* ベクトルのビット列)。

説明

関数 ippsBuildSignTable は、ippsr.h ファイルで宣言される。これらの関数は、ガウス分布選択技術を使用してガウス混合計算用の符号テーブルをビルドする。符号テーブルは、ippsLogGaussMixture_Select 関数の入力引数として使用される。入力ベクトルは、1 つまたは複数のコードブック・クラスタを有効にできる。ショートリスト・テーブルの行はクラスタに対応し、列はガウスに対応する。行のサブ文字列の長さ *clust* は、混合成分でクラスタが有効であるかどうかを示す。

たとえば、ippBuildSignTable 関数では $l(i) = i \cdot num$ 、

ippBuildSignTable_Var 関数では、$l(i) = \sum_{j=0}^{i-1} pNum[j]$

とする($i = 0, \ldots$ *frames* - 1 の場合)。

また、c_k を *mShortlist* の k 番目の行のビット・サブ文字列とする。これには、ビット *shift*,...*shift+comps-1* が格納されている(k = 0,... *clust* - 1 の場合)。

次に、*pIndex* ベクトルの要素 l(i),...l(i+1)-1 は、i 番目の入力ベクトルの有効なクラスタ番号となる。*pSign* の i 番目の行は、サブ文字列の論理和としてセットされる。

$c_{l(i)}, \ldots c_{l(i+1)-1}$, $i = 0, \ldots$ *frames* - 1

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pIndx*、*pSign*、*mShortlist* または *pNum* が NULL。
ippStsSizeErr	エラー。*clust*、*width*、*frames*、*comps*、*num*、または *pNum* ベクトルの要素の 1 つがゼロ以下、または *shift* がゼロより小さい。
ippStsStrideErr	エラー。*width* が (*shift*+*comps*+7)/8 より小さい。
ippStsBadArgErr	エラー。*pIndx* ベクトルの要素の 1 つがゼロより小さい、または *clust* 以上。

FillShortlist_Row

ガウス分布選択の行方向ショートリスト・
テーブルを埋める。

```
IppStatus ippsFillShortlist_Row_1u(const Ipp32s* pIndx, int height, int
    num, Ipp8u** mShortlist, int clust, int width, int shift);
IppStatus ippsFillShortlist_RowVar_1u(const Ipp32s* pIndx, const int*
    pNum, int height, Ipp8u** mShortlist, int clust, int width, int
    shift);
```

引数

pIndx	クラスタのインデックス・ベクトル（[*num***clust*] または *pNum* 要素の和）へのポインタ。
height	ガウス混合成分の数。
num	各クラスタのガウス混合成分の数。
pNum	ガウス・ベクトル [*clust*] の数。
mShortlist	ショートリスト行列 [*clust***width*] へのポインタ（バイト単位）。
clust	ショートリスト行列の行数（コードブック・サイズと等しい）。
width	ショートリスト行列の行の長さ（バイト単位）。
shift	ショートリスト行列の行での最初の要素変位（ビット単位）。

説明

関数 ippsFillShortlist_Row は、ippsr.h ファイルで宣言される。この関数は、ショートリスト・テーブルを埋めて、各クラスタに対して 1 つ以上のガウス混合を含むテーブルを提供する。この処理は、ガウス混合の平均値で構成されたコードブックのクラスタ・セントロイドを量子化することで行われる。量子化した結果は、*pIndx* ベクトルに格納される。

ショートリスト・テーブル *mShortlist* に含まれたビットで、ガウス混合の平均値に最も近い 1 つ以上のクラスタ・セントロイドに対応するビットは 1 に設定される。ショートリストは、入力ベクトルがコードブックの k 番目のクラスタを有効にする場合、i 番目のガウスを計算するかどうかを示す要素を含むビット・テーブルである。行数はコードブックのサイズと等しい。列数はモデル内のガウスの総数と等しい。行のステップは、バイト境界から各行を開始するように選択されている。ショートリスト・テーブルは、定義後にゼロにする必要がある。混合成分には、順次インデックスがあり、*shift* 引数は、モデルに含まれた最初の混合成分のインデックス、*height* 引数は混合成分の数である。

たとえば、ippsFillShortlist_Row 関数では $l(k) = k \cdot num$ 、

ippsFillShortlist_RowVar 関数では、$l(k) = \sum_{j=0}^{k-1} pNum[j]$

そして、$c(k,i)$ は *mShortlist* の k 番目の行の $(shift + i)$ 番目のビットと等しい ($i = 0,... height - 1$, $k = 0,... clust - 1$ の場合)。

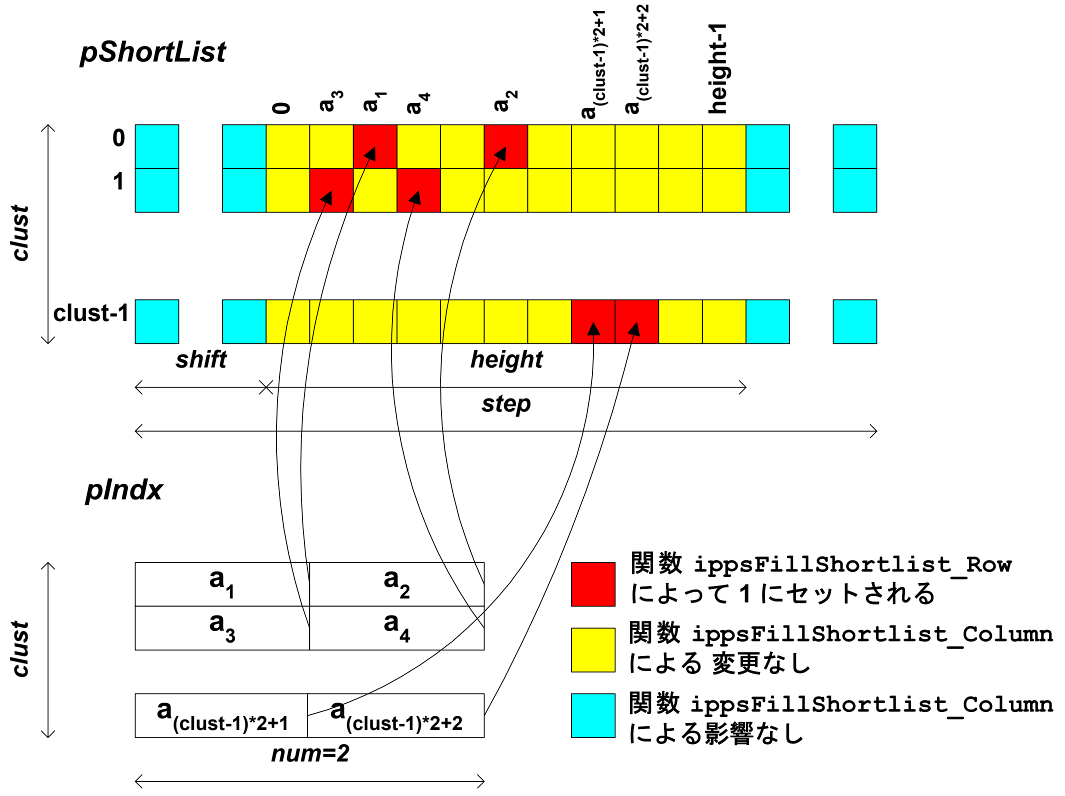
次に、ショートリスト・テーブルの *height* 列は次のように埋められる。

$c(k, pIndx[m]) = 1$, $m = l(k),...l(k+1)-1$, $k = 0,... clust - 1$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pIndx* または pShortlist が NULL。
ippStsSizeErr	エラー。*clust*、*height*、*width*、*num* または *pNum* ベクトルの要素の 1 つがゼロ以下、または *shift* がゼロより小さい。
ippStsStrideErr	エラー。*width* が (*shift*+*height*+7)/8 より小さい。
ippStsBadArgErr	エラー。*pIndx* ベクトルの要素の 1 つがゼロより小さい、または *height* 以上。

図 8-3 num=2 の場合の ippsFillShortlist_Row 関数の実行



FillShortlist_Column

ガウス分布選択の列方向ショートリスト・
テーブルを埋める。

```
IppStatus ippsFillShortlist_Column_1u(const Ipp32s* pIndx, int num,
    Ipp8u** mShortlist, int clust, int width, int shift, int height);
IppStatus ippsFillShortlist_ColumnVar_1u(const Ipp32s* pIndx, const
    int* pNum, Ipp8u** mShortlist, int clust, int width, int shift, int
    height);
```

引数

pIndx クラスタのインデックス・ベクトル（[*num**height*] または *pNum* 要素の和）へのポインタ。

height	ガウス混合成分の数。
num	各ガウス平均値のクラスタの数。
pNum	クラスタ・ベクトル [*height*] の数。
mShortlist	ショートリスト行列 [*clust*width*] へのポインタ（バイト単位）。
clust	ショートリスト行列の行数（コードブック・サイズと等しい）。
width	ショートリスト行列の行の長さ（バイト単位）。
shift	ショートリスト行列の行での最初の要素変位（ビット単位）。

説明

関数 ippsFillShortlist_Column は、ippsr.h ファイルで宣言される。この関数は、ガウス混合に対応するショートリスト・テーブルの一部を埋める。

ショートリストは、入力ベクトルがコードブックの k 番目のクラスタを有効にする場合、i 番目のガウスを計算するかどうかを示す要素を含むビット・テーブルである。行数はコードブックのサイズと等しい。列数はモデル内のガウスの総数と等しい。行のステップは、バイト境界 から各行を開始するように選択されている。ショートリスト・テーブルは、定義後にゼロにする必要がある。混合成分には、順次インデックスがあり、*shift* 引数は、モデルに含まれた最初の混合成分のインデックス、*height* 引数は混合成分の数である。

ippsVQSingle_Sort, VQSingle_Thresh 関数の 1 つを使用してガウス混合の平均値が量子化された後、その結果をショートリスト・テーブルに埋めることができる。混合成分の有効なクラスタに対応するビットは 1 にセットされる。

たとえば、ippsFillShortlist_Column 関数では $l(i) = i \cdot num$ 、

ippsFillShortlist_ColumnVar 関数では、$l(i) = \sum_{j=0}^{i-1} pNum[j]$

そして、$c(k,i)$ は *mShortlist* の k 番目の行の ($shift + i$) 番目のビットと等しい ($i = 0,\ldots height - 1,\ k = 0,\ldots clust - 1$ の場合)。

次に、ショートリスト・テーブルの *height* 列は次のように埋められる。

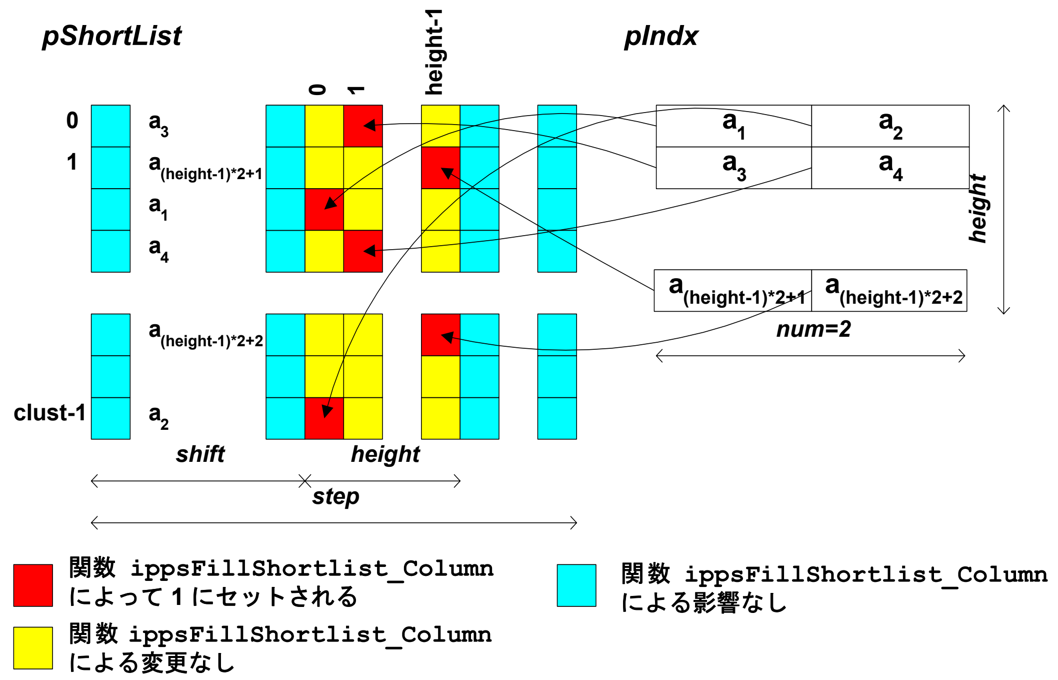
$c(pIndx[m], i) = 1\ ,\ m = l(i),\ldots l(i+1)-1\ ,\ i = 0,\ldots height - 1$

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pIndx* または pShortlist が NULL。
ippStsSizeErr	エラー。*clust*、*height*、*width*、*num* または *pNum* ベクトルの要素の 1 つがゼロ以下、または *shift* がゼロより小さい。
ippStsStrideErr	エラー。width が (shift+*height*+7)/8 より小さい。
ippStsBadArgErr	エラー。*pIndx* ベクトルの要素の 1 つがゼロより小さい、または *clust* 以上。

図 8-4　num=2 の場合の ippsFillShortlist_Column 関数の実行



DTW

動的タイム・ワープ・アルゴリズムを使用して、
観測ベクトルのシーケンスと基準ベクトルの
シーケンスの間の距離を計算する。

```
IppStatus ippsDTW_L2_8u32s_D2Sfs(const Ipp8u* pSrc1, int height1, const
    Ipp8u* pSrc2, int height2, int width, int step, Ipp32s* pDist, int
    delta, Ipp32s beam, int scaleFactor);
IppStatus ippsDTW_L2_8u32s_D2LSfs(const Ipp8u** mSrc1, int height1,
    const Ipp8u** mSrc2, int height2, int width, Ipp32s* pDist, int
    delta, Ipp32s beam, int scaleFactor);
IppStatus ippsDTW_L2Low_16s32s_D2Sfs(const Ipp16s* pSrc1, int height1,
    const Ipp16s* pSrc2, int height2, int width, int step, Ipp32s*
    pDist, int delta, Ipp32s beam, int scaleFactor);
IppStatus ippsDTW_L2Low_16s32s_D2LSfs(const Ipp16s** mSrc1, int
    height1, const Ipp16s** mSrc2, int height2, int width, Ipp32s*
    pDist, int delta, Ipp32s beam, int scaleFactor);
IppStatus ippsDTW_L2_32f_D2(const Ipp32f* pSrc1, int height1, const
    Ipp32f* pSrc2, int height2, int width, int step, Ipp32f* pDist, int
    delta, Ipp32f beam);
IppStatus ippsDTW_L2_32f_D2L(const Ipp32f** mSrc1, int height1, const
    Ipp32f** mSrc2, int height2, int width, Ipp32f* pDist, int delta,
    Ipp32f beam);
```

引数

pSrc1	1 番目の入力（観測）ベクトル **x** [*height1*step*] へのポインタ。
pSrc2	2 番目の入力（基準）ベクトル **y** [*height2*step*] へのポインタ。
mSrc1	1 番目の入力（観測）行列 **x** [height1][width] へのポインタ。
mSrc2	2 番目の入力（基準）行列 **y** [height2][width] へのポインタ
height1	1 番目の入力行列（*N1*）の行数。
height2	2 番目の入力行列（*N2*）の行数。
width	入力行列の行（*M*）の長さ。
step	*pSrc1* および *pSrc2* の行のステップ。

pDist	距離の値へのポインタ。
beam	ビーム値（正の場合に使用)。
delta	終点の制約条件値。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDTW は、ippsr.h ファイルで宣言される。この関数は、動的タイム・ワープ（DTW）の原理を利用して、2 つのベクトル・シーケンスの間の距離を計算する。計算は、次のように実行される。

$$pDist[0] = \min_{W} \sum_{i=0}^{N1-1} \left| \boldsymbol{x}_i - \boldsymbol{y}_{W(i)} \right| ,$$

$0 \le W(0) \le \delta$, $0 \le W(i) - W(i-1) \le 2$, $i = 1, \ldots N1-1$, $N2 - \delta \le W(N1-1) \le N2-1$ の場合。

この関数は、ベクトル *a* と *b* の間の距離を計算するために、次の式から得られる L2 ノルムを使用する。

$$|a-b| = \sqrt{\sum_{j=0}^{M-1} (a(j) - b(j))^2}$$

beam の値が正の場合は、ビームの外側にある（すなわち、以下の条件を満たす）パス *V* は削除される。

$$\sum_{i=0}^{k} \left| \boldsymbol{x}_i - \boldsymbol{y}_{V(i)} \right| > \min_{W} \sum_{i=0}^{k} \left| \boldsymbol{x}_i - \boldsymbol{y}_{W(i)} \right| - beam \ , \ k = 0, \ldots N1-1$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、*mSrc1*、*mSrc2* または *pDist* が NULL。
ippStsSizeErr	エラー。*height1*、*height2* または *width* がゼロ以下、あるいは *delta* がゼロより小さいか *height2* より大きい。

ippStsStrideErr	エラー。*step* が *width* より小さい。
ippStsNoOperation	*height1*、*height2*、*delta* の値に対して許容されるパスが存在する。

モデル推定

この節では、音響モデルと言語モデルのパラメータの推定に必要な関数について説明する。

MeanColumn

列要素の平均値を計算する。

```
IppStatus ippsMeanColumn_16s_D2(const Ipp16s* pSrc, int height, int
    step, Ipp16s* pDstMean, int width);
IppStatus ippsMeanColumn_16s_D2L(const Ipp16s** mSrc, int height,
    Ipp16s* pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2(const Ipp32f* pSrc, int height, int
    step, Ipp32f* pDstMean, int width);
IppStatus ippsMeanColumn_32f_D2L(const Ipp32f** mSrc, int height,
    Ipp32f* pDstMean, int width);
```

引数

pSrc	入力ベクトル [*height*step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
height	入力行列 *mSrc* の行数。
step	入力行列の行のステップ（*pSrc* の要素単位）。
pDstMean	出力平均ベクトル [*width*] へのポインタ。
width	入力行列 *mSrc* の列数、および出力平均ベクトル *pDstMean* の長さ。

説明

関数 ippsMeanColumn は、ippsr.h ファイルで宣言される。この関数は、入力行列の列要素の平均値を次のように計算する。

D2 サフィックスが付いた関数の場合、

$$pDstMean[j] = \frac{1}{height}\sum_{i=1}^{height-1} pSrc[i \cdot step + j]\ ,\ 0 \le j < width$$

D2L サフィックスが付いた関数の場合、

$$pDstMean[j] = \frac{1}{height}\sum_{i=1}^{height-1} mSrc[i][j],\ 0 \le j < width$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、または *pDstMean* が NULL。
ippStsSizeErr	エラー。*height* または *width* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

VarColumn

列要素の分散値を計算する。

```
IppStatus ippsVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int
    step, Ipp16s* pSrcMean, Ipp16s* pDstVar, int width, int
    scaleFactor);
IppStatus ippsVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height,
    Ipp16s* pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsVarColumn_32f_D2(const Ipp32f* pSrc, int height, int
    step, Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);
IppStatus ippsVarColumn_32f_D2L(const Ipp32f** mSrc, int height,
    Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);
```

引数

pSrc	入力ベクトル [*height***step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
height	入力行列 *mSrc* の行数。
step	入力ベクトル *pSrc* の行のステップ。

pSrcMean	入力平均ベクトル [*width*] へのポインタ。
pDstVar	出力分散ベクトル [*width*] へのポインタ。
width	入力行列 *mSrc* の列数、および入力平均ベクトル *pSrcMean* と出力分散ベクトル *pDstVar* の長さ。
scaleFactor	スケール係数。第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsVarColumn は、ippsr.h ファイルで宣言される。この関数は、行列の列要素の分散値を次のように計算する。
D2 サフィックスが付いた関数の場合、

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} pSrc[i \cdot step + j]^2}{height - 1},$$

$0 \le j < width$

D2L サフィックスが付いた関数の場合、

$$pDstVar[j] = \frac{-height \cdot (pSrcMean[j])^2 + \sum_{i=1}^{height-1} mSrc[i][j]^2}{height - 1},$$

$0 \le j < width$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pSrcMean* または *pDstVar* が NULL。
ippStsSizeErr	エラー。*width* がゼロ以下または *height* が 1 以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

MeanVarColumn

行列の列要素の平均値および分散値を
計算する。

```
IppStatus ippsMeanVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height,
    int step, Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int
    scaleFactor);
IppStatus ippsMeanVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height,
    Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);
IppStatus ippsMeanVarColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int
    height, int step, Ipp16s* pDstMean, Ipp32s* pDstVar, int width, int
    scaleFactor);
IppStatus ippsMeanVarColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int
    height, Ipp16s* pDstMean, Ipp32s* pDstVar, int width, int
    scaleFactor);
IppStatus ippsMeanVarColumn_32f_D2(const Ipp32f* pSrc, int height, int
    step, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
IppStatus ippsMeanVarColumn_32f_D2L(const Ipp32f** mSrc, int height,
    Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
```

引数

pSrc	入力ベクトル [*height*step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
height	入力行列 *mSrc* の行数。
step	入力ベクトル *pSrc* の行のステップ。
pDstMean	出力平均ベクトル [*width*] へのポインタ。
pDstVar	出力分散ベクトル [*width*] へのポインタ。
width	入力行列 *mSrc* の列数、および出力平均ベクトル *pDstMean* と分散ベクトル *pDstVar* の長さ。
scaleFactor	スケール係数。第 2 章の「整数のスケーリング」を参照。*scaleFactor* 引数（および整数のスケーリング）は、*pDstVar* に対してのみ使用される。*pDstMean* の要素はスケーリングされない。

説明

関数 ippsMeanVarColumn は、ippsr.h ファイルで宣言される。この関数は、入力行列の列要素の平均値と分散値の両方を計算する。計算の詳細は、関数 ippsMeanColumn と ippsVarColumn を参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pDstMean* または *pDstVar* が NULL。
ippStsSizeErr	エラー。*width* がゼロ以下または *height* が 1 以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

WeightedMeanColumn

列要素の加重平均値を計算する。

```
IppStatus ippsWeightedMeanColumn_32f_D2(const Ipp32f* pSrc, int step,
    const Ipp32f* pWgt, int height, Ipp32f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f* pWgt, int height, Ipp32f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_64f_D2(const Ipp64f* pSrc, int step,
    const Ipp64f* pWgt, int height, Ipp64f* pDstMean, int width);
IppStatus ippsWeightedMeanColumn_64f_D2L(const Ipp64f** mSrc, const
    Ipp64f* pWgt, int height, Ipp64f* pDstMean, int width);
```

引数

pSrc	入力ベクトル [*height***step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
pWgt	重みベクトル [*height*] へのポインタ。
height	入力行列の行数。
step	入力行列の行のステップ（*pSrc* の要素単位）。
pDstMean	出力平均ベクトル [*width*] へのポインタ。
width	入力行列の列数、および出力平均ベクトル *pDstMean* の長さ。

説明

関数 ippsWeightedMeanColumn は、ippsr.h ファイルで宣言される。この関数は、入力行列の列要素の加重平均値を次のように計算する。
D2 サフィックスが付いた関数の場合、

$$pDstMean[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot pSrc[i \cdot step + j] \quad , 0 \le j < width$$

D2L サフィックスが付いた関数の場合、

$$pDstMean[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot mSrc[i][j] \quad , 0 \le j < width$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pWgt* または *pDstMean* が NULL。
ippStsSizeErr	エラー。*height* または *width* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

WeightedVarColumn

列要素の加重分散値を 計算する。

```
IppStatus ippsWeightedVarColumn_32f_D2(const Ipp32f* pSrc, int step,
    const Ipp32f* pWgt, int height, const Ipp32f* pSrcMean, Ipp32f*
    pDstVar, int width);
IppStatus ippsWeightedVarColumn_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f* pWgt, int height, const Ipp32f* pSrcMean, Ipp32f* pDstVar,
    int width);
IppStatus ippsWeightedVarColumn_64f_D2(const Ipp64f* pSrc, int step,
    const Ipp64f* pWgt, int height, const Ipp64f* pSrcMean, Ipp64f*
    pDstVar, int width);
IppStatus ippsWeightedVarColumn_64f_D2L(const Ipp64f** mSrc, const
    Ipp64f* pWgt, int height, const Ipp64f* pSrcMean, Ipp64f* pDstVar,
    int width);
```

引数

pSrc	入力ベクトル [*height***step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
pWgt	重みベクトル [*height*] へのポインタ。
height	入力行列の行数。
step	入力行列の行のステップ（*pSrc* の要素単位）。
pSrcMean	入力平均ベクトル [*width*] へのポインタ。
pDstVar	出力分散ベクトル [*width*] へのポインタ。
width	入力行列の列数、および入力平均ベクトル *pSrcMean* と出力分散ベクトル *pDstVar* の長さ。

説明

関数 ippsWeightedVarColumn は、ippsr.h ファイルで宣言される。この関数は、入力行列の列要素の加重分散値を次のように計算する。
D2 サフィックスが付いた関数の場合、

$$pDstVar[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot pSrc[i \cdot step + j]^2 - pSrcMean[j]^2 \quad ,$$

$$0 \le j < width$$

D2L サフィックスが付いた関数の場合、

$$pDstVar[j] = \sum_{i=0}^{height-1} pWgt[i] \cdot mSrc[i][j]^2 - pSrcMean[j]^2 \quad , 0 \le j < width$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pWgt*、*pSrcMean* または *pDstVar* が NULL。
ippStsSizeErr	エラー。*height* または *width* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

WeightedMeanVarColumn

列要素の加重平均値と加重分散値を
計算する。

```
IppStatus ippsWeightedMeanVarColumn_32f_D2(const Ipp32f* pSrc, int
    step, const Ipp32f* pWgt, int height, Ipp32f* pDstMean, Ipp32f*
    pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_32f_D2L(const Ipp32f** mSrc, const
    Ipp32f* pWgt, int height, Ipp32f* pDstMean, Ipp32f* pDstVar, int
    width);
IppStatus ippsWeightedMeanVarColumn_64f_D2(const Ipp64f* pSrc, int
    step, const Ipp64f* pWgt, int height, Ipp64f* pDstMean, Ipp64f*
    pDstVar, int width);
IppStatus ippsWeightedMeanVarColumn_64f_D2L(const Ipp64f** mSrc, const
    Ipp64f* pWgt, int height, Ipp64f* pDstMean, Ipp64f* pDstVar, int
    width);
```

引数

pSrc	入力ベクトル [*height***step*] へのポインタ。
mSrc	入力行列 [*height*][*width*] へのポインタ。
pWgt	重みベクトル [*height*] へのポインタ。
height	入力行列の行数。
step	入力行列の行のステップ（*pSrc* の要素単位）。
pDstMean	出力平均ベクトル [*width*] へのポインタ。
pDstVar	出力分散ベクトル [*width*] へのポインタ。
width	入力行列の列数、および出力平均ベクトル *pSrcMean* と出力分散ベクトル *pDstVar* の長さ。

説明

関数 ippsWeightedMeanVarColumn は、ippsr.h ファイルで宣言される。この関数は、入力行列の列要素の加重平均値と加重分散値の両方を計算する。計算の詳細は、関数 ippsWeightedMeanColumn と ippsWeightedVarColumn を参照のこと。

以下のコード例は、関数 ippsWeightedMeanVarColumn の使用方法を示している。

例 8-2　　重み、平均値、分散値の EM アルゴリズムによる再評価

```
/* Input:  int height;                 // mixture components number
           int width;                  // observation space dimension
          int step;            // row step for mean, var and obs (step>=width)
           int num;                    // observations number
           int step1;                  // row step for gamma (step1>=num)
           float obs [num*step}        // observation vectors
   Update: float weight[height]        // Gaussian weights
           float mean[height*step]     // Gaussian mean vectors
           float var [height*step]     // Gaussian variance vectors
   Output: float result;               // probability logarithms sum */
{
   float gamma [height*step1] // gamma matrix
   float gammaT [height]      // gamma sums vector
   int k; float sum, sumGamma;
   /* adjust determinants for probability calculation */
   ippsLn_32f_I(weight,height);
   for (k=0; k<height; k++) {
      ippsSumLn_32f(var+k*step,width,&sum);
      weight[k]+=0.5f*(sum-width*log(2.0*3.1415926));
   }
   /* invert variances for probability calculation */
   ippsDivCRev_32f_I(var,height*step);
   /* logarithm of weighted Gaussian probabilities */
   ippsLogGauss_32f_D2(obs,step,mean,var,width,gamma,num,weight[0]);
   ippsCopy_32f(gamma,probs,num);
   for (k=1; k<height; k++) {
      ippsLogGauss_32f_D2(obs,step,mean+k*step,var+k*step,width,
                     gamma+k*step1,num,weight[k]);
      ippsLogAdd_32f(gamma+k*step1,probs,num,ippAlgHintNone);
   }
  ippsSum_32f(probs,num,&result,ippAlgHintNone);
   /* gamma matrix and sum calculation */
   ippsExp_32f_I(gamma,height*step1);
```

例 8-2　重み、平均値、分散値の EM アルゴリズムによる再評価

```
        ippsSumRow_32f_D2(gamma,height,step1,gammaT,num);
       ippsSum_32f(gammaT,height,&sumGamma,ippAlgHintNone);
       /* weights update */
       ippsDivC_32f(gammaT,sumGamma,weight,height);
       /* means and variances update */
       for (k=0; k<height; k++) {
          ippsDivC_32f_I(gammaT[k],gamma+k*step1,num);
          ippsWeightedMeanVarColumn_32f_D2(obs,step,gamma+k*step1,num,
                            mean+k*step,var+k*step,width);
       }
}
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*mSrc*、*pWgt*、*pDstMean* または *pDstVar* が NULL。
ippStsSizeErr	エラー。*height* または *width* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

NormalizeColumn

列の平均値と分散値に基づいて、
行列の列を正規化する。

```
IppStatus ippsNormalizeColumn_16s_D2Sfs(Ipp16s* pSrcDst, int step, int
    height, const Ipp16s* pMean, const Ipp16s* pVar, int width, int
    scaleFactor);
IppStatus ippsNormalizeColumn_16s_D2LSfs(Ipp16s** mSrcDst, int height,
    const Ipp16s* pMean, const Ipp16s* pVar, int width, int
    scaleFactor);
IppStatus ippsNormalizeColumn_32f_D2(Ipp32f* pSrcDst, int step, int
    height, const Ipp32f* pMean, const Ipp32f* pVar, int width);
IppStatus ippsNormalizeColumn_32f_D2L(Ipp32f** mSrcDst, int height,
    const Ipp32f* pMean, const Ipp32f* pVar, int width);
```

引数

pSrcDst	入力および出力ベクトル [*height***step*] へのポインタ。
mSrcDst	入力および出力行列 [*height*][*width*] へのポインタ。
pMean	列の平均ベクトル [*width*] へのポインタ。
pVar	列の分散ベクトル [*width*] へのポインタ。
width	平均ベクトルと分散ベクトルの長さ。
step	入力および出力ベクトル *pSrcDst* の行のステップ。
height	入力および出力行列 *mSrcDst* の行数。

説明

関数 ippsNormalizeColumn は、ippsr.h ファイルで宣言される。この関数は、行列 *mSrcDst* の列を次のように正規化する。

D2 サフィックスが付いた関数の場合、

$$pSrcDst[i \cdot step + j] = (pSrcDst[i \cdot step + j] - pMean[j]) \cdot pVar[j] \quad ,$$

$0 \le i < height,\ 0 \le j < width$

D2L サフィックスが付いた関数の場合、

$$mSrcDst[i][j] = (mSrcDst[i][j] - pMean[j]) \cdot pVar[j] \ ,$$

$0 \le i < height,\ 0 \le j < width$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst*、*mSrcDst*、*pMean* または *pVar* が NULL。
ippStsSizeErr	エラー。*height* または *width* がゼロ以下。
ippStsStrideErr	エラー。*step* が *width* より小さい。

NormalizeInRange

入力ベクトルの要素を正規化し、
スケーリングする。

```
IppStatus ippsNormalizeInRange_16s8u(const Ipp16s* pSrc, Ipp8u* pDst,
    int len, Ipp32f lowCut, Ipp32f highCut, Ipp8u range);
IppStatus ippsNormalizeInRange_16s(const Ipp16s* pSrc, Ipp16s* pDst,
    int len, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRange_16s_I(Ipp16s* pSrcDst, int len, Ipp32f
    lowCut, Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRange_32f8u(const Ipp32f* pSrc, Ipp8u* pDst,
    int len, Ipp32f lowCut, Ipp32f highCut, Ipp8u range);
IppStatus ippsNormalizeInRange_32f16s(const Ipp32f* pSrc, Ipp16s* pDst,
    int len, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);
IppStatus ippsNormalizeInRange_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    int len, Ipp32f lowCut, Ipp32f highCut, Ipp32f range);
IppStatus ippsNormalizeInRange_32f_I(Ipp32f* pSrcDst, int len, Ipp32f
    lowCut, Ipp32f highCut, Ipp32f range);
IppStatus ippsNormalizeInRangeMinMax_16s8u(const Ipp16s* pSrc, Ipp8u*
    pDst, int len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f
    highCut, Ipp8u range);
IppStatus ippsNormalizeInRangeMinMax_16s(const Ipp16s* pSrc, Ipp16s*
    pDst, int len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f
    highCut, Ipp16s range);
IppStatus ippsNormalizeInRangeMinMax_16s_I(Ipp16s* pSrcDst, int len,
    Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s
    range);
IppStatus ippsNormalizeInRangeMinMax_32f8u(const Ipp32f* pSrc, Ipp8u*
    pDst, int len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f
    highCut, Ipp8u range);
IppStatus ippsNormalizeInRangeMinMax_32f16s(const Ipp32f* pSrc, Ipp16s*
    pDst, int len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f
    highCut, Ipp16s range);
IppStatus ippsNormalizeInRangeMinMax_32f(const Ipp32f* pSrc, Ipp32f*
    pDst, int len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f
    highCut, Ipp32f range);
IppStatus ippsNormalizeInRangeMinMax_32f_I(Ipp32f* pSrcDst, int len,
    Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp32f
    range);
```

引数

pSrc	入力配列 [*len*] へのポインタ。
pSrcDst	インプレース演算用の入力および出力配列 [*len*] へのポインタ。
pDst	出力配列 [*len*] へのポインタ。
len	入力および出力配列の要素の数。
lowCut	下位の切り捨て値。
highCut	上位の切り捨て値。
range	出力データ値の上限（下限は 0）。
valMin	入力データの最小値。
valMax	入力データの最大値。

説明

関数 ippsNormalizeInRange は、ippsr.h ファイルで宣言される。この関数は、下位および上位の切り捨て値（$0 \le lowCut < highCut \le 1$）を適用して入力ベクトルの要素を 0 ～ 1 の範囲に正規化し、それを *range* の値でスケーリングする。

入力値（非インプレース演算の場合は *pSrc*[*k*]、インプレース演算の場合は *pSrcDst*[*k*]）を x_k で表すと、出力値 y_k（非インプレース演算の場合は *pDst*[*k*] に書き込まれ、インプレース演算の場合は *pSrcDst*[*k*] に書き込まれる）は、次の式に従って計算される。

$$y_k = \max\left(0\,,\ \min\left(range\,,\ \frac{\dfrac{x_k - x_{min}}{x_{max} - x_{min}} - lowCut}{highCut - lowCut} \cdot range\right)\right)\ ,\ k = 0,\ldots len - 1\ ,$$

ここで、関数 ippsNormalizeInRange の場合、$x_{min} = \min_{i=0,\ldots len-1} x_i$ 、

$x_{max} = \max_{i=0,\ldots len-1} x_i$

および、

関数 ippsNormalizeInRangeMinMax の場合、$x_{min} = valMin$ 、$x_{max} = valMax$

この関数は、スペクトログラム・データの生成に使用できる。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsBadArgErr	エラー。条件 $0 \le lowCut < highCut \le 1$ を満たしていないか、*range* がゼロより小さいか、*valMin* が *valMax* より大きい。
ippStsInvZero	$x_{min} = x_{max}$ という警告。出力ベクトルのすべての要素がゼロに設定されている。

MeanVarAcc

平均と分散の再評価のために推定値を累算する。

```
IppStatus ippsMeanVarAcc_32f(Ipp32f const* pSrc, Ipp32f const*
    pSrcMean, Ipp32f* pDstMeanAcc, Ipp32f* pDstVarAcc, int len, Ipp32f
    val);
IppStatus ippsMeanVarAcc_64f(Ipp64f const* pSrc, Ipp64f const*
    pSrcMean, Ipp64f* pDstMeanAcc, Ipp64f* pDstVarAcc, int len, Ipp64f
    val);
```

引数

pSrc	観測ベクトル [*len*] へのポインタ。
pSrcMean	古い平均ベクトル [*len*] へのポインタ。
pDstMeanAcc	平均アキュムレータ [*len*] へのポインタ。
pDstVarAcc	分散アキュムレータ [*len*] へのポインタ。
len	観測ベクトルの長さ。
val	再評価の定数値。

説明

関数 ippsMeanVarAcc は、ippsr.h ファイルで宣言される。この関数は、フォワードバックワード・アルゴリズムによる平均と分散の再評価のために推定値を累算する。次のようにする。

$$pDstMeanAcc[i] = pDstMeanAcc[i] + val \cdot (pSrc[i] - pSrcMean[i]),$$

$$pDstVarAcc[i] = pDstVarAcc[i] + val \cdot (pSrc[i] - pSrcMean[i])^2,$$

$$0 \le i < len$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pSrcMean*、*pDstMeanAcc* または *pDstVarAcc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

GaussianDist

2 つのガウス間の距離を計算する。

```
IppStatus ippsGaussianDist_32f(const Ipp32f* pMean1, const Ipp32f*
    pVar1, const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f*
    pResult, Ipp32f wgt1, Ipp32f det1, Ipp32f wgt2, Ipp32f det2);
IppStatus ippsGaussianDist_64f(const Ipp64f* pMean1, const Ipp64f*
    pVar1, const Ipp64f* pMean2, const Ipp64f* pVar2, int len, Ipp64f*
    pResult, Ipp64f wgt1, Ipp64f det1, Ipp64f wgt2, Ipp64f det2);
```

引数

pMean1	1 番目のガウス [*len*] の平均ベクトルへのポインタ。
pVar1	1 番目のガウス [*len*] の分散ベクトルへのポインタ。
pMean2	2 番目のガウス [*len*] の平均ベクトルへのポインタ。
pVar2	2 番目のガウス [*len*] の平均ベクトルへのポインタ。
len	平均ベクトルと分散ベクトルの長さ。
pResult	距離の値へのポインタ。
wgt1	1 番目のガウスの重さ。
det1	1 番目のガウスの行列式（対数表現）。
wgt2	2 番目のガウスの重さ。
det2	2 番目のガウスの行列式（対数表現）。

説明

関数 ippsGaussianDist は、ippsr.h ファイルで宣言される。この関数は、2 つのガウス間の距離を計算する。

$$pResult[0] = (wgt1 \cdot det1) + (wgt2 \cdot det2) - (wgt1 + wgt2) \cdot (len \cdot \ln(2\pi) - \ln(V)) ,$$

ここで、

$$V = \prod_{i=0}^{len-1} \frac{W_1[i] + W_2[i] - (wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i])^2}{wgt1 + wgt2} ,$$

また、次の式が使用される。

$$W_1[i] = wgt1 \cdot (pVar1[i] + pMean1[i]^2) ,$$

$$W_2[i] = wgt2 \cdot (pVar2[i] + pMean2[i]^2)$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMean1*、*pMean2*、*pVar1*、*pVar2* または *pResult* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

GaussianSplit

単一のガウス成分を、同じ分散を持つ 2 つの成分に分割する。

```
IppStatus ippsGaussianSplit_32f(Ipp32f* pMean1, Ipp32f* pVar1, Ipp32f*
    pMean2, Ipp32f* pVar2, int len, Ipp32f val);
IppStatus ippsGaussianSplit_64f(Ipp64f* pMean1, Ipp64f* pVar1, Ipp64f*
    pMean2, Ipp64f* pVar2, int len, Ipp64f val);
```

引数

pMean1	入力ガウス平均ベクトルと、分割後の 1 番目のガウスの平均ベクトル [*len*] へのポインタ。
pVar1	入力ガウス分散ベクトルと、分割後の 1 番目のガウスの分散ベクトル [*len*] へのポインタ。
pMean2	分割後の 2 番目のガウスの平均ベクトル [*len*] へのポインタ。
pVar2	分割後の 2 番目のガウスの分散ベクトル [*len*] へのポインタ。
len	平均ベクトルと分散ベクトルの長さ。

val	分散摂動値。

説明

関数 ippsGaussianSplit は、ippsr.h ファイルで宣言される。この関数は、次のようにガウス成分を 2 つのガウス混合成分に分割する。

$$pMean2[i] = pMean1[i] - val \cdot \sqrt{pVar1[i]}\ ,$$

$$pMean1[i] = pMean1[i] + val \cdot \sqrt{pVar1[i]}\ ,$$

$$pVar2[i] = pVar1[i]\ ,$$

$$0 \le i < len$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMean1*、*pMean2*、*pVar1* または *pVar2* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

GaussianMerge

2 つのガウス確率分布関数を結合する。

```
IppStatus ippsGaussianMerge_32f(const Ipp32f* pMean1, const Ipp32f*
    pVar1, const Ipp32f* pMean2, const Ipp32f* pVar2, Ipp32f* pDstMean,
    Ipp32f* pDstVar, int len, Ipp32f* pDstDet, Ipp32f wgt1, Ipp32f
    wgt2);
IppStatus ippsGaussianMerge_64f(const Ipp64f* pMean1, const Ipp64f*
    pVar1, const Ipp64f* pMean2, const Ipp64f* pVar2, Ipp64f* pDstMean,
    Ipp64f* pDstVar, int len, Ipp64f* pDstDet, Ipp64f wgt1, Ipp64f
    wgt2);
```

引数

pMean1	1 番目のガウス [len] の平均ベクトルへのポインタ。
pVar1	1 番目のガウス [len] の分散ベクトルへのポインタ。
pMean2	2 番目のガウス [len] の平均ベクトルへのポインタ。
pVar2	2 番目のガウス [len] の平均ベクトルへのポインタ。

len	平均ベクトルと分散ベクトルの長さ。
pDstMean	結合されたガウス [*len*] の平均ベクトルへのポインタ。
pDstVar	結合されたガウス [*len*] の分散ベクトルへのポインタ。
pDstDet	結合されたガウスの行列式へのポインタ。
wgt1	1 番目のガウスの重さ。
wgt2	2 番目のガウスの重さ。

説明

関数 ippsGaussianMerge は、ippsr.h ファイルで宣言される。この関数は、2 つのガウス確率分布関数を結合する。

$$pDstMean[i] = \frac{wgt1 \cdot pMean1[i] + wgt2 \cdot pMean2[i]}{wgt1 + wgt2} ,$$

$$pDstVar[i] = \frac{wgt1^2 \cdot (pVar1[i] + pMean1[i]^2) + wgt2^2 \cdot (pVar2[i] + pMean2[i]^2)}{wgt1 + wgt2} ,$$

$$pDstDet[0] = len \cdot \ln(2\pi) - \sum_{i=0}^{len-1} \ln pDstVar[i] ,$$

$0 \le i < len$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMean1*、*pMean2*、*pVar1*、*pVar2*、*pDstMean*、*pDstVar* または *pDstDet* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Entropy

入力ベクトルのエントロピを計算する。

```
IppStatus ippsEntropy_32f(const Ipp32f* pSrc, int len, Ipp32f*
    pResult);
```

```
IppStatus ippsEntropy_16s32s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
    int len, Ipp32s* pResult, int scaleFactor);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pResult	デスティネーション・エントロピ値へのポインタ。
len	入力ベクトルの長さ。
srcShiftVal	第 2 章の「整数のスケーリング」を参照。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsEntropy は、ippsr.h ファイルで宣言される。この関数は、次の式から得られる入力ベクトルのエントロピを計算する。

$$pResult[0] = \sum_{i=0}^{len-1} pSrc[i] \cdot \log_2 pSrc[i]$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pResult* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsLnNegArg	警告。いくつかの入力ベクトルの要素がゼロより小さい。演算の実行は中止されない。浮動小数点演算の場合、デスティネーション値は NaN に設定される。出力ベクトルのすべての要素がゼロに設定されている。

SinC

正弦を引数で割った値を計算する。

```
IppStatus ippsSinc_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinc_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinc_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinc_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSinc_64f_I(Ipp64f* pSrcDst, int len);
```


intel®

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pSrcDst	入力およびデスティネーション・ベクトル [*len*] へのポインタ。
pDst	デスティネーション・ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。

説明

関数 ippsSinc は、ippsr.h ファイルで宣言される。この関数は、次の式に従って、デスティネーション・ベクトルの要素を計算する。

$$pDst[k] = \frac{\sin(pSrc[k])}{pSrc[k]},\ k = 0,\ldots len - 1$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pSrcDst* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

ExpNegSqr

引数を 2 乗して符号を反転した値の指数を計算する。

```
IppStatus ippsExpNeqSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExpNeqSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExpNeqSqr_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int
    len);
IppStatus ippsExpNeqSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsExpNeqSqr_64f_I(Ipp64f* pSrcDst, int len);
```

引数

pSrc	入力ベクトル [*len*] へのポインタ。
pSrcDst	入力およびデスティネーション・ベクトル [*len*] へのポインタ。

pDst	デスティネーション・ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。

説明

関数 ippsExpNeqSqr は、ippsr.h ファイルで宣言される。この関数は、次の式に従って、デスティネーション・ベクトルの要素を計算する。

$$pDst[k] = \exp(-pSrc[k]^2), k = 0,\ldots len - 1$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pSrcDst* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

BhatDist

2 つのガウス間の Bhattacharia 距離を計算する。

```
IppStatus ippsBhatDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
    const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f*
    pResult);
IppStatus ippsBhatDist_32f64f(const Ipp32f* pMean1, const Ipp32f*
    pVar1, const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f*
    pResult);
IppStatus ippsBhatDistSLog_32f(const Ipp32f* pMean1, const Ipp32f*
    pVar1, const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f*
    pResult, Ipp32f sumLog1, Ipp32f sumLog2);
IppStatus ippsBhatDistSLog_32f64f(const Ipp32f* pMean1, const Ipp32f*
    pVar1, const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f*
    pResult, Ipp32f sumLog1, Ipp32f sumLog2);
```

引数

pMean1	1 番目の平均ベクトル [*len*] へのポインタ。
pVar1	1 番目の分散ベクトル [*len*] へのポインタ。
pMean2	2 番目の平均ベクトル [*len*] へのポインタ。

pVar2	2 番目の分散ベクトル [*len*] へのポインタ。
pResult	結果へのポインタ。
len	入力の平均および分散ベクトルへのポインタ。
sumLog1	1 番目のガウス分散の和（対数表現）。
sumLog2	2 番目のガウス分散の和（対数表現）。

説明

関数 `ippsBhatDist` および関数 `ippsBhatDistSLog` は、`ippsr.h` ファイルで宣言される。

関数 `ippsBhatDist` は、2 つのガウス間の Bhattacharia 距離を次のように計算する。

$$pResult[0] = \frac{1}{4}\sum_{i=0}^{len-1}\frac{(pMean1[i]-pMean2[i])^2}{pVar1[i]+pVar2[i]} + $$

$$+\frac{1}{2}\sum_{i=0}^{len-1}\left(\ln\left(\frac{pVar1[i]+pVar2[i]}{2}\right)-\frac{\ln(pVar1[i])+\ln(pVar2[i])}{2}\right)$$

関数 `ippsBhatDistSLog` は、2 つのガウス間の Bhattacharia 距離を次のように計算する。

$$pResult[0] = \frac{1}{4}\sum_{i=0}^{len-1}\frac{(pMean1[i]-pMean2[i])^2}{pVar1[i]+pVar2[i]} + $$

$$+\frac{1}{2}\sum_{i=0}^{len-1}\ln\left(\frac{pVar1[i]+pVar2[i]}{2}\right)-\frac{sumLog1+sumLog2}{4}$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pMean1*、*pVar1*、*pMean2*、*pVar2* または *pResult* が NULL。
`ippStsSizeErr`	エラー。*len* がゼロ以下。
`ippStsLnZeroArg`	警告。入力ベクトル内でゼロの値が検出された。実行は中止されない。ベクトル内に負の値が存在しない場合は、結果の値は `-Inf` に設定される。

ippStsLnNegArg	警告。入力ベクトル内で負の値が検出された。実行は中止されない。結果の値は NaN に設定される。

UpdateMean

EM トレーニング・アルゴリズムの
平均ベクトルを更新する。

```
IppStatus ippsUpdateMean_32f(const Ipp32f* pMeanAcc, Ipp32f* pMean, int
    len, Ipp32f meanOcc);
IppStatus ippsUpdateMean_64f(const Ipp64f* pMeanAcc, Ipp64f* pMean, int
    len, Ipp64f meanOcc);
```

引数

pMeanAcc	平均アキュムレータ [*len*] へのポインタ。
pMean	平均ベクトル [*len*] へのポインタ。
len	平均ベクトルの長さ。
meanOcc	ガウス混合のオキュペーション和。

説明

関数 ippsUpdateMean は、ippsr.h ファイルで宣言される。この関数は、EM (Expectation-Maximization) トレーニング・アルゴリズムの更新された平均ベクトルを次のように計算する。

$$pMean[i] = pMean[i] + \frac{pMeanAcc[i]}{meanOcc}, 0 \le i < len$$

ただし、$meanOcc \le 0$ の場合は、平均ベクトル *pMean* は更新されない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMean* または *pMeanAcc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsZeroOcc	警告。*meanOcc* がゼロ。
ippStsNegOccErr	エラー。*meanOcc* がゼロより小さい。

UpdateVar

EM トレーニング・アルゴリズムの
分散ベクトルを更新する。

```
IppStatus ippsUpdateVar_32f(const Ipp32f* pMeanAcc, const Ipp32f*
    pVarAcc, const Ipp32f* pVarFloor, Ipp32f* pVar, int len, Ipp32f
    meanOcc, Ipp32f varOcc);
IppStatus ippsUpdateVar_64f(const Ipp64f* pMeanAcc, const Ipp64f*
    pVarAcc, const Ipp64f* pVarFloor, Ipp64f* pVar, int len, Ipp64f
    meanOcc, Ipp64f varOcc);
```

引数

pMeanAcc	平均アキュムレータ [*len*] へのポインタ。
pVarAcc	分散アキュムレータ [*len*] へのポインタ。
pVarFloor	分散フロア・ベクトル [*len*] へのポインタ。
pVar	分散ベクトル [*len*] へのポインタ。
len	分散ベクトルの長さ。
meanOcc	ガウス混合のオキュペーション和。
varOcc	分散混合の平方オキュペーション和。

説明

関数 ippsUpdateVar は、ippsr.h ファイルで宣言される。この関数は、EM アルゴリズムの更新された分散ベクトルを計算する。共分散行列は対角行列と見なされる。アキュムレータはトレーニング・データから計算される。更新方程式は次のとおりである。

$$pVar[i] = \max\left[varFloor[i],\ \frac{varAcc[i]}{varOcc} - \left(\frac{meanAcc[i]}{meanOcc}\right)^2\right],\ 0 \le i < len$$

ただし、*meanOcc* ≦ 0 または *varOcc* ≦ 0 の場合は、分散ベクトル *pVar* は更新されない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMeanAcc*、*pVarAcc*、*pVarFloor* または *pVar* が NULL。

ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsZeroOcc	警告。*meanOcc* または *varOcc* がゼロ。
ippStsNegOccErr	エラー。*meanOcc* および *varOcc* がゼロより小さい。
ippStsResFloor	警告。すべての分散値がフロア。

UpdateWeight

EM トレーニング・アルゴリズムの
ガウス混合の重みの値を更新する。

```
IppStatus ippsUpdateWeight_32f(const Ipp32f* pWgtAcc, Ipp32f* pWgt, int
    len, Ipp32f* pWgtSum, Ipp32f wgtOcc, Ipp32f wgtThresh);
IppStatus ippsUpdateWeight_64f(const Ipp64f* pWgtAcc, Ipp64f* pWgt, int
    len, Ipp64f* pWgtSum, Ipp64f wgtOcc, Ipp64f wgtThresh);
```

引数

pWgtAcc	重みアキュムレータ [*len*] へのポインタ。
pWgt	重みベクトル [*len*] へのポインタ。
len	ガウス混合成分の数。
pWgtSum	重みの値の出力和へのポインタ。
wgtOcc	重みの更新方程式のノミネータ。
wgtThresh	重みの値のしきい値。

説明

関数 ippsUpdateWeight は、ippsr.h ファイルで宣言される。この関数は、ガウス混合の更新された重みの値を計算する。アキュムレータはトレーニング・データから計算される。更新方程式は次のとおりである。

$$pWgt[i] = \max\left(\frac{pWgtAcc[i]}{wgtOcc},\ wgtThresh\right), 0 \le i < len$$

$$pWgtSum[0] = \sum_{i=0}^{len-1} pWgt[i]$$

ただし、$wgtOcc \le 0$ の場合は、重みベクトル *pWgt* は更新されない。

この関数は、HMM 遷移行列の更新にも使用できる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pWgt*、*pWgtAcc* または *pWgtSum* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsZeroOcc	警告。*wgtOcc* がゼロ。
ippStsNegOccErr	エラー。*wgtOcc* がゼロより小さい。
ippStsResFloor	警告。すべての重みがフロア。

UpdateGConst

ガウスの出力確率密度関数の固定定数を更新する。

```
IppStatus ippsUpdateGConst_32f(const Ipp32f* pVar, int len, Ipp32f*
    pDet);
IppStatus ippsUpdateGConst_64f(const Ipp64f* pVar, int len, Ipp64f*
    pDet);
IppStatus ippsUpdateGConst_DirectVar_32f(const Ipp32f* pVar, int len,
    Ipp32f* pDet);
IppStatus ippsUpdateGConst_DirectVar_64f(const Ipp64f* pVar, int len,
    Ipp64f* pDet);
```

引数

pVar	分散ベクトル [*len*] へのポインタ。
len	分散ベクトルの次元。
pDet	結果の値へのポインタ。

説明

関数 ippsUpdateGConst は、ippsr.h ファイルで宣言される。この関数は、固定分散定数を計算する。
DirectVar サフィックスが付かない関数の場合、ガウス共分散行列は逆対角行列と見なされる。

$$pDet[0] = len \cdot \ln 2\pi - \sum_{i=o}^{len-1} \ln(pVar[i])$$

DirectVar サフィックスが付いた関数の場合、ガウス共分散行列は対角行列と見なされる。

$$pDet[0] = len \cdot \ln 2\pi + \sum_{i=o}^{len-1} \ln(pVar[i])$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pVar* または *pDet* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsLnZeroArg	警告。入力ベクトルの要素がゼロである。演算の実行は中止されない。ベクトル内に負の要素が存在しない場合は、結果の値は -Inf に設定される。
ippStsLnNegArg	警告。入力ベクトルの要素が負である。演算の実行は中止されない。結果の値は NaN に設定される。

OutProbPreCalc

ガウス混合の出力確率のうち、
観測ベクトルに無関係な部分を
事前に計算する。

```
IppStatus ippsOutProbPreCalc_32s(const Ipp32s* pWeight, const Ipp32s*
    pSrc, Ipp32s* pDst, int len);
IppStatus ippsOutProbPreCalc_32s_I(const Ipp32s* pWeight, Ipp32s*
    pSrcDst, int len);
IppStatus ippsOutProbPreCalc_32f(const Ipp32f* pWeight, const Ipp32f*
    pSrc, Ipp32f* pDst, int len);
IppStatus ippsOutProbPreCalc_64f(const Ipp64f* pWeight, const Ipp64f*
    pSrc, Ipp64f* pDst, int len);
IppStatus ippsOutProbPreCalc_32f_I(const Ipp32f* pWeight, Ipp32f*
    pSrcDst, int len);
IppStatus ippsOutProbPreCalc_64f_I(const Ipp64f* pWeight, Ipp64f*
    pSrcDst, int len);
```

引数

pWeight	ガウス混合重みベクトル [*len*] へのポインタ。
pSrc	ippsUpdateGConst 関数によって計算される入力ベクトルへのポインタ。
pSrcDst	ippsUpdateGConst 関数によって計算される入力 / 出力ベクトルへのポインタ。
pDst	結果ベクトル [*len*] へのポインタ。
len	HMM ステート内の混合成分の数。

説明

関数 ippsOutProbPreCalc は、ippsr.h ファイルで宣言される。この関数は、ガウス混合の出力確率のうち、観測ベクトルに無関係な部分を事前に計算する。

関数 ippsOutProbPreCalc の場合、

$$pDst[i] = pWeight[i] - 0.5 \cdot pSrc[i] \quad , 0 \le i < len$$

関数 ippsOutProbPreCalc_I の場合、

$$pSrcDst[i] = pWeight[i] - 0.5 \cdot pSrcDst[i] \quad , 0 \le i < len$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pWeight*、*pDet* または *pVal* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

DcsClustLAccumulate

決定木クラスタリング・アルゴリズムのステート・クラスタの尤度を計算するためのアキュムレータを更新する。

```
IppStatus ippsDcsClustLAccumulate_32f(const Ipp32f* pMean, const
    Ipp32f* pVar, Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f
    occ);
```

```
IppStatus ippsDcsClustLAccumulate_64f(const Ipp64f* pMean, const
    Ipp64f* pVar, Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f
    occ);
IppStatus ippsDcsClustLAccumulate_DirectVar_32f(const Ipp32f* pMean,
    const Ipp32f* pVar, Ipp32f* pDstSum, Ipp32f* pDstSqr, int len,
    Ipp32f occ);
IppStatus ippsDcsClustLAccumulate_DirectVar_64f(const Ipp64f* pMean,
    const Ipp64f* pVar, Ipp64f* pDstSum, Ipp64f* pDstSqr, int len,
    Ipp64f occ);
```

引数

pMean	クラスタ内の HMM ステートの平均ベクトル [*len*] へのポインタ。
pVar	クラスタ内の HMM ステートの分散ベクトル [*len*] へのポインタ。
pDstSum	アキュムレータ [*len*] の和の部分へのポインタ。
pDstSqr	アキュムレータ [*len*] の 2 乗和の部分へのポインタ。
len	平均ベクトルと分散ベクトルの長さ。
occ	HMM ステートのオキュペーション・カウント。

説明

関数 `ippsDcsClustLAccumulate` は、`ippsr.h` ファイルで宣言される。この関数は、決定木クラスタリング・アルゴリズムのアキュムレータを更新する。これらのアキュムレータを使用して、HMM ステート・クラスタの尤度を計算できる。

計算方程式は次のとおりである。

`DirectVar` サフィックスが付かない関数の場合、ガウス共分散行列は逆対角行列と見なされる。

$$pDstSum[i] = pMean[i] \cdot occ\ ,$$

$$pDstSqr[i] = \left[\frac{1}{pVar[i]} + (pMean[i])^2\right] \cdot occ\ ,\quad (0 \le i < len \text{ の場合})$$

`DirectVar` サフィックスが付いた関数の場合、ガウス共分散行列は対角行列と見なされる。

$$pDstSum[i] = pMean[i] \cdot occ\ ,$$

$$pDstSqr[i] = [pVar[i] + (pMean[i])^2] \cdot occ\ ,\quad (0 \le i < len \text{ の場合})$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMean*、*pVar*、*pDstSum* または *pDstSqr* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

DcsClustLCompute

決定木ステート・クラスタリング・アルゴリズムの HMM ステート・クラスタの尤度を計算する。

```
IppStatus ippsDcsClustLCompute_64f(const Ipp64f* pSrcSum, const Ipp64f*
    pSrcSqr, int len, Ipp64f* pDst, Ipp64f occ);
IppStatus ippsDcsClustLCompute_32f64f(const Ipp32f* pSrcSum, const
    Ipp32f* pSrcSqr, int len, Ipp64f* pDst, Ipp32f occ);
```

引数

pSrcSum	アキュムレータの和の部分 [*len*] へのポインタ。
pSrcSqr	アキュムレータの 2 乗和の部分 [*len*] へのポインタ。
len	*pSrcSum* ベクトルおよび *pSrcSqr* ベクトルの長さ。
pDst	結果の尤度値へのポインタ。
occ	HMM ステート・クラスタのオキュペーション和。

説明

関数 ippsDcsClustLCompute は、ippsr.h ファイルで宣言される。この関数は、ippsDcsClustLAccumulate 関数によって計算されたアキュムレータに従って、HMM ステート・クラスタの尤度を計算する。この尤度値を使用して、決定木ステート・クラスタリング・アルゴリズムの決定木ノードの分割方法が決定される。次のようにする。

$$pDst[0] = -\frac{1}{2}occ\cdot\left\{len\cdot[1+\ln 2\pi-2\ln(occ)] + \sum_{i=0}^{len-1}\ln[pSrcSqr[i]\cdot occ-(pSrcSum[i])^2]\right\}$$

ただし、$occ = 0$ の場合は、*pDst*[0] は IPPLOGZERO に設定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcSum*、*pSrcSqr* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下または *occ* がゼロ以下。
ippStsZeroOcc	警告。*occ* がゼロ。
ippStsNegOccErr	エラー。*occ* がゼロより小さい。
ippStsLnZeroArg	警告。入力ベクトルの要素がゼロである。演算の実行は中止されない。ベクトル内に負の要素が存在しない場合は、結果の値は -Inf に設定される。
ippStsLnNegArg	警告。入力ベクトルの要素が負である。演算の実行は中止されない。結果の値は NaN に設定される。

モデル適応

この項では、音響モデルおよび言語モデルの適応に使用される関数について説明する。適応アルゴリズムは、少数の学習サンプルに基づいて、ユーザが設定した特性に合わせて既存のモデルのパラメータを調整する。

AddMulColumn

重み付きの行列の列を別の列に加算する。

```
IppStatus ippsAddMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int
    height, int col1, int col2, int row1, const Ipp64f val);
```

引数

mSrcDst	ソースおよびデスティネーション行列 [*height*][*width*] へのポインタ。
width	行列 *mSrcDst* の列数。
height	行列 *mSrcDst* の行数。
col1	1 番目のオペランドの列番号。
col2	2 番目のオペランドの列番号。
row1	開始行番号。
val	重み係数。

説明

関数 AddMulColumn は、ippsr.h ファイルで宣言される。この関数は、*val* で重み付けされた行列の列を別の行列の列に加算する。これは、SVD アルゴリズムを高速に実行するために使用される。
次のようにする。

$row1 \le i < height$ の場合、
$mSrcDst[i][col2] = mSrcDst[i][col2] + mSrcDst[i][col1] \cdot val$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *mSrcDst* が NULL。
ippStsSizeErr	エラー。*height*、*width*、*col1*、*col2* または *row1* がゼロ以下、*col1* または *col2* が *width* 以上、あるいは *row1* が *height* 以上。

AddMulRow

重み付きのベクトルを別のベクトルに
加算する。

```
ippsAddMulRow_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len,
    const Ipp64f val);
```

引数

pSrc	ソース・ベクトル [*len*] へのポインタ。
pSrcDst	ソースおよびデスティネーション・ベクトル [*len*] へのポインタ。
len	ソースおよびデスティネーション・ベクトルの長さ。
val	重み係数。

説明

関数 AddMulRow は、ippsr.h ファイルで宣言される。この関数は、*val* で重み付けされた 1 つのベクトルを別のベクトルに加算する。これは、SVD アルゴリズムを高速に実行するために使用される。次のようにする。

$$pSrcDst[i] = pSrcDst[i] + pSrc[i] \cdot val \text{ , } 0 \le i < len$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

QRTransColumn

QR 変換を実行する。

```
IppStatus ippsQRTransColumn_64f_D2L(Ipp64f** mSrcDst, int width, int
    height, int col1, int col2, const Ipp64f val1, const Ipp64f val2);
```

引数

mSrcDst	ソース行列およびデスティネーション行列 [*height*][*width*] へのポインタ。
width	行列 *mSrcDst* の列数。
height	行列 *mSrcDst* の行数。
col1	1 番目の列番号。
col2	2 番目の列番号。
val1	1 番目の重み係数。
val2	2 番目の重み係数。

説明

関数 ippsQRTransColumn は、ippsr.h ファイルで宣言される。この関数は QR 変換を実行する。これは、SVD アルゴリズムを高速に実行するために使用される。次のようにする。

$0 \le i < height$ の場合、

$$mSrcDst[i][col2] = mSrcDst[i][col2] \cdot val1 + mSrcDst[i][col1] \cdot val2 \text{ ,}$$

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val1 + mSrcDst[i][col2] \cdot val2 \text{ ,}$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*mSrcDst* ポインタが NULL。
ippStsSizeErr	エラー。*height*、*width*、*col1* または *col2* がゼロ以下、*col1* または *col2* が *width* 以上、あるいは *col2* が *height* 以上。

DotProdColumn

2 つの行列の列に対して内積を計算する。

```
IppStatus ippsDotProdColumn_64f_D2L(const Ipp64f** mSrc, int width,
    int height, Ipp64f* pSum, int col1, int col2, int row1);
```

引数

mSrc	ソース行列 [*height*][*width*] へのポインタ。
width	行列 *mSrc* の列数。
height	行列 *mSrc* の行数。
pSum	計算された合計へのポインタ。
col1	1 番目の列番号。
col2	2 番目の列番号。
row1	1 番目の行番号。

説明

関数 ippsDotProdColumn は、ippsr.h ファイルで宣言される。この関数は、2 つの行列の列に対する内積を計算する。これは、SVD アルゴリズムを高速に実行するために使用される。次のようにする。

$$pSum[0] = \sum_{i = row1}^{height-1} mSrc[i][col1] \cdot mSrc[i][col2]$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *mSrc* または *pSum* が NULL。

ippStsSizeErr	エラー。*height*、*width*、*col1* または *row1* がゼロ以下、*row1* が *height* 以上、あるいは *col1* が *width* 以上。

MulColumn

行列の列に値を掛ける。

```
IppStatus ippsMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int
    height, int col1, int row1, const Ipp64f val);
```

引数

mSrcDst	ソースおよびデスティネーション行列 [*height*][*width*] へのポインタ。
width	行列 *mSrcDst* の列数。
height	行列 *mSrcDst* の行数。
col1	1 番目の列番号。
row1	1 番目の行番号。
val	重み係数。

説明

関数 ippsMulColumn は、ippsr.h ファイルで宣言される。この関数は、*mSrcDst* 行列の列に *val* を掛ける。これは、SVD アルゴリズムを高速に実行するために使用される。次のようにする。

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val\ ,\quad row1 \le i < height$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*mSrcDst* ポインタが NULL。
ippStsSizeErr	エラー。*height*、*width*、*col1* または *row1* がゼロ以下、*row1* が *height* 以上、あるいは *col1* が *width* 以上。

intel®

SumColumnAbs

行列の列要素の和の絶対値を計算する。

```
IppStatus ippsSumColumnAbs_64f_D2L(const Ipp64f** mSrc, int width,
    int height, Ipp64f* pSum, int col1, int row1);
```

引数

mSrc	ソース行列 [*height*][*width*] へのポインタ。
width	行列 *mSrc* の列数。
height	行列 *mSrc* の行数。
pSum	計算された合計へのポインタ。
col1	1 番目の列番号。
row1	1 番目の行番号。

説明

関数 ippsSumColumnAbs は、ippsr.h ファイルで宣言される。この関数は、列の要素の絶対値を合計する。これは、SVD アルゴリズムを高速に実行するために使用される。次のようにする。

$$pSum[0] = \sum_{i = row1}^{height-1} |mSrc[i][col1]|$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *mSrc* または *pSum* が NULL。
ippStsSizeErr	エラー。*width*、*height*、*col1* または *row1* がゼロ以下、*row1* が *height* 以上、あるいは *col1* が *width* 以上。

SumColumnSqr

重み付けされた行列の列の要素の 2 乗和を計算する。

```
IppStatus ippsSumColumnSqr_64f_D2L(Ipp64f** mSrcDst, int width, int
    height, Ipp64f* pSum, int col1, int row1, const Ipp64f val);
```

引数

mSrcDst	ソースおよびデスティネーション行列 [*height*][*width*] へのポインタ。
width	行列 *mSrcDst* の列数。
height	行列 *mSrcDst* の行数。
pSum	計算された合計値へのポインタ。
col1	1 番目の列番号。
row1	2 番目の行番号。
val	重み係数。

説明

関数 ippsSumColumnSqr は、ippsr.h ファイルで宣言される。この関数は、行列 *mSrcDst* の列に *val* を掛け、その列の要素の 2 乗を計算する。これは、SVD アルゴリズムを高速に実行するために使用される。次のようにする。

$$mSrcDst[i][col1] = mSrcDst[i][col1] \cdot val \;,\quad 0 \le i < height$$

$$pSum[0] = \sum_{i = row1}^{height-1} (mSrcDst[i][col1])^2$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *mSrcDst* または *pSum* が NULL。
ippStsSizeErr	エラー。*width*、*height*、*col1* または *row1* がゼロ以下、*row1* が *height* 以上、あるいは *col1* が *width* 以上。

SumRowAbs

ベクトル要素の和の絶対値を計算する。

```
IppStatus ippsSumRowAbs_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
```

引数

pSrc	ソース・ベクトル [*len*] へのポインタ。
pSum	計算された合計値へのポインタ。
len	ソース・ベクトル *pSrc* の長さ。

説明

関数 ippsSumRowAbs は、ippsr.h ファイルで宣言される。この関数は、ベクトル要素の和の絶対値を次のように計算する。

$$pSum[0] = \sum_{i=0}^{len-1} |pSrc[i]|$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pSum* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

SumRowSqr

重み付けされたベクトルの要素の 2 乗和を
計算する。

```
IppStatus ippsSumRowSqr_64f(Ipp64f* pSrcDst, int len, Ipp64f* pSum,
    const Ipp64f val);
```

引数

pSrcDst	ソースおよびデスティネーション・ベクトル [*len*] へのポインタ。
len	ソース・ベクトル *pSrcDst* の長さ。
pSum	計算された合計値へのポインタ。
val	重み係数。

説明

関数 ippsSumRowSqr は、ippsr.h ファイルで宣言される。この関数は、ベクトル *pSrcDst* に *val* を掛け、ベクトルの要素の 2 乗和を計算する。

$$pSrcDst[i] = pSrcDst[i] \cdot val\ ,\ 0 \le i < len$$

$$pSum[0] = \sum_{i=0}^{len-1} (pSrcDst[i])^2$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst* または *pSum* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

SVD

行列の特異値分解を実行する。

```
IppStatus ippsSVD_64f_D2(const Ipp64f* pSrcA, Ipp64f* pDstU, int
    height, Ipp64f* pDstW, Ipp64f* pDstV, int width, int step, int
    nIter);
IppStatus ippsSVD_64f_D2L(const Ipp64f** mSrcA, Ipp64f** mDstU, int
    height, Ipp64f* pDstW, Ipp64f** mDstV, int width, int nIter);
IppStatus ippsSVD_64f_D2_I(Ipp64f* pSrcDstA, int height, Ipp64f* pDstW,
    Ipp64f* pDstV, int width, int step, int nIter);
IppStatus ippsSVD_64f_D2L_I(Ipp64f** mSrcDstA, int height, Ipp64f*
    pDstW, Ipp64f** mDstV, int width, int nIter);
```

引数

pSrcA	入力ベクトル *A* [*height***step*] へのポインタ。
pDstU	出力ベクトル *U* [*height***step*] へのポインタ。
pSrcDstA	入力行列 *A* および出力行列 *U* [*height***step*] へのポインタ。
pDstV	出力ベクトル *V* [*width***step*] へのポインタ。
mSrcA	入力行列 *A* [*height*][*width*] へのポインタ。
mDstU	出力行列 *U* [*height*][*width*] へのポインタ。
mSrcDstA	入力行列 *A* および出力行列 *U* [*height*][*width*] へのポインタ。
pDstW	出力ベクトル *W* [*width*] へのポインタ。
mDstV	出力行列 *V* [*width*][*width*] へのポインタ。
height	入力行列の行数。
width	入力行列の列数。
step	ベクトル *pSrcA*、*pSrcDstA* または *pDstV* の行のステップ。
nIter	対角化の反復回数。

説明

関数 ippsSVD は、ippsr.h ファイルで宣言される。この関数は、入力行列 *A* の特異値分解（SVD）を実行する。
出力行列 *U*、*W*、*V* は、次の条件を満たす。

$$A = U \circ W \circ V^T ,$$

ここで、行列 *U* は列直交行列、行列 *W* は対角行列（ベクトルとして格納される）、行列 *V* は直交行列である。

V^T は行列 *V* の転置行列である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。

ippStsSizeErr	エラー。*height*、*width*、*step* または *nIter* がゼロ以下、あるいは *width* が *step* より大きい。
ippStsSVDCnvgErr	エラー。*nIter* 回の反復後に SVD アルゴリズムが収束しなかった。

WeightedSum

2 つの入力ベクトルの要素の重み付けされた和を計算する。

```
IppStatus ippsWeightedSum_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSum_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSum_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);
IppStatus ippsWeightedSumHalf_16s(const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_32f(const Ipp32f* pSrc1, const Ipp32f*
    pSrc2, Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);
IppStatus ippsWeightedSumHalf_64f(const Ipp64f* pSrc1, const Ipp64f*
    pSrc2, Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);
```

引数

pSrc1	1 番目の入力ベクトル [*len*] へのポインタ。
pSrc2	2 番目の入力ベクトル [*len*] へのポインタ。
pDst	出力ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。
weight1	1 番目の重みの値。
weight2	2 番目の重みの値。

説明

関数 ippsWeightedSum および関数 ippsWeightedSumHalf は、ippsr.h ファイルで宣言される。関数 ippsWeightedSum は、重み付けされた和を次のように計算する。

$$pDst[i] = \frac{weight1 \cdot pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2}, \quad i = 0, \ldots len-1$$

関数 ippsWeightedSumHalf は、次の式から得られる重み付けされた和を計算する。

$$pDst[i] = \frac{pSrc1[i] + weight2 \cdot pSrc2[i]}{weight1 + weight2}, \quad i = 0, \ldots len-1$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsDivByZero	警告。除数ベクトルの要素の値がゼロ。実行は中止されない。浮動小数点演算の場合、デスティネーション・ベクトル要素の値は、次のように設定される

NaN	被除数ベクトル要素の値がゼロの場合
+Inf	被除数ベクトル要素の値が正の場合
-Inf	被除数ベクトル要素の値が負の場合

ベクトル量子化

この項では、ベクトル量子化およびコードブック演算用の関数について説明する。これらの関数は、一般に音響モデルおよび言語モデルの圧縮に使用される。

FormVector

コードブック・エントリから複数のストリームの出力ベクトルを作成する。

```
IppStatus ippsFormVector_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc,
    const Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s*
    pSteps, int nStream, Ipp16s* pDst);
```

```
IppStatus ippsFormVector_16s16s(const Ipp16s* pInd, const Ipp16s**
    mSrc, const Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s*
    pSteps, int nStream, Ipp16s* pDst);
IppStatus ippsFormVector_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc,
    const Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s*
    pSteps, int nStream, Ipp32f * pDst);
IppStatus ippsFormVector_16s32f(const Ipp16s* pInd, const Ipp32f**
    mSrc, const Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s*
    pSteps, int nStream, Ipp32f * pDst);

IppStatus ippsFormVector_2i_8u16s(const Ipp8u* pInd, const Ipp16s**
    mSrc, const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_2i_16s16s(const Ipp16s* pInd, const Ipp16s**
    mSrc, const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_2i_8u32f(const Ipp8u* pInd, const Ipp32f**
    mSrc, const Ipp32s* pHeights, Ipp32f* pDst, int len);
IppStatus ippsFormVector_2i_16s32f(const Ipp16s* pInd, const Ipp32f**
    mSrc, const Ipp32s* pHeights, Ipp32f* pDst, int len);

IppStatus ippsFormVector_4i_8u16s(const Ipp8u* pInd, const Ipp16s**
    mSrc, const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_4i_16s16s(const Ipp16s* pInd, const Ipp16s**
    mSrc, const Ipp32s* pHeights, Ipp16s* pDst, int len);
IppStatus ippsFormVector_4i_8u32f(const Ipp8u* pInd, const Ipp32f**
    mSrc, const Ipp32s* pHeights, Ipp32f* pDst, int len);
IppStatus ippsFormVector_4i_16s32f(const Ipp16s* pInd, const Ipp32f**
    mSrc, const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

引数

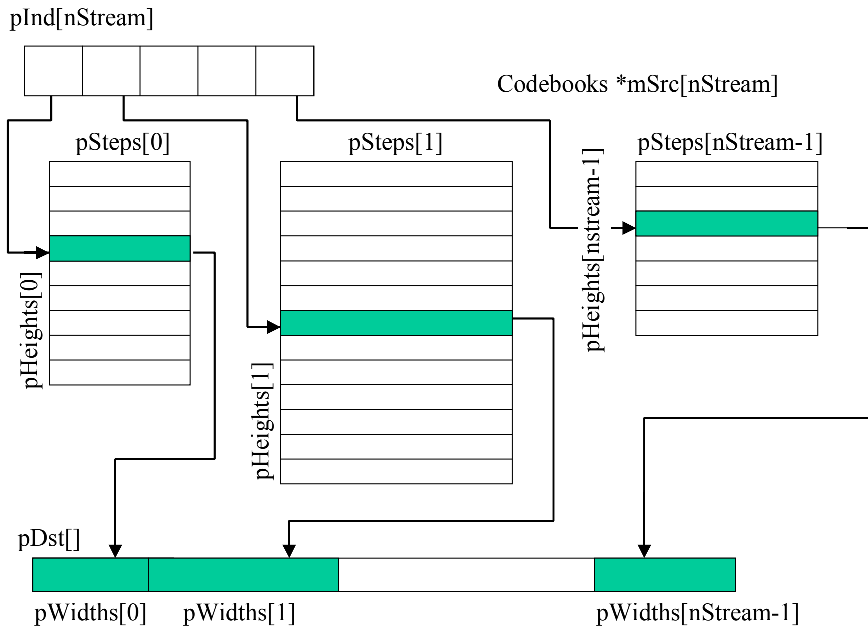
pInd	インデックス・ベクトル [*nStream*] へのポインタ。
mSrc	コードブックへのポインタの配列 [*nStream*] へのポインタ。
pHeights	コードブックのサイズ [*nStream*] へのポインタ。
pWidths	ストリームの長さ [*nStream*] へのポインタ。
pSteps	コードベクトルの長さ [*nStream*] へのポインタ。
nStream	コードブックの数。
pDst	出力ベクトルへのポインタ。
len	出力ベクトルの長さ。

説明

関数 ippsFormVector は、ippsr.h ファイルで宣言される。この関数は、複数のストリームの出力ベクトルを作成する。各ストリーム（サイズは *pWeights*[]）は、*pInd*[] によってインデックスが付けられたコードブック・エントリである。コードブックは、*mSrc*[] によって参照され、*pHeights*[] 個のコードベクトルを持つ。各コードベクトルのサイズは、*pSteps*[] である。

次の図にレイアウトを示す。

図 8-5　ippsFormVector 関数のストリームのレイアウト



出力ベクトルは、次の式から得られる。

$$pDst\left[i + \sum_{j=0}^{k-1} pWidths[j]\right] = mSrc[k][pInd[k] \cdot pSteps[k] + i] \quad ,$$

$k = 0,...nStream - 1;\ i = 0,...pWidths[k] - 1$

関数 ippsFormVector_2i は、*pWidths*[] = *pSteps*[] = 2 および *nStream* = *len*/2 の制約条件を前提として、抽出プロセスを簡単にする。

同様に、関数 ippsFormVector_4i は、*pWidths*[] = *pSteps*[] = 4 および *nStream* = *len*/4 の制約条件を前提とする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意の指定されたポインタが NULL。
ippStsSizeErr	エラー。*pInd*[*k*] がゼロより小さい、*len*、*nStream*、*pWidths*[*k*]、または *pSteps*[*k*] がゼロ以下、あるいは *pHeights*[*k*] が *pInd*[*k*] 以下。

CdbkGetSize

コードブックのサイズをバイト単位で計算する。

```
IppStatus ippsCdbkGetSize_16s(int width, int step, int height, int
    cdbkSize, IppCdbk_Hint hint, int *pSize);
```

引数

width	入力ベクトルの長さ（0 < *width* < 512）。
step	ソース・ベクトル pSrc の行のステップ（0 < *step* < 512）。
height	ソース・ベクトル pSrc の行数（現在のところ *height* = *cdbkSize* だけが使用できる）。
cdbkSize	コードブックのサイズ（0 < *cdbkSize* ≤ 8192）。
hint	コードブックの形式を示すフラグ。詳細は、「ippsCdbkInit」を参照のこと。
pSize	コードブック構造体および関連ストレージのサイズを格納する変数へのポインタ（バイト単位）。

説明

関数 ippsCdbkGetSize は、ippsr.h ファイルで宣言される。この関数は、コードブックと高速検索で使用する追加情報のサイズをバイト単位で計算する。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSize* が NULL。
ippStsSizeErr	エラー。*cdbkSize* がゼロ以下、*cdbkSize* が 8192 より大きい、*cdbkSize* が *height* と等しくない、*width*、*step*、または *height* がゼロ以下、あるいは *width* が *step* より大きい。
ippStsCdbkFlagErr	エラー。*hint* の値が正しくない、またはサポートされていない。

注： この関数で使用できる *hint* の値は IPP_CDBK_FULL のみである。このモードでは、*height* パラメータは必要ない。この値は無視される。

CdbkInit

コードブックを格納する構造体を
初期化する。

```
IppStatus ippsCdbkInit_L2_16s(IppsCdbkState_16s* pCdbk, const Ipp16s*
    pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint
    hint);
```

引数

pSrc	*height*step* エントリを含むソース・ベクトルへのポインタ。
width	入力ベクトルの長さ（$0 < width < 512$）。
step	ソース・ベクトル *pSrc* の行のステップ（$0 < step < 512$）。
height	ソース・ベクトル *pSrc* の行数（$0 < height \leq cdbkSize$）。
cdbkSize	コードブックのサイズ（$0 < cdbkSize \leq 8192$）。
hint	次の値のうち 1 つ。

IPP_CDBK_FULL	ソース・データはコードブックのエントリである。*height* は nCluster 以上でなければならない。フル検索によって、最も近いコードブック・エントリを見つける。

IPP_CDBK_KMEANS_LONG	コードブックの作成に、LBG アルゴリズムとサイズが最も大きいクラスタの分割を使用する。対数検索によって、最も近いコードブック・エントリを見つける。
IPP_CDBK_KMEANS_NUM	コードブックの作成に、LBG アルゴリズムと数が最も多いクラスタの分割を使用する。対数検索によって、最も近いコードブック・エントリを見つける。

pCdbk 初期化されるコードブック構造体へのポインタ。

説明

関数 ippsCdbkInit は、ippsr.h ファイルで宣言される。この関数は、コードブックと高速検索用の追加情報を格納する構造体を初期化する。この構造体は、ippsSplitVQ 関数によるベクトル量子化に使用される。2 つのベクトルの相似性の測定には、ユークリッド距離が使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCdbk* または *pSrc* が NULL。
ippStsSizeErr	エラー。*width*、*step*、*height*、または *cdbkSize* がゼロ以下、*width* が *step* より大きい、あるいは *cdbkSize* が 8192 より大きい。
ippStsCdbkFlagErr	エラー。*hint* の値が正しくない、またはサポートされていない。

注: この関数で使用できる *hint* の値は IPP_CDBK_FULL のみである。*pCdbk* に格納されたステート・メモリ・アドレスは、32 ビットのワード境界にアライメントする必要がある。

CdbkInitAlloc

コードブック構造体を初期化する。

```
IppStatus ippsCdbkInitAlloc_L2_16s(IppsCdbkState_16s** pCdbk, const
    Ipp16s* pSrc, int width, int step, int height, int cdbkSize,
    Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_L2_32f(IppsCdbkState_32f** pCdbk, const
    Ipp32f* pSrc, int width, int step, int height, int cdbkSize,
    Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_WgtL2_16s(IppsCdbkState_16s** pCdbk, const
    Ipp16s* pSrc, const Ipp16s* pWgt, int width, int step, int height,
    int cdbkSize, Ipp_Cdbk_Hint hint);
IppStatus ippsCdbkInitAlloc_WgtL2_32f(IppsCdbkState_32f** pCdbk, const
    Ipp32f* pSrc, const Ipp32f* pWgt, int width, int step, int height,
    int cdbkSize, Ipp_Cdbk_Hint hint);
```

引数

pCdbk	作成されるコードブック構造体へのポインタ。
pSrc	ソース・ベクトル [*height*step*] へのポインタ。
pWgt	重みベクトル [*width*] へのポインタ。
width	入力ベクトルの長さ。
step	ソース・ベクトル *pSrc* の行のステップ。
height	ソース・ベクトル *pSrc* の行数。
cdbkSize	コードブックのサイズ。
hint	次の値のうち 1 つ。

IPP_CDBK_FULL	ソース・データはコードブックのエントリである。*height* は *cdbkSize* 以上でなければならない。フル検索によって、最も近いコードブック・エントリを見つける。
IPP_CDBK_KMEANS_LONG	コードブックの作成に、LBG アルゴリズムとサイズが最も大きいクラスタの分割を使用する。対数検索によって、最も近いコードブック・エントリを見つける。

`IPP_CDBK_KMEANS_NUM`	コードブックの作成に、LBG アルゴリズムと数が最も多いクラスタの分割を使用する。対数検索によって、最も近いコードブック・エントリを見つける。

説明

関数 `ippsCdbkInitAlloc` は、`ippsr.h` ファイルで宣言される。この関数は、コードブックと高速検索用の追加情報を格納する構造体を初期化する。この構造体は、`ippsVQ` または `ippsSplitVQ` 関数によるベクトル量子化に使用される。`ippsCdbkInitAlloc_L2` 関数は、ユークリッド距離を使用して 2 つのベクトルの相似性を測定する。`ippsCdbkInitAlloc_WgtL2` 関数は、重み付けされたユークリッド距離を使用して 2 つのベクトルの相似性を測定する。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pCdbk* または *pSrc* が `NULL`。
`ippStsSizeErr`	エラー。*width*、*step* または *cdbkSize* がゼロ以下、*cdbkSize* が *height* より大きい、*width* が *step* より大きい、*cdbkSize* が 16383 より大きい、あるいは、*hint* が `IPP_CDBK_FULL` と等しく、*cdbkSize* が *height* と等しくない。
`ippStsCdbkFlagErr`	エラー。*hint* の値が正しくない。
`ippStsMemAllocErr`	エラー。メモリが割り当てられていない。
`ippStsBadArgErr`	エラー。1 つの *pWgt*[*i*] がゼロ以下。

CdbkFree

コードブック構造体を破壊する。

```
IppStatus ippsCdbkFree_16s(IppsCdbkState_16s* pCdbk);
IppStatus ippsCdbkFree_32f(IppsCdbkState_32f* pCdbk);
```

引数

pCdbk	コードブック構造体へのポインタ。

説明

関数 ippsCdbkFree は、ippsr.h ファイルで宣言される。この関数は、コードブック構造体を破壊し、その構造体に割り当てられたすべてのメモリを解放する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pCdbk* ポインタが NULL。

GetCdbkSize

コードブック内のコードベクトルの数を取得する。

```
IppStatus ippsGetCdbkSize_16s(const IppsCdbkState_16s* pCdbk, int*
    pNum);
IppStatus ippsGetCdbkSize_32f(const IppsCdbkState_32f* pCdbk, int*
    pNum);
```

引数

pCdbk	コードブック構造体へのポインタ。
pNum	結果のコードベクトル数へのポインタ。

説明

関数 ippsGetCdbkSize は、ippsr.h ファイルで宣言される。この関数は、コードブック *pCdbk* 内のコードベクトルの数を取得する。*pSrc* 内の異なるベクトルの数が *cdbkSize* より小さい場合、結果の値が *cdbkSize* より小さくなることがある。コードブック構造体 *pCdbk* は、ippsCdbkAlloc 関数によって初期化される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCdbk* または *pNum* が NULL。

GetCodebook

コードブックからコードベクトルを
取得する。

```
IppStatus ippsGetCodebook_16s(const IppsCdbkState_16s* pCdbk, Ipp16s*
    pDst, int step);
IppStatus ippsGetCodebook_32f(const IppsCdbkState_32f* pCdbk, Ipp32f*
    pDst, int step);
```

引数

pCdbk	コードブック構造体へのポインタ。
pDst	コードベクトル用のデスティネーション・ベクトル [*pNum*[0]**step*] へのポインタ。
step	デスティネーション・ベクトル *pDst* の行のステップ。

説明

関数 ippsGetCodebook は、ippsr.h ファイルで宣言される。この関数は、コードブック構造体 *pCdbk* からコードベクトルを取得し、行のステップが *step* の *pDst* ベクトルに格納する。

コードブック構造体 *pCdbk* は、関数 ippsCdbkAlloc によって初期化される。クラスタの数は、関数 ippsGetCdbkSize で取得できる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCdbk* または *pDst* が NULL。
ippStsSizeErr	エラー。*step* がゼロ、または *step* が *width* より小さい。

VQ

コードブックに基づいて入力ベクトルを
量子化する。

```
IppStatus ippsVQ_16s(const Ipp16s* pSrc, int step, Ipp32s* pIndx, int
    height, const IppsCdbkState_16s* pCdbk);
IppStatus ippsVQ_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx, int
    height, const IppsCdbkState_32f* pCdbk);
IppStatus ippsVQDist_16s32s_Sfs(const Ipp16s* pSrc, int step, Ipp32s*
    pIndx, Ipp32s* pDist, int height, const IppsCdbkState_16s* pCdbk,
    int scaleFactor);
IppStatus ippsVQDist_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx,
    Ipp32f* pDist, int height, const IppsCdbkState_32f* pCdbk);
```

引数

pCdbk	コードブック構造体へのポインタ。
pSrc	ソース・ベクトル [*height***step*] へのポインタ。
step	ソース・ベクトル *pSrc* の行のステップ。
height	ソース・ベクトル *pSrc* の行数。
pIndx	最も近いコードベクトルから得られたインデックス・ベクトル [*height*] へのポインタ。
pDist	ソース・ベクトル [*height*] から得られる量子化距離へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsVQ および関数 ippsVQDist は、ippsr.h ファイルで宣言される。関数 ippsVQ は、入力ベクトルのベクトル量子化（VQ）を実行する。最も近いコードベクトルから得られたインデックスは、ベクトル *pIndx* に格納される。関数 ippsVQDist は、出力距離ベクトル *pDist* に量子化距離も格納する。整数バージョンでは この距離は *scaleFactor* を使用してスケーリングされる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCdbk*、*pSrc*、*pIndx*、または *pDist* が NULL。

ippStsSizeErr	エラー。*step* または *height* がゼロ以下。

VQSingle_Sort, VQSingle_Thresh

コードブックに基づいて入力ベクトルを量子化し、最も近い複数のクラスタを取得する。

```
IppStatus ippsVQSingle_Sort_32f(const Ipp32f *pSrc, Ipp32s *pIndx,
    const IppsCdbkState_32f* pCdbk, int num);
IppStatus ippsVQSingle_Sort_16s(const Ipp16s *pSrc, Ipp32s *pIndx,
    const IppsCdbkState_16s* pCdbk, int num);
IppStatus ippsVQDistSingle_Sort_32f(const Ipp32f *pSrc, Ipp32s *pIndx,
    Ipp32f *pDist, const IppsCdbkState_32f* pCdbk, int num);
IppStatus ippsVQDistSingle_Sort_16s32s_Sfs(const Ipp16s *pSrc, Ipp32s
    *pIndx, Ipp32s *pDist, const IppsCdbkState_16s* pCdbk, int num, int
    scaleFactor);
IppStatus ippsVQSingle_Thresh_32f(const Ipp32f *pSrc, Ipp32s *pIndx,
    const IppsCdbkState_32f* pCdbk, Ipp32f val, int *pnum);
IppStatus ippsVQSingle_Thresh_16s(const Ipp16s *pSrc, Ipp32s *pIndx,
    const IppsCdbkState_16s* pCdbk, Ipp32f val, int *pnum);
IppStatus ippsVQDistSingle_Thresh_32f(const Ipp32f *pSrc, Ipp32s
    *pIndx, Ipp32f *pDist, const IppsCdbkState_32f* pCdbk, Ipp32f val,
    int *pnum);
IppStatus ippsVQDistSingle_Thresh_16s32s_Sfs(const Ipp16s *pSrc, Ipp32s
    *pIndx, Ipp32s *pDist, const IppsCdbkState_16s* pCdbk, Ipp32f val,
    int *pnum, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pIndx	デスティネーション・インデックス・ベクトルへのポインタ。
pDist	デスティネーション距離ベクトルへのポインタ。
pCdbk	コードブック構造体へのポインタ。
val	相対しきい値。
num	各入力ベクトルから検索する最も近いクラスタの数。
pnum	しきい値 [1] 内にあるクラスタの数へのポインタ。
scaleFactor	中間和のスケール係数。

説明

関数 ippsVQSingle_Sort と ippsVQSingle_Thresh は、ippsr.h ファイルで宣言される。これらの関数は、入力ベクトルに対して複数のベクトル量子化（VQ）を実行する。Sort サフィックスが付いた関数は、最も近い *num* 個のコードブック・セントロイドのインデックスを、距離に基づいた昇順に並べ替えて提供する。Thresh サフィックスが付いた関数は、最小値に *val* を掛けた値より小さい距離と、それらのクラスタの数を含むコードブック・セントロイドのインデックスを提供する。Dist サフィックスが付いた関数は、最も近いクラスタの距離の値を提供する。ベクトル *pSrc* の長さはコードベクトルの長さに等しく、出力ベクトル *pIndx* と *pDist* の長さはコードブックのサイズに等しい。フル検索は、すべてのコードブック形式に対して行われる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pIndx*、*pDist*、*pCdbk* または *pnum* が NULL。
ippStsSizeErr	エラー。*num* がゼロ以下またはコードブックのサイズより大きい。
ippStsBadArgErr	エラー。*val* が 1 より小さい。

SplitVQ

コードブックに基づいてマルチストリーム・ベクトルを量子化する。

```
IppStatus ippsSplitVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s*
    pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks,
    int nStream);
IppStatus ippsSplitVQ_16s8u(const Ipp16s* pSrc, int srcStep, Ipp8u*
    pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks,
    int nStream);
IppStatus ippsSplitVQ_16s1u(const Ipp16s* pSrc, int srcStep, Ipp8u*
    pDst, int dstBitStep, int height, const IppsCdbkState_16s** pCdbks,
    int nStream);
IppStatus ippsSplitVQ_32f16s(const Ipp32f* pSrc, int srcStep, Ipp16s*
    pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks,
    int nStream);
```

```
IppStatus ippsSplitVQ_32f8u(const Ipp32f* pSrc, int srcStep, Ipp8u*
    pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks,
    int nStream);
IppStatus ippsSplitVQ _32f1u(const Ipp32f* pSrc, int srcStep, Ipp8u*
    pDst, int dstBitStep, int height, const IppsCdbkState_32f** pCdbks,
    int nStream);
```

引数

pCdbks	コードブック構造体 [*nStream*] へのポインタ。
pSrc	ソース・ベクトル [*height***srcStep*] へのポインタ。
srcstep	ソース・ベクトル *pSrc* の行のステップ。
pDst	デスティネーション・インデックス・ベクトル [*height***dststep*] へのポインタ。
dstStep	デスティネーション・ベクトル *pDst* の行のステップ。
height	ソースおよびデスティネーション・ベクトルの行数。
dstBitstep	デスティネーション・ベクトルの行のステップ (ビット単位)。
nStream	ソース・ベクトルのストリームの数。

説明

関数 ippsSplitVQ は、ippsr.h ファイルで宣言される。この関数は、コードブック *pDdbks* に基づいてマルチストリーム・ベクトル *pSrc* を量子化する。各ストリームの長さは、対応するコードブック・ベクトルの長さに等しいと見なされる。出力 *pDst* は、コードブック・エントリのインデックスである。

1u サフィックスが付いた関数の場合、出力インデックスはビット単位でパックされる。各ストリームは、そのストリームのコードブック・インデックスを表現するのに必要な最小限のビット数を使用する。

ippsFormVector、ippsFormVectorVQ 関数も参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCdbk*、*pCdbk*[*k*]、*pSrc*、または *pDst* が NULL。
ippStsSizeErr	エラー。*srcStep*、*dstStep*、*height*、または *nStream* がゼロ以下、ストリームの長さの合計が *srcStep* より大きい、*nStream* が *dstStep* より大きい (16s または 8u サフィックスが付いた関数の場合)、インデックスを

表現するのに必要なビット数が *dstStep* より大きい（1u サフィックスが付いた関数の場合）、またはコードブックのサイズが 256 より大きい（8u サフィックスが付いた関数の場合）。

FormVectorVQ

インデックスに基づいて、コードブックから
マルチストリーム・ベクトルを作成する。

```
IppStatus ippsFormVectorVQ_16s16s(const Ipp16s* pSrc, int srcStep,
    Ipp16s* pDst, int dstStep, int height, const IppsCdbkState_16s**
    pCdbks, int nStream);
IppStatus ippsFormVectorVQ_8u16s(const Ipp8u* pSrc, int srcStep,
    Ipp16s* pDst, int dstStep, int height, const IppsCdbkState_16s**
    pCdbks, int nStream);
IppStatus ippsFormVectorVQ_1u16s(const Ipp8u* pSrc, int srcBitStep,
    Ipp16s* pDst, int dstStep, int height, const IppsCdbkState_16s**
    pCdbks, int nStream);
IppStatus ippsFormVectorVQ_16s32f(const Ipp16s* pSrc, int srcStep,
    Ipp32f* pDst, int dstStep, int height, const IppsCdbkState_32f**
    pCdbks, int nStream);
IppStatus ippsFormVectorVQ_8u32f(const Ipp8u* pSrc, int srcStep,
    Ipp32f* pDst, int dstStep, int height, const IppsCdbkState_32f**
    pCdbks, int nStream);
IppStatus ippsFormVectorVQ_1u32f(const Ipp8u* pSrc, int srcBitStep,
    Ipp32f* pDst, int dstStep, int height, const IppsCdbkState_32f**
    pCdbks, int nStream);
```

引数

pCdbks	コードブック構造体 [*nStream*] へのポインタ。
pSrc	インデックス・ベクトル [*height*srcStep*] へのポインタ。
pDst	作成されたベクトル [*height*dststep*] へのポインタ。
srcStep	インデックス・ベクトル *pSrc* の行のステップ。
srcBitstep	インデックス・ベクトル *pSrc* の行のステップ（ビット単位）。
dstStep	作成されたベクトル *pDst* の行のステップ。

height	ベクトル *pSrc* の行数。
nStream	ストリームの数。

説明

関数 ippsFormVectorVQ は、ippsr.h ファイルで宣言される。この関数は、インデックス *pSrc* に基づいて、コードブック *pCdbks* からマルチストリーム・ベクトル *pDst* を作成する。各ストリームの長さは、対応するコードブック・ベクトルの長さに等しいと見なされる。
1u サフィックスが付いた関数の場合、各ストリーム・インデックスはパック形式と見なされる。各ストリームは、そのストリームのコードブック・インデックスを表現するのに必要なビット数を使用する。
ippsFormVector、ippsSplitVQ 関数も参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCdbk*、*pCdbk*[*k*]、*pSrc*、または *pDst* が NULL。
ippStsSizeErr	エラー。*srcStep*、*dstStep*、*height*、または *nStream* がゼロ以下、コードベクトルの長さの和が *dstStep* より大きい、*nStream* が *srcStep* より大きい(16s または 8u サフィックスが付いた関数の場合)、またはインデックスを表現するのに必要なビット数が *srcStep* より大きい (1u サフィックスが付いた関数の場合)。

ポリフェーズ・リサンプリング

この項で説明するインテル® IPP 関数は、データ・リサンプリングで Kaiser(カイザー)窓のポリフェーズ・フィルタをビルド、適用、解放する。Fixed サフィックスの付いた関数は、固定の有理リサンプリング係数のための関数であり、高速な処理を実現する。このサフィックスが付いていない関数は、フィルタ係数の線形補間を使用した汎用のリサンプリング・フィルタをビルドし、変数係数を使用できる。

ResamplePolyphaseInitAlloc

ポリフェーズ・データ・リサンプリングの
構造体を初期化する。

```
IppStatus ippsResamplePolyphaseInitAlloc_16s(IppsResamplePolyphaseSpec_16s**
    pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha,
    IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseInitAlloc_32f(IppsResamplePolyphaseSpec_32f**
    pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha,
    IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseFixedInitAlloc_16s(
    IppsResamplePolyphaseSpecFixed_16s** pSpec, int inRate, int
    outRate, int len, Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm
    hint);
IppStatus ippsResamplePolyphaseFixedInitAlloc_32f(
    IppsResamplePolyphaseSpecFixed_32f** pSpec, int inRate, int
    outRate, int len, Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm
    hint);
```

引数

window	理想的なローパス・フィルタ窓のサイズ。
nStep	フィルタ係数の離散ステップ。
rollf	フィルタのロールオフ周波数。
alpha	Kaiser（カイザー）窓のパラメータ。
inRate	固定係数を使用したリサンプリングの入力レート。
outRate	固定係数を使用したリサンプリングの出力レート。
len	固定係数を使用したリサンプリングのフィルタの長さ。
pSpec	生成するリサンプリング・ステート構造体へのポインタ。
hint	特別のコードの使用を指定する。*hint* 引数の値に関する詳細は、表 7-3 の「flag 引数と hint 引数」に掲載されている。

説明

これらの関数は、ippsr.h ファイルで宣言される。関数 ippsResamplePolyphaseInitAlloc は、理想的なローパス・フィルタを使用してデータ・リサンプリングの構造体を作成し、パラメータ *alpha* と幅 *widow* を指定した Kaiser（カイザー）窓を適用する。

$$h[i] = \begin{cases} \dfrac{rollf}{nStep}, & i = 0 \\ \dfrac{\sin(rollf \cdot i\pi / nStep)}{i\pi / nStep}, & i \neq 0 \end{cases}, \quad |i| \leq window \cdot nStep \quad , \qquad 0 < rollf \leq 1$$

作成した構造体は、任意順序のリサンプリング係数を指定した関数 ippsResample により、入力サンプルをリサンプルするのに使用できる。線形補間は、各出力サンプルのフィルタ係数を計算するときに使用される。フィルタのサイズは、リサンプリング係数によって異なる。

関数 ippsResamplePolyphaseFixedInitAlloc は、*inRate/outRate* と等しい係数を使用してデータ・リサンプリング用の構造体を作成する。上記の式で使用される *window* と *nStep* の値は、次に示すフィルタの長さを提供する。

$$flen = \min_{l \geq len, l\%4=0} l$$

また、ゼロ・フェーズの場合は、$flen+1$ となる。

$$fnum = outRate / \mathrm{HHO}(inRate, outRate)$$

上記のフィルタは異なるフェーズで作成される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSpec* ポインタが NULL。
ippStsSizeErr	エラー。*inRate*、*outRate*、*nStep* または *len* がゼロ以下。
ippStsBadArgErr	エラー。*rollf* がゼロ以下か 1 より大きい、*alpha* が 1 より小さい、あるいは、*window* が 2/*nStep* より小さい。
ippStsMemAllocErr	メモリの割り当てエラー。

ResamplePolyphaseFree

ポリフェーズ・データ・リサンプリングの
構造体を解放する。

```
IppStatus ippsResamplePolyphaseFree_16s(IppsResamplePolyphaseSpec_16s*
    pSpec);
IppStatus ippsResamplePolyphaseFree_32f(IppsResamplePolyphaseSpec_32f*
    pSpec);
IppStatus
    ippsResamplePolyphaseFixedFree_16s(IppsResamplePolyphaseSpecFixed_16s*
    pSpec);
IppStatus
    ippsResamplePolyphaseFixedFree_32f(IppsResamplePolyphaseSpecFixed_32f*
    pSpec);
```

引数

pSpec	リサンプリング・ステート構造体へのポインタ。

説明

関数 `ippsResamplePolyphaseFree` と `ippsResamplePolyphaseFixedFree` は、`ippsr.h` ファイルで宣言される。これらの関数は、`ippsResamplePolyphaseInitAlloc` または `ippsResamplePolyphaseFixedInitAlloc` を使用して、割り当てたすべてのメモリを解放することでリサンプリング構造体を閉じる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSpec* ポインタが NULL。

ResamplePolyphase

ポリフェーズ・フィルタを使用して
入力データをリサンプリングする。

```
IppStatus ippsResamplePolyphase_16s(const Ipp16s *pSrc, int len, Ipp16s
    *pDst, int *pOutlen, Ipp64f factor, Ipp32f norm, Ipp64f *pTime,
    const IppsResamplePolyphaseSpec_16s *pSpec);
```

```
IppStatus ippsResamplePolyphase_32f(const Ipp32f *pSrc, int len, Ipp32f
    *pDst, int *pOutlen, Ipp64f factor, Ipp32f norm, Ipp64f *pTime,
    const IppsResamplePolyphaseSpec_32f *pSpec);
IppStatus ippsResamplePolyphaseFixed_16s(const Ipp16s *pSrc, int len,
    Ipp16s *pDst, int *pOutlen, Ipp32f norm, Ipp64f *pTime, const
    IppsResamplePolyphaseSpecFixed_16s *pSpec);
IppStatus ippsResamplePolyphaseFixed_32f(const Ipp32f *pSrc, int len,
    Ipp32f *pDst, int *pOutlen, Ipp32f norm, Ipp64f *pTime, const
    IppsResamplePolyphaseSpecFixed_32f *pSpec);
```

引数

pSpec	リサンプリング・ステート構造体へのポインタ。
pSrc	入力ベクトルへのポインタ。
pDst	出力ベクトルへのポインタ。
len	リサンプルする入力ベクトルの数。
norm	出力サンプルのノルム係数。
factor	リサンプリング係数。
pTime	リサンプリングの開始時間へのポインタ (入力ベクトル要素単位)。
pOutlen	計算した出力ベクトル要素の数。

説明

関数 `ippsResamplePolyphase` と `ippsResamplePolyphaseFixed` は、`ippsr.h` ファイルで宣言される。これらの関数は、周波数を変更する入力ベクトルからデータを変換する。出力周波数と入力周波数の比率は、関数 `ippsResamplePolyphase` の *factor* 引数によって定義される。関数 `ippsResamplePolyphaseFixed` では、この比率はリサンプリング構造体を作成するときに定義される。値 *pTime*`[0]` は、最初の出力サンプルが計算される時間を定義する。

フィルタのヒストリ・データは、*pTime*`[0]` 以下のインデックスの入力ベクトルに格納される。ヒストリの長さは、関数 `ippsResamplePolyphaseFixed` では *flen/2* に等しく、関数 `ippsResamplePolyphase` では

$$\left[\frac{1}{2} window \cdot \max(1,1/factor)\right] + 1$$

と等しい。入力ベクトルには、インデックスと同じ数の要素が格納されていなければならない。このインデックスは、最後の要素の右フィルタ側の *pTime*`[0]+`*len* より大きい。

関数を実行した後は、時間の値が更新され、*pOutlen*[0] には計算された出力サンプルの数が格納される。

出力サンプルは、飽和する前に $norm \cdot \min(1, factor)$ が掛けられる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSpec*、*pSrc*、*pDst*、*pTime*、または *pOutlen* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsBadArgErr	エラー。*factor* がゼロ以下。

例 8-3　PCM モノラル入力ファイルのリサンプリング

```
void resampleIPP(
   int       inRate,     // input frequency
   int       outRate,    // output frequency
   FILE     *infd,       // input pcm file
   FILE     *outfd)      // output pcm file
{
   short *inBuf,*outBuf;
   int bufsize=4096;
   int history=128;
   double time=history;
   int lastread=history;
   int inCount=0,outCount=0,inLen,outLen;
   IppsResamlePolyphaseSpecFixed_16s *state;


ippsResamplePolyphaseFixedInitAlloc_16s(&state,inRate,outRate,2*history,
                                  0.95f,9.0f,ippAlgHintAccurate);
   inBuf=ippsMalloc_16s(bufsize+history+2);

outBuf=ippsMalloc_16s((int)((bufsize-history)*outRate/(float)inRate+2));

   ippsZero_16s(inBuf,history);
   while
((inLen=fread(inBuf+lastread,sizeof(short),bufsize-lastread,infd))>0)
{
      inCount+=inLen;
      lastread+=inLen;
```

```
ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-history-(int)time,
                             outBuf,0.98f,&time,&outLen);
      fwrite(outBuf,outLen,sizeof(short),outfd);
      outCount+=outLen;

ippsMove_16s(inBuf+(int)time-history,inBuf,lastread+history-(int)time);
      lastread-=(int)time-history;
      time-=(int)time-history;
   }
   ippsZero_16s(inBuf+lastread,history);
   ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-(int)time,
                             outBuf,0.98f,&time,&outLen);
   fwrite(outBuf,outLen,sizeof(short),outfd);
   outCount+=outLen;

   printf("%d inputs resampled to %d outputs\n",inCount,outCount);
   ippsFree(outBuf);
   ippsFree(inBuf);
   ippsResamplePolyphaseFixedFree_16s(state);
}
```

アドバンスト・オーロラ関数

この項では、ETSI ES 202 050 V1.1.1 規格（[ES202] を参照）に基づいた音声処理と圧縮アルゴリズムをサポートするインテル® IPP 関数について説明する。

SmoothedPowerSpectrum_Aurora

FFT 出力の平滑化された大きさを計算する。

```
IppStatus ippsSmoothedPowerSpectrum_Aurora_16s(const Ipp16s* pSrc,
    Ipp16s* pDst, int len);
IppStatus ippsSmoothedPowerSpectrum_Aurora_32f(const Ipp32f* pSrc,
    Ipp32f* pDst, int len);
```

引数

pSrc	PERM 形式の入力ベクトル [*len*] へのポインタ。
pDst	出力ベクトル [*len*/4+1] へのポインタ。

len 入力ベクトルの長さ（4 の倍数）。

説明

関数 ippsSmoothedPowerSpectrum_Aurora は、ippsr.h ファイルで宣言される。この関数は、高速フーリエ変換（FFT）出力ベクトルの平滑化された 2 乗の値を PERM 形式で計算する（[ES202] , 5.1.3-5.1.4）。

$$pDst[0] = \left(pSrc[0]^2 + pSrc[2]^2 + pSrc[3]^2\right)/2$$

$$pDst[i] = \left(pSrc[4\cdot i]^2 + pSrc[4\cdot i+1]^2 + pSrc[4\cdot i+2]^2 + pSrc[4\cdot i+3]^2\right)/2$$

$$pDst[len/4] = pSrc[1]^2$$

ここで、$0 < i < len/4$ である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下、または *len* が 4 の倍数ではない。

NoiseSpectrumUpdate_Aurora

ノイズ・スペクトルを更新する。

```
IppStatus ippsNoiseSpectrumUpdate_Aurora_32f(const Ipp32f* pSrc, const
    Ipp32f* pSrcNoise,  Ipp32f* pDst, int len);
```

引数

pSrc	パワー・スペクトル密度平均値の入力ベクトル [*len*] へのポインタ。
pSrcNoise	直前のノイズレス信号スペクトルの入力ベクトル [*len*] へのポインタ。
pDst	向上した伝達関数の出力ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。

説明

関数 ippsNoiseSpectrumUpdate は、ippsr.h ファイルで宣言される。この関数は、次の式に基づいて、ノイズ・リダクションの第 2 段階におけるノイズ・スペクトルの推定を更新する（[ES202], 5.1.5）。

$$pDst[i] = pSrcNoise[i] \cdot \left(0.9 + \frac{0.1 \cdot pSrc[i]}{pSrc[i] + pSrcNoise[i]} \cdot \left(1 + \frac{pSrcNoise[i]}{pSrcNoise[i] + 0.1 \cdot pSrc[i]}\right)\right)$$

ここで、$0 \le i < len$ である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pSrcNoise*、または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

WienerFilterDesign_Aurora

適応 Wiener（ウィーナ）フィルタの向上した伝達関数を計算する。

```
IppStatus ippsWienerFilterDesign_Aurora_32f(const Ipp32f* pSrc, const
    Ipp32f* pNoise, const Ipp32f* pDen, Ipp32f* pDst, int len);
```

引数

pSrc	パワー・スペクトル密度平均値の平方根の入力ベクトル [*len*] へのポインタ。
pNoise	ノイズ・スペクトル推定の平方根の入力ベクトル [*len*] へのポインタ。
pDen	直前のノイズレス信号スペクトルの平方根の入力ベクトル [*len*] へのポインタ。
pDst	向上した伝達関数の出力ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。

説明

関数 ippsWienerFilterDesign_Aurora は、ippsr.h ファイルで宣言される。この関数は、次の式に従って、入力ベクトルを処理する（[ES202], 5.1.5）。

1. ノイズレス信号が推定される。

   $$p[i] = 0.98 \cdot pDen[i] + 0.02 \cdot \max(pSrc[i] - pNoise[i], 0)$$

2. 事前 SNR（信号－ノイズ比）が計算される。

   $$\eta[i] = p[i] / pNoise[i]$$

3. 次に、フィルタ伝達関数

   $$H[i] = \frac{\eta[i]}{1 + \eta[i]}$$

   が使用され、ノイズレス信号スペクトルの推定が向上する。

   $$p2[i] = H[i] \cdot pSrc[i]$$

4. 向上した事前 SNR は、次のように計算される。

   $$\eta 2[i] = \max(p2[i] / pNoise[i], 0.079432823)$$

5. 向上したフィルタ伝達関数は、次のようになる。

   $$pDst[i] = \frac{\eta 2[i]}{1 + \eta 2[i]}$$

ここで、$0 \le i < len$ である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pNoise*、*pDen*、または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsBadArg	エラー。平方根演算で負または NaN の引数が検出された。

MelFBankInitAlloc_Aurora

メル周波数フィルタ・バンク分析を
実行するための構造体を初期化する。

```
IppStatus ippsMelFBankInitAllocLow_Aurora_16s(IppsFBankState_16s**
    pFBank);
IppStatus ippsMelFBankInitAllocLow_Aurora_32f(IppsFBankState_32f**
    pFBank);
IppStatus ippsMelFBankInitAllocHigh_Aurora_16s(IppsFBankState_16s**
    pFBank);
IppStatus ippsMelFBankInitAllocHigh_Aurora_32f(IppsFBankState_32f**
    pFBank);
```

引数

pFBank	作成されるメルスケール・フィルタ・バンク構造体へのポインタ。

説明

関数 ippsMelFBankInitAlloc_Aurora は、ippsr.h ファイルで宣言される。この関数は、メル周波数フィルタ・バンク分析用の三角フィルタ・バンクを初期化する。Low サフィックスの付いた関数は、8 Khz データを処理する 23 個のフィルタで構成されたフィルタ・バンクをビルドする。High サフィックスの付いた関数は、16 Khz データを処理する高周波帯域用の 3 個のフィルタで構成されたフィルタ・バンクをビルドする（[ES202], 5.1.7)。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFBank* が NULL。

TabsCalculation_Aurora

残留信号フィルタのフィルタ係数を
計算する。

```
IppStatus ippsTabsCalculation_Aurora_16s_Sfs(const Ipp16s* pSrc,
    Ipp16s* pDst);
```

```
IppStatus ippsTabsCalculation_Aurora_32f(const Ipp32f* pSrc, Ipp32f*
    pDst);
```

引数

pSrc	メルスケール・フィルタ・バンク出力の入力ベクトル [10] へのポインタ。
pDst	出力フィルタ係数ベクトル [17] へのポインタ。

説明

関数 ippsTabsCalculation_Aurora は、ippsr.h ファイルで宣言される。この関数は、メルスケール・フィルタ・バンク出力の残留信号フィルタ係数を計算する（[ES202], 5.1.9-5.1.10）。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* が 23 または 26 に等しくない。

ResidualFilter_Aurora

ノイズ除去された波形信号を計算する。

```
IppStatus ippsResidualFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s*
    pDst, const Ipp16s* pTabs, int scaleFactor);
IppStatus ippsResidualFilter_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst,
    const Ipp32f* pTabs);
```

引数

pTabs	フィルタ係数ベクトル [17] へのポインタ。
pSrc	入力ベクトル [96] へのポインタ。
pDst	出力ベクトル [80] へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsResidualFilter_Aurora は、ippsr.h ファイルで宣言される。
この関数は、入力ベクトルをフィルタしてロー・バンドとハイ・バンドの部分を取得する（[ES202], 5.1.10）。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pTabs* が NULL。

WaveProcessing_Aurora

ノイズ・リダクションの後に波形データを処理する。

```
IppStatus ippsWaveProcessing_Aurora_32f(const Ipp32f* pSrc, Ipp32f*
    pDst);
```

引数

pSrc	入力ベクトル [200] へのポインタ。
pDst	出力ベクトル [200] へのポインタ。

説明

関数 ippsWaveProcessing_Aurora は、ippsr.h ファイルで宣言される。この関数は、次の式に従って入力ベクトルを処理する（[ES202], 5.2）。

1. 平滑化された Teager 演算子が計算される。

$$s[i] = \frac{1}{9}\sum_{k=-4}^{4} s_T[i] \quad , \qquad 0 \le i < 200$$

ここで、

$$s_T[i] = \left| pSrc[i]^2 - pSrc[\max(0, i-1)] \cdot pSrc[\min(199, i+1)] \right|$$

$0 \le i < 200$ である。

また、$i < 0$ and $200 \le i$ の場合、$s[i] = 0$ である。

2. ピーク

$$p[l_{\min}], \ldots\ p[l_{\max}]$$

が検出される。

$$p[0] = \underset{0 \le i < 200}{\operatorname{argmax}} s[i]$$

$$p[l+1] = \underset{\substack{p[l]+25 \le k < p[l]+80 \\ k < 200}}{\operatorname{argmax}} s[k] \ , \qquad l = 0, \ldots\ l_{\max} - 1$$

$$p[l-1] = \underset{\substack{p[l]-25 \ge k > p[l]-80 \\ k \ge 0}}{\operatorname{argmax}} s[k] \ , \qquad l = l_{\min} + 1, \ldots\ 0$$

3. 重み付け関数が計算される。

$$w[i] = \begin{cases} 1.2, & p[k] - 4 < i < p[k] + 0.8 \cdot]p[k+1] - p[k][\\ 1.0, & i = p[k] - 4, i = p[k] + 0.8 \cdot]p[k+1] - p[k][\\ 0.8 & \text{otherwise} \end{cases}$$

$l_{\min} \le k < l_{\max}$

4. 出力ベクトルは、次のように計算される。

$$pDst[i] = w[i] \cdot pSrc[i], \qquad 0 \le i < 200$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrc* または *pDst* が NULL。

LowHighFilter_Aurora

ロー・バンドとハイ・バンドのフィルタを
計算する。

```
IppStatus ippsLowHighFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s*
    pDstLow, Ipp16s* pDstHigh, int len, const Ipp16s* pTabs, int
    tapsLen, int scaleFactor);
IppStatus ippsLowHighFilter_Aurora_32f(const Ipp32f* pSrc, Ipp32f*
    pDstLow, Ipp32f* pDstHigh, int len, const Ipp32f* pTabs, int
    tapsLen);
```

引数

pSrc	入力ベクトル [*len*+*tapsLen*-1] へのポインタ。
pDstLow	低周波帯域の出力ベクトル [*len*/2] へのポインタ。
pDstHigh	高周波帯域の出力ベクトル [*len*/2] へのポインタ。
len	入力サンプル番号（偶数）。
pTabs	フィルタ係数ベクトル [*tapsLen*] へのポインタ。
tapsLen	フィルタ・タップ番号（偶数）。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsLowHighFilter_Aurora は、ippsr.h ファイルで宣言される。この関数は、入力ベクトルをフィルタしてロー・バンドとハイ・バンドの部分を取得する ([ES202], 5.1.10)。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDstLow*、*pDstHigh* または *pTabs* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下、*tapsLen* がゼロ以下、あるいは、*len* または *tapsLen* が偶数ではない。

HighBandCoding_Aurora

高周波帯域のエネルギー値をコードおよびデコードする。

```
IppStatus ippsHighBandCoding_Aurora_32f(const Ipp32f* pSrcHFB, const
    Ipp32f* pInSWP, const Ipp32f* pDSWP, Ipp32f* pDstHFB);
```

引数

pSrcHFB	入力高周波帯域エネルギー・ベクトル [3] へのポインタ。
pInSWP	入力信号平滑化パワー・スペクトル・ベクトル [65] へのポインタ。
pDSWP	ノイズ除去信号パワー・スペクトル・ベクトル [129] へのポインタ。
pDstHFB	出力コード高周波帯域ログ・エネルギー・ベクトル [3] へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsHighBandCoding_Aurora は、ippsr.h ファイルで宣言される。この関数は、次の式に従って高周波帯域エネルギーをコードおよびデコードする（[ES202], 5.5.3）。

1. 対数演算は、高周波帯域エネルギーに対して適用される。

$$S_{HFB}[k] = \ln pSrcHFB[k]\ , \qquad 0 \le k < 3$$

2. 入力信号パワー・スペクトルの補助ログ・エネルギーが計算される。

$$S_{LFB_aux}[l] = \max\left(-50, \ln\left(\sum_{i=n1[l]}^{n1[l+1]-1} pInSWP[i]\right)\right),\qquad 0 \le l < 3$$

$$n1[0] = 33,\ n1[1] = 39,\ n1[2] = 49,\ n1[3] = 65$$

3. エネルギーは、次のようにコードされる。

$$code[k,l] = S_{LFB_aux}[k] - S_{HFB}[l]\ , \qquad 0 \le k,l < 3$$

4. デコード用の補助バンドが計算される。

$$S_{Dec_aux}[l] = \max\left(-50, \ln\left(\frac{1}{2}\sum_{i=n2[l]}^{n2[l+1]-1} pDSWP[i]\right)\right), \qquad 0 \le l < 3$$

$$n2[0] = 66,\ n2[1] = 77,\ n2[2] = 97,\ n2[3] = 129$$

5. デコードされた高周波帯域エネルギーは、次のように計算される。

$$pDstHFB[k] = \sum_{l=0}^{2} w[l] \cdot \left(S_{Dec_aux}[l] - code[l,k]\right), \qquad 0 \le k < 3$$

$$w[0] = 0.1,\ w[1] = 0.2,\ w[2] = 0.7$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcHFB*、*pInSWP*、*pDSWP*、または *pDstHFB* が NULL。
ippStsBadArg	エラー。対数演算で負または NaN の引数が検出された。

BlindEqualization_Aurora

ケプストラム係数を均等にする。

```
IppStatus ippsBlindEqualization_Aurora_32f(const Ipp32f* pRefs, Ipp32f*
    pCeps, Ipp32f* pBias, int len, Ipp32f val);
```

引数

pRefs	参照ケプストラムの入力ベクトル [*len*] へのポインタ。
pCeps	ケプストラムの入力および出力ベクトル [*len*] へのポインタ。
pBias	バイアスの入力および出力ベクトル [*len*] へのポインタ。
len	ケプストラム係数の数。
val	ログ・エネルギーの値。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsBlindEqualization_Aurora は、ippsr.h ファイルで宣言される。この関数は、LMS アルゴリズムを使用してケプストラム係数を均等にする（[ES202], 5.4）。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pRefs*、*pCeps*、または *pBias* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

DeltaDelta_Aurora

ETSI ES 202 050 規格に基づいて、
1 次導関数と 2 次導関数を計算する。

```
IppStatus ippsDeltaDelta_Aurora_16s_D2Sfs(const Ipp16s* pSrc, Ipp16s*
    pDst, int dstStep, int height, int deltaMode, int scaleFactor);
IppStatus ippsDeltaDeltaMul_Aurora_16s_D2Sfs(const Ipp16s* pSrc, const
    Ipp16s* pVal, Ipp16s* pDst, int dstStep, int height, int deltaMode,
    int scaleFactor);
IppStatus ippsDeltaDelta_Aurora_32f_D2(const Ipp32f* pSrc, Ipp32f*
    pDst, int dstStep, int height, int deltaMode);
IppStatus ippsDeltaDeltaMul_Aurora_32f_D2(const Ipp32f* pSrc, const
    Ipp32f* pVal, Ipp32f* pDst, int dstStep, int height, int
    deltaMode);
```

引数

pSrc	入力特徴シーケンス [*height**14] へのポインタ。
pDst	出力特徴シーケンス [*height***dstStep*] へのポインタ。
dstStep	出力シーケンス *pDst* の特徴ベクトルの長さ。
height	特徴ベクトルの数。
deltaMode	実行モード。
pVal	デルタ係数ベクトル [39] へのポインタ。
scaleFactor	第 2 章の「整数のスケーリング」を参照。

説明

関数 ippsDeltaDelta_Aurora と ippsDeltaDeltaMul_Aurora は、ippsr.h ファイルで宣言されている。これらの関数は、ETSI ES 202 050 規格に基づいて完全な特徴ベクトルを計算する。
サイズが 14 の入力ベクトルには、$c_1,...c_{12}, c_0, \ln E$ の値が格納される。最初に、入力特徴ベクトルが出力シーケンスにコピーされる。次に、1 次導関数と 2 次導関数が計算される（[ES202], 9.1-9.2）。

この関数は、デルタ係数に関する次の制約条件を前提とする。

ippsDeltaDelta_Aurora 関数では、$val[j] = pVal[j]$
ippsDeltaDeltaMul_Aurora 関数では、$val[j] = 1$。

$vel[-4] = -1.0,\ vel[-3] = -0.75,\ vel[-2] = -0.5,\ vel[-1] = -0.25,\ vel[0] = 0.0,$
$vel[1] = 0.25,\ vel[2] = 0.5,\ vel[3] = 0.75,\ vel[4] = 1.0$。

$acc[-4] = 1.0,\ acc[-3] = 0.25,\ acc[-2] = -0.285714,\ acc[-1] = -0.607143,$
$acc[0] = -0.714286,\ acc[1] = -0.607143,\ acc[2] = -0.285714,\ acc[3] = 0.25,\ acc[4] = 1.0$。

実行モード *deltaMode* は、ベースとなる特徴のコピーと導関数の計算プロセスを制御する。*deltaMode* の許容される値と、各値に対応する関数実行ロジックは、次のとおりである。

1. *deltaMode* が IPP_DELTA_BEGIN|IPP_DELTA_END の場合

関数は、オフラインのデルタ特徴計算を実行する。計算の実行時にすべてのベースとなる特徴が利用可能になっていると見なされる。最初に、次の式に従って、入力ストリーム *pSrc* から出力ストリーム *pDst* にベースとなる特徴がコピーされる。

$$pDst[i \cdot dstStep + j] = val[j] \cdot pSrc[i \cdot 14 + j] \quad , \quad 0 \le j < 12$$

$$pDst[i \cdot dstStep + 12] = val[12] \cdot (pSrc[i \cdot 14 + 12] \cdot 0.623 + pSrc[i \cdot 14 + 13] \cdot 0.4)$$

$$0 \le i < height \qquad (8.11)$$

続いて、次の式に従って 1 次導関数と 2 次導関数が計算される。

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \sum_{k=-4}^{4} vel[k] \cdot pDst[max(0, min(i + k, height - 1)) \cdot dstStep + j]$$

および

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26]\sum_{k=-4}^{4} acc[k] \cdot pDst[max(0, min(i + k, height - 1)) \cdot dstStep + j]$$

$0 \leq i < height,\ 0 \leq j < 13$ の場合。

2. *deltaMode* が 0 の場合

オンラインのデルタ特徴計算が実行される。入力特徴は、連続的なストリームの現在のセグメントである。関数 `ippsDeltaDelta_Aurora` は、現在の入力に従って部分的なデルタ特徴を計算する。最初に、次の式に従ってベースとなる特徴がコピーされる。

$$pDst[(i + 8) \cdot dstStep + j] = val[j] \cdot pSrc[i \cdot 14 + j] \quad , \ 0 \leq j < 12$$

$$pDst[(i + 8) \cdot dstStep + 12] = val[12] \cdot (pSrc[i \cdot 14 + 12] \cdot 0.623 + pSrc[i \cdot 14 + 13] \cdot 0.4)$$

$$0 \leq i < height \qquad (8.12)$$

続いて、次の式に従って 1 次導関数と 2 次導関数が計算される。

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[(i + k) \cdot dstStep + j]$$

また、

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[(i + k) \cdot dstStep + j]$$

$0 \leq i - 4 < height,\ 0 \leq j < 13$ の場合。

この実行モードでは、$pDst[i \cdot dstStep + j]$ 内のベースとなる特徴、

$pDst[k \cdot dstStep + 13 + l]$ 内の 1 次導関数と 2 次導関数は、$0 \leq j < 13$, $0 \leq i < 8$, $0 \leq k < 26$, $0 \leq l < 4$ の場合における前回の導関数計算で利用可能とみなされる。

3. *deltaMode* が `IPP_DELTA_BEGIN` の場合

入力ストリームの開始点がわかっている、部分的なオンライン・デルタ特徴計算を実行する。最初に、式（8.11）に従ってベースとなる特徴がコピーされる。続いて、次の式に従って 1 次導関数と 2 次導関数が計算される。

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[\max(0, i + k) \cdot dstStep + j]$$

また、

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[\max(0, i + k) \cdot dstStep + j]$$

$0 \le i < height - 4,\ 0 \le j < 13$ の場合。

4. *deltaMode* が `IPP_DELTA_END` の場合

入力ストリームの終了点がわかっている、部分的なオンライン・デルタ特徴計算を実行する。最初に、式（8.12）に従ってベースとなる特徴がコピーされる。続いて、次の式に従って 1 次導関数と 2 次導関数が計算される。

$$pDst[i \cdot dstStep + 13 + j] = val[j + 13] \cdot \sum_{k=-4}^{4} vel[k] \cdot pDst[min(i + k, height - 1) \cdot dstStep + j]$$

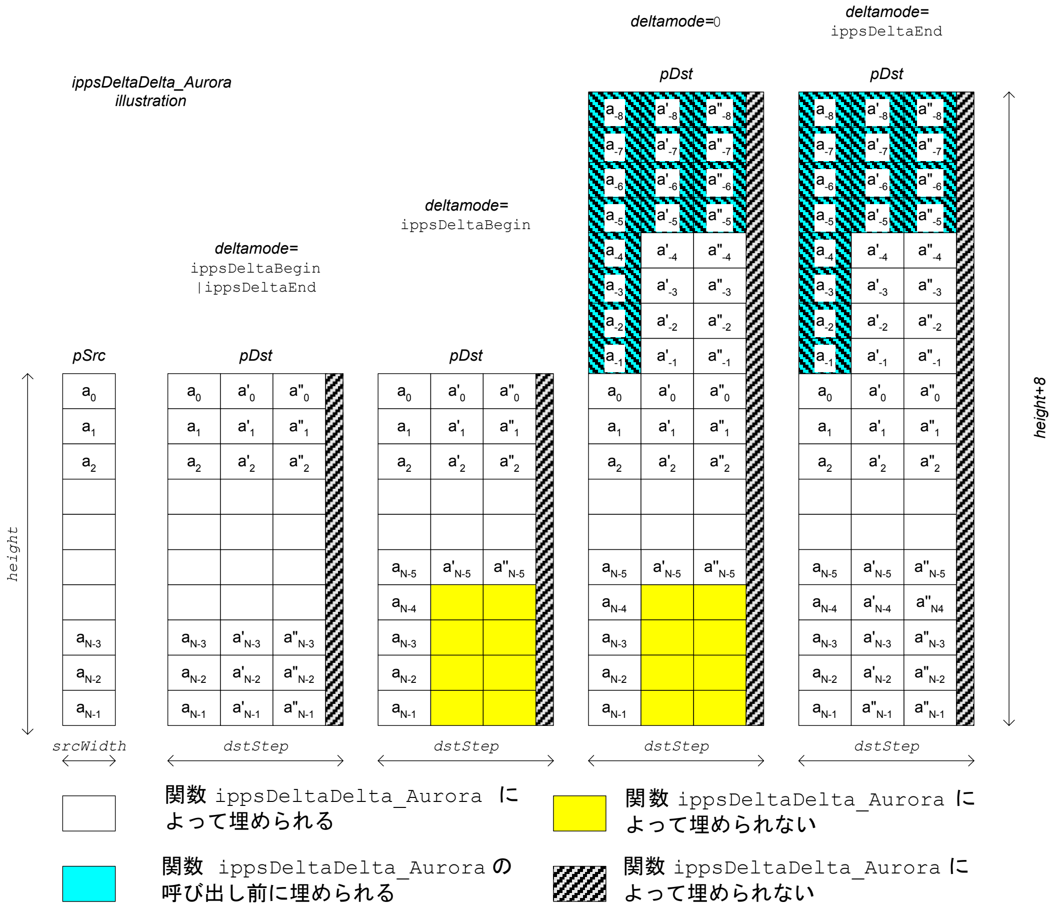
また、

$$pDst[i \cdot dstStep + 26 + j] = val[j + 26] \cdot \sum_{k=-4}^{4} acc[k] \cdot pDst[min(i + k, height - 1) \cdot dstStep + j]$$

$0 \le i - 4 < height + 4,\ 0 \le j < 13$ の場合。この実行モードでは、$pDst[i \cdot dstStep + j]$ 内のベースとなる特徴、$pDst[k \cdot dstStep + 13 + l]$ 内の 1 次導関数と 2 次導関数は、$0 \le j < 13$, $0 \le i < 8$, $0 \le k < 26$, $0 \le l < 4$ の場合における前回の導関数計算で利用可能とみなされる。

次の図は、上の 4 つのデルタ計算モードを示している。

図 8-6　ippsDeltaDelta_Aurora 関数の実行モード



戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pVal* が NULL。
ippStsSizeErr	エラー。IPP_DELTA_BEGIN が設定されていて、*height* が 8 以下、あるいは、IPP_DELTA_BEGIN が設定されてなく、*height* がゼロより小さい。
ippStsStrideErr	エラー。*dstStep* が 39 より小さい。

VADGetBufSize_Aurora

VAD 決定のメモリ・サイズをクエリする。

```
IppStatus ippsVADGetBufSize_Aurora_32f(int* pSize);
```

引数

pSize	VAD 決定で必要なメモリ・サイズの出力値へのポインタ。

説明

関数 ippsVADGetBufSize_Aurora は、ippsr.h ファイルで宣言される。この関数は、ユーザによって割り当てられるメモリのサイズを返す。*pSize*[0] バイトのメモリ・ブロックは、ippsVadInit_Aurora 関数が VAD アルゴリズムを初期化するときに使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSize* が NULL。

VADInit_Aurora

VAD 構造体のサイズを取得する。

```
IppStatus ippsVADInit_Aurora_32f(char* pVADmem);
```

引数

pVADmem	VAD 決定メモリへのポインタ。

説明

関数 ippsVADInit_Aurora は、ippsr.h ファイルで宣言される。この関数は、VAD 決定処理を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pVADmem* が NULL。

VADDecision_Aurora

VAD 決定を行う。

```
IppStatus ippsVADDecision_Aurora_32f(const Ipp32f* pCoeff, const
    Ipp32f* pTrans, IppVADDecision_Aurora * pRes, int nbSpeechFrame,
    char* pVADmem);
```

引数

pCoeff	メルワープ Wiener（ウィーナ）フィルタ係数の入力ベクトル [25] へのポインタ。
pTrans	Wiener（ウィーナ）フィルタ伝達関数の入力ベクトル [64] へのポインタ。
pRes	VAD 決定へのポインタ（音声が検出されると「1」、それ以外は「0」）。
nbSpeechFrame	音声フレーム・ハングオーバー・カウンタ。
pVADmem	VAD 決定メモリへのポインタ。

説明

関数 ippsVADDecision_Aurora は、ippsr.h ファイルで宣言される。この関数は、([ES202]、Annex A）に従って入力フレームの VAD 決定を行う。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCoeff*、*pTrans*、*pRes*、または *pVADmem* が NULL。

VADFlush_Aurora

ゼロ入力フレームの VAD 決定を行う。

```
IppStatus ippsVADFlush_Aurora_32f(IppVADDecision_Aurora* pRes, char*
    pVADmem);
```

引数

pCoeff	メルワープ Wiener（ウィーナ）フィルタ係数の入力ベクトル `[25]` へのポインタ。
pTrans	Wiener（ウィーナ）フィルタ伝達関数の入力ベクトル `[64]` へのポインタ。
pRes	VAD 決定へのポインタ（音声が検出されると「1」、それ以外は「0」)。
pVADmem	VAD 決定メモリへのポインタ。

説明

関数 `ippsVADFlush_Aurora` は、`ippsr.h` ファイルで宣言される。この関数は、ゼロ・フレームの VAD 決定を行う。VAD 決定は、音声が終了した後に使用される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pCoeff*、*pTrans*、*pRes*、または *pVADmem* が `NULL`。

Ephraim-Malah ノイズ・サプレッサ

この項では、[Eph84] で説明されている Ephraim-Malah ノイズ・サプレッサ（EMNS）の実装を行うインテル® IPP 関数について説明する。プリミティブは主に、計算に時間のかかる演算を行う。

EMNS は、短時間の音声スペクトル振幅推定の最小平均 2 乗誤差を最小限にする、周波数領域ノイズ・サプレッション・アルゴリズムである。

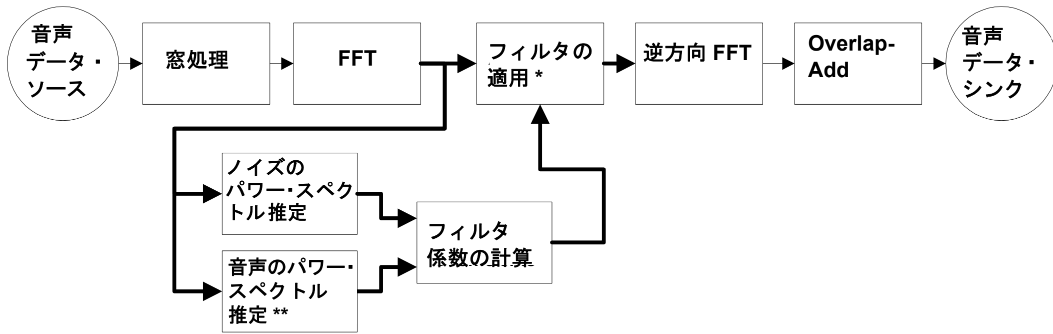
インテル IPP EMNS プリミティブは、次の機能をサポートする。

- フィルタの更新
- ノイズ・フロア推定

ノイズ・サプレッサ・アーキテクチャ

Ephraim-Malah ノイズ・サプレッサの基本要素を示す 図 8-7 では、音声信号ストリームのノイズ削減を適用するステップについて説明する。

図 8-7　Ephraim-Malah ノイズ・サプレッション・システムの基本要素



* フィルタの適用とは、実数値フィルタ係数を各 FFT ビンに掛けることである

** 音声のパワー・スペクトラル推定は、通常スペクトラルの減算で実行される

Ephraim-Malah ノイズ・サプレッサの詳細

アルゴリズムのステップ

1. **新規入力ブロック。**最後に取得した N/2 入力サンプル ζ_n と以前に取得した N/2 入力サンプル ζ_{n-1} は、新しい入力ブロック $\mathbf{z}_n$ を構成する。

$$\mathbf{z}_n = \begin{bmatrix} \zeta_{n-1} \\ \zeta_n \end{bmatrix}$$

2. **窓 DFT。**入力ブロックには、窓関数の平方根が掛けられる。窓関数は、1 つ目の式の値と 2 つ目の式の値をすべて加算すると、合計が 1 になるように制約される。便利な窓関数である三角窓関数は、次のように定義される。

$$w(m) = \begin{cases} \dfrac{m+0.5}{N/2} & \text{for } m = 0,\ldots,N/2-1 \\ 1 - w(m - N/2) & \text{for } m = N/2,\ldots,N-1 \end{cases}$$

入力の離散フーリエ変換は、次のように計算される。

$$\mathbf{Z}_n = \mathbf{F}\left(\mathbf{z}_n \bullet \sqrt{\mathbf{w}}\right)$$

ここで、• は乗算記号を意味し、$\sqrt{\mathrm{w}}$ はエントリ **w** の平方根を格納しているベクトルを意味する。F は、エントリ f(m,n)=exp(-j2ππmn/N) を含むフーリエ変換行列を意味する。N は変換のサイズを示す。

3. **ノイズを含む音声 PSD の更新。**ノイズを含む音声の大きさを 2 乗したスペクトル成分は、ノイズを含む音声のパワー・スペクトル推定を提供するために平均化される。

$$\mathbf{P}_n^z(k) = \beta_n \cdot \left|\mathbf{Z}_n(k)\right|^2 + (1-\beta_n) \cdot \mathbf{P}_{n-1}^z(k)$$

アダプティブ・ステップ・サイズは、次のように定義される。

$$\beta_n = \beta_{\min} + \rho_{n-1}^y(\beta_{\max} - \beta_{\min})$$

ここで、$\beta_{\min} = 0.9$、$\beta_{\max} = 1.0$、および ρ_{n-1}^y は

ビン k に音声が存在する尤度を示す。

4. **クリーンな音声 PSD の更新**。クリーンな音声のパワー・スペクトル成分は、スペクトルの減算と平均化で取得する。

$$\mathbf{P}_n^y(k) = \alpha_n \cdot \left|\hat{\mathbf{Y}}_{n-1}(k)\right|^2 + (1-\alpha_n) \cdot \psi_0\left(\mathbf{P}_n^z(k) - \mathbf{P}_{n-1}^v(k)\right)$$

しきい値演算 ψ は、次のように定義される。

$$\psi_c(x) = \begin{cases} c, & x \le c \\ x, & x > c \end{cases}$$

アダプティブ・ステップ・サイズは、次のように定義される。

$$\alpha_n = \alpha_{\min} + (1-\rho_{n-1}^y)(\alpha_{\max} - \alpha_{\min})$$

ここで、$\alpha_{\min} = 0.91$、$\alpha_{\max} = 0.95$、および ρ_{n-1}^y は

ビン k に音声が存在する尤度を示す。

注 : この計算では、直前のノイズのパワー・スペクトル成分が使用される。ノイズ・フロア推定量がその他のアルゴリズムから独立している場合、現在のフレームのノイズ推定を使用することができる。本書では、ノイズ・フロア推定量が依存するとして説明する。

5. **Wiener（ウィーナ）フィルタの重み。**Ephraim-Malah サプレッション・ルールを計算するために使用した重みのうちの 1 つは、実際は Wiener（ウィーナ）フィルタである（異なるノイズ・サプレッション・ルール）。Wiener（ウィーナ）フィルタの重みは、次の式から得られる。

$$\mathbf{W}_n^y(k) = \psi_{W_{\min}}\left(\frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^y(k) + \mathbf{P}_{n-1}^v(k)}\right)$$

$W_{\min}$ は [Cap94] で定義されたしきい値に対応する。ここで、次に示す事前 SNR

$$\Re_n^{prio}(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_{n-1}^v(k)}$$

の $\Re_{\min dB}^{prio} = -15.0\,\mathrm{dB}$ 下限値では、音楽ノイズを避けることを推奨する。これは次の式に相当する。

$$W_{\min} = \frac{1}{1 + 10^{\frac{\Re_{\min dB}^{prio}}{10}}}$$

Wiener（ウィーナ）フィルタが事前 SNR として記述されている場合、Wiener（ウィーナ）フィルタの計算は、ルックアップ・テーブルで置換することができる。これは、除算が多いプロセッサにおいて利点となる。

6. **事後 SNR の更新。**各周波数ビンの事後 SNR（信号－ノイズ比）は、次のように定義される。

$$\Re_n^{post}(k) = \frac{\mathbf{P}_n^z(k)}{\mathbf{P}_{n-1}^v(k)}$$

7. **Ephraim-Malah フィルタの重みの更新**。Ephraim-Malah フィルタの重みは、次の式から得られる。

$$\mathbf{H}_n^y(k) = \frac{1}{\Re_n^{post}(k)} \cdot M\left(\mathbf{W}_n^y(k)\Re_n^{post}(k)\right)$$

ここで、$M(\cdot)$ は関数である。

$$M(\theta) = \frac{1}{2} \cdot \sqrt{\pi\theta} \cdot e^{-\frac{\theta}{2}}\left[(1+\theta) \cdot I_0\left(\frac{\theta}{2}\right) + \theta \cdot I_1\left(\frac{\theta}{2}\right)\right]$$

Bessel（ベッセル）関数の評価が多いプロセッサでは、この関数をルックアップ・テーブルで置換することができる。

8. **ノイズ PSD 推定の更新**。ノイズのパワー・スペクトル推定量を使用して、$\mathbf{P}_n^y(k)$ を計算する必要がある。最小統計および最適スムージング

   [Mar01] に基づいた推定量の 1 つは、Martin 手法である。

9. **音声が存在する尤度の更新。**音声が存在する可能性は、直接計算されない。この可能性は、音声エネルギー全体の MMSE（Wiener）推定量によっておおよそで計算される。

$$\rho_n^y = \frac{\sum_{k=0}^{N/2} \mathbf{P}_n^y(k)}{\sum_{k=0}^{N/2} \mathbf{P}_n^y(k) + \sum_{k=0}^{N/2} \mathbf{P}_n^v(k)}$$

10. **ヒューリスティックの追加変更をフィルタ係数に適用する**。

フィルタ係数 $\mathbf{H}_n^y(k)$ を変更することで、知覚的な音声品質を向上したり、知覚的な音楽トーンを低減することができる。

例えば、自動車環境における大きく、ローパスのノイズを効果的に処理するには、低周波係数（例 : 60 Hz 以下）がゼロに設定される。

11. **フィルタ出力の計算。**フィルタ出力は、次のように定義される。

$$\hat{\mathbf{Y}}_n(k) = \mathbf{H}_n^y(k) \cdot \mathbf{Z}_n(k)$$

12. **逆方向 DFT と Overlap-Add**。時間領域フィルタ出力は、次のように定義される。

$$\hat{\mathbf{y}}_{n-1} = \begin{bmatrix} \mathbf{0}_{\frac{N}{2} \times \frac{N}{2}} & \mathbf{I}_{\frac{N}{2} \times \frac{N}{2}} \end{bmatrix} \cdot \sqrt{\mathbf{w}} \bullet \mathbf{F}^{-1} \hat{\mathbf{Y}}_{n-1} + \begin{bmatrix} \mathbf{I}_{\frac{N}{2} \times \frac{N}{2}} & \mathbf{0}_{\frac{N}{2} \times \frac{N}{2}} \end{bmatrix} \cdot \sqrt{\mathbf{w}} \bullet \mathbf{F}^{-1} \hat{\mathbf{Y}}_n$$

このアルゴリズムは、出力で N/2 の遅延が発生する。

データ構造体

MCRA ノイズ・フロア推定量に関連付けられた構造体は `IppsMCRAParam` である。この構造体は内部で使用され、プログラマによって変更することはできない。

フィルタ更新プリミティブ

FilterUpdateEMNS

ノイズ・サプレッション・フィルタの
係数を計算する。

```
IppStatus ippsFilterUpdateEMNS_32s(const Ipp32s *pSrcWienerCoefsQ31,
    const Ipp32s *pSrcPostSNRQ15, Ipp32s *pDstFilterCoefsQ31, int len);
```

引数

pSrcWienerCoefsQ31	Q31 で表現される Wiener（ウィーナ）フィルタ係数が格納された実数値ベクトルへのポインタ（0.0 $_{Q31}$ ≤ *pSrcWienerCoefsQ31*[k] < 1.0 $_{Q31}$）。
pSrcPostSNRQ15	Q15 で表現される事後 SNR の推定が格納された実数値ベクトルへのポインタ（0 $_{Q15}$ < *pSrcPostSNRQ15* < 32768.0 $_{Q15}$）。
len	入力および出力ベクトルに格納された要素の数（0 < *len* < 65536）。
pDstFilterCoefsQ31	Q31 で表現されるフィルタ係数が格納された実数値ベクトルへのポインタ（0.0 $_{Q31}$ ≤ *pDstFilterCoefsQ31*[k] < 1.0 $_{Q31}$）。

説明

関数 ippsFilterUpdateEMNS は、ippsr.h ファイルで宣言される。この関数は、ノイズ・サプレッションのフィルタ係数を計算する。通常、これらのフィルタのサイズは 65、129、および 257 である（対応する FFT のサイズは 128、256、512）。サンプル・レートはそれぞれ Fs ≤ 11025 Hz、11025 Hz < Fs ≤ 22050 Hz、および 22050 Hz < Fs ≤ 44100 Hz を推奨する。ノイズ・サプレッションのフィルタ係数は、各 FFT ビンに適用されるゲインとなる（ゼロから 1 までのスカラ値）。これらのゲインは、次の式の解で表示される。

$$\min E\left\{\left(\left|\mathbf{Y}_n(k)\right| - \left|\hat{\mathbf{Y}}_n(k)\right|\right)^2\right\}$$

ここで、$\mathbf{Y}_n(k)$ は n 番目の音声サンプルのブロックに対応する k 番目の DFT コンポーネントであり、$\hat{\mathbf{Y}}_n(k)$ は $\mathbf{Y}_n(k)$ の推定である。音声のスペクトル・コンポーネントは、ガウス確率密度を持つものと見なされる。音声のスペクトル・コンポーネント

は直接監視することができないため、ソリューションはノイズ観測 $\mathbf{Z}_n(k)$ によって記述される。ノイズは、加法的なガウスとして見なされる。最小平均 2 乗誤差の大きさ推定は、

$$\hat{\mathbf{Y}}_n(k) = \mathbf{H}_n(k) \cdot \mathbf{Z}_n(k)$$

である。ここで、

$$\mathbf{H}_n(k) = \frac{\sqrt{\pi}}{2} \cdot \frac{\sqrt{\mathbf{W}_n^y(k) \cdot \Re_n^{post}(k)}}{\Re_n^{post}(k)} \cdot M\left(\mathbf{W}_n^y(k) \cdot \Re_n^{post}(k)\right)$$

$$M(\theta) = e^{-\frac{\theta}{2}}\left[(1+\theta) \cdot I_0\left(\frac{\theta}{2}\right) + \theta \cdot I_1\left(\frac{\theta}{2}\right)\right]$$

$$\mathbf{W}_n^y(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^y(k) + \mathbf{P}_{n-1}^v(k)}$$

$$\Re_n^{post}(k) = \frac{\left|\mathbf{Z}_n(k)\right|^2}{\mathbf{P}_{n-1}^v(k)}$$

I_0 と I_1 は、Bessel（ベッセル）関数である。$\mathbf{P}_n^v(k)$ は、ノイズのパワー・スペクトルにおける k 番目のコンポーネントの推定値である。

$\mathbf{P}_n^y(k)$ は、音声のパワー・スペクトルにおける k 番目のコンポーネントの推定値である。

関数 `ippsFilterUpdate_EMNS` は、上記の式に従ってフィルタ係数 $\mathbf{H}_n(k)$ を計算する。パワー・スペクトルの推定値は、通常 `ippsAddWeighted` プリミティブを使用して計算される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	ポインタ *pSrcWienerCoefsQ31*、*pSrcPostSNRQ15*、または *pDstFilterCoefsQ31* が NULL。
`ippStsLengthErr`	エラー。*len* の値が無効。

FilterUpdateWiener

Wiener（ウィーナ）フィルタの
係数を計算する。

```
IppStatus ippsFilterUpdateWiener_32s(const Ipp32s *pSrcPriorSNRQ15,Ipp32s
     *pDstFilterCoefsQ31, int len);
```

引数

pSrcPriorSNRQ15	Q15 で表現される事前 SNR の推定が格納された実数値ベクトルへのポインタ（0.0_{Q15} < *pSrcPriorSNRQ15[k]* < 32768.0_{Q15})。
pDstFilterCoefsQ31	Q31 で表現されるフィルタ係数が格納された実数値ベクトルへのポインタ(0.0_{Q31} ≤ *pDstFilterCoefsQ31*[k] < 1.0_{Q31})。
len	入力および出力ベクトルに格納された要素の数（0 < *len* < 65536)。

説明

関数 `ippsFilterUpdateWiener` は、`ippsr.h` ファイルで宣言される。この関数は、Wiener（ウィーナ）フィルタ係数を計算する。通常、これらのフィルタのサイズは 65、129、および 257 である（対応する FFT のサイズは 128、256、512）。サンプル・レートは、それぞれ Fs ≤ 11025 Hz、11025 Hz < Fs ≤ 22050 Hz、および 22050 Hz < Fs ≤ 44100 Hz を推奨する。Wiener（ウィーナ）フィルタ係数は、各 FFT ビンに適用されるゲインとなる（ゼロから 1 までのスカラ値)。これらのゲインは、次の式の解で表示される。

$$\min E\left\{\left|\mathbf{Y}_n(k)-\hat{\mathbf{Y}}_n(k)\right|^2\right\}$$

ここで、$\mathbf{Y}_n(k)$ は n 番目の音声サンプルのブロックに対応する k 番目の DFT コンポーネントであり、$\hat{\mathbf{Y}}_n(k)$ は $\mathbf{Y}_n(k)$ の推定である。音声のスペクトル・コンポーネントは、

ガウス確率密度を持つものと見なされる。音声のスペクトル・コンポーネントは、

直接監視することができないため、ソリューションはノイズ観測 $\mathbf{Z}_n(k)$ によって記述される。

ノイズは、加法的なガウスとして見なされる。最小平均 2 乗誤差の大きさ推定は、

$$\hat{\mathbf{Y}}_n(k) = \mathbf{W}_n^y(k) \cdot \mathbf{Z}_n(k)$$

である。ここで、

$$\mathbf{W}_n^y(k) = \frac{1}{1 + 1/\Re_n^{prio}(k)}$$

$$\Re_n^{prio}(k) = \frac{\mathbf{P}_n^y(k)}{\mathbf{P}_n^v(k)}$$

このプリミティブは、近似値を $\mathbf{W}_n^y(k)$ に使用して実行時間を短縮する。この近似値は多くのノイズ・リダクション・アプリケーションで十分な働きをする。ただし、高い精度を必要とする場合は、`ippsFilterUpdateWiener_32s` の代わりに `ippsDiv_32s_Sfs` にパラメータを指定して呼び出す。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcPriorSNRQ15* または *pDstFilterCoefsQ31* が NULL。
`ippStsLengthErr`	エラー。*len* の値が無効。

ノイズ・フロア推定プリミティブ

GetSizeMCRA

`IppMCRAState` ステート構造体で必要なサイズをバイト単位で計算する。

```
IppStatus ippsGetSizeMCRA_32s(int nFFTSize, int *pDstSize);
```

引数

nFFTSize	ノイズ PSD 推定で使用する FFT のサイズ（8 ≤ *nFFTSize* ≤ 8192）。

pDstSize	バイト単位のサイズを格納する変数へのポインタ。

説明

関数 ippsGetSizeMCRA は、ippsr.h ファイルで宣言される。この関数は、ippsUpdateNoisePSDMCRA 関数で必要な IppMCRAState ステート構造体用に割り当てるメモリのサイズ（バイト単位）を計算する。関数 ippsGetSizeMCRA は、ippsInitMCRA の呼び出し前、およびメモリの割り当て前に呼び出す必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDstSize* が NULL。
ippStsSizeErr	エラー。*nFFTSize* の値が不正。

InitMCRA

IppMCRAState ステート構造体を初期化する。

```
IppStatus ippsInitMCRA_32s_I(int nSamplesPerSec, int nFFTSize,
    IppMCRAState *pDst);
```

引数

nSamplesPerSec	入力サンプル・レート（8000 ≤ *nSamplesPerSec* ≤ 48000）。
nFFTSize	ノイズ PSD 推定で使用する FFT のサイズ（8 ≤ *nFFTSize* ≤ 8192 ）。
pDst	ステート構造体へのポインタ。

説明

関数 ippsInitMCRA は、ippsr.h ファイルで宣言される。この関数は、ippsUpdateNoisePSDMCRA 関数のステート構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* が NULL。

ippStsSizeErr	エラー。*nFFTSize* の値が不正。
ippStsRangeErr	*nSamplesPerSec* が範囲外。

注： *pDst* に格納されたステート・メモリ・アドレスは、32 ビットのワード境界にアライメントする必要がある。

InitAllocMCRA

メモリを割り当て、IppMCRAState ステート構造体を初期化する。

```
IppStatus ippsInitAllocMCRA_32s_I(int nSamplesPerSec, int nFFTSize,
    IppMCRAState **ppDst);
```

引数

nSamplesPerSec	入力サンプル・レート（0 < *nSamplesPerSec* ≤ 48000）。
nFFTSize	ノイズ PSD 推定で使用する FFT のサイズ (8 ≤ *nFFTSize* ≤ 8192)。
ppDst	ステート構造体へのポインタのポインタ。

説明

関数 ippsInitAllocMCRA は、ippsr.h ファイルで宣言される。この関数は、メモリを割り当て、ippsUpdateNoisePSDMCRA 関数のステート構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDst* ポインタが NULL。
ippStsSizeErr	エラー。*nFFTSize* の値が不正。
ippStsRangeErr	*nSamplesPerSec* が範囲外。

UpdateNoisePSDMCRA

ノイズのパワー・スペクトルを再推定する。

```
IppStatus ippsUpdateNoisePSDMCRA_32s_I(const Ipp32s *pSrcNoisySpeech,
    IppMCRAState *pSrcDstState, Ipp32s *pSrcDstNoisePSD);
```

引数

pSrcNoisySpeech	ノイズ音声の FFT の大きさの 2 乗が格納された実数値ベクトルへのポインタ（$0 \le pSrcNoisySpeech[k] < 2^{31}$）。
pSrcDstState	ステート構造体へのポインタ。
pSrcDstNoisePSD	ノイズのパワー・スペクトル・ベクトルへのポインタ。

説明

関数 ippsUpdateNoisePSDMCRA は、ippsr.h ファイルで宣言される。この関数は、ノイズ音声の大きさの 2 乗の新しい値に基づいてノイズのパワー・スペクトルを再推定する。このアルゴリズムは、[Coh02] で説明されている MCRA（Minima Controlled Recursive Averaging）アプローチに基づいている。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcNoisySpeech*、*pSrcDstState*、または *pSrcDstNoisePSD* が NULL。

音響エコー・キャンセラ

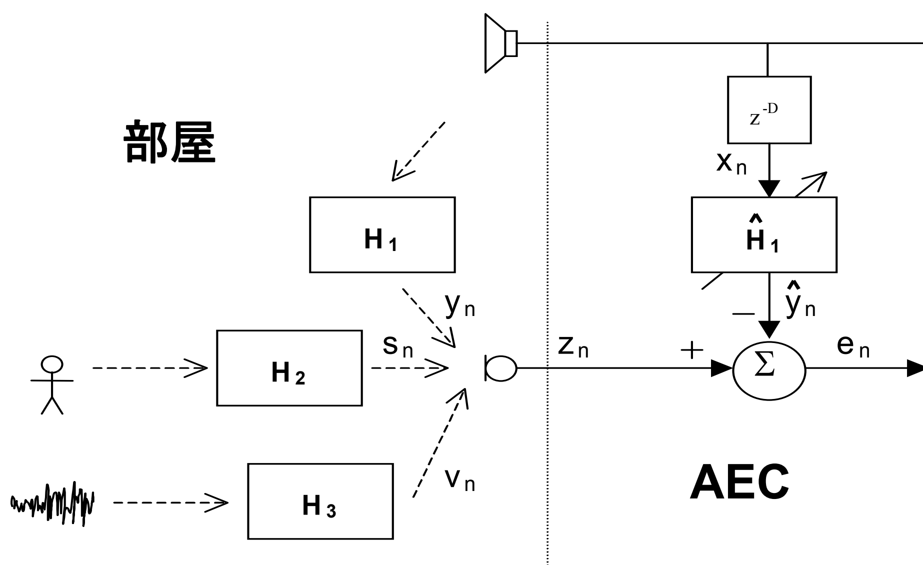
この項では、音響エコー・キャンセラ（AEC）を構築するインテル® IPP 関数について説明する。プリミティブは主に、計算に時間のかかる演算を行う。
AEC は、周波数領域適応フィルタ・アルゴリズムとコントローラで構成される。
IPP AEC プリミティブは、次の機能をサポートする。

- フィルタリング
- フィルタ係数
- ステップ・サイズの更新
- AEC コントローラ

音響エコー・キャンセラ・アーキテクチャ

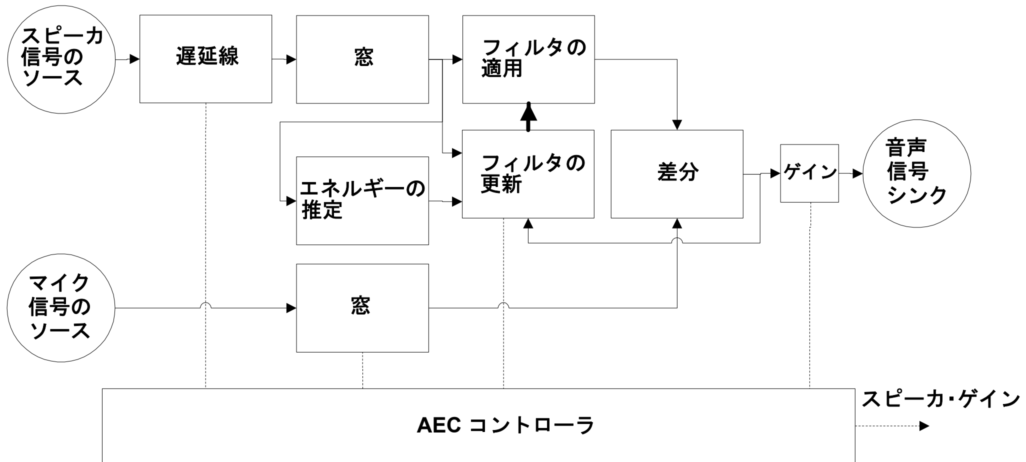
図 8-8 は、音響エコー・キャンセル処理を行う一般的なアプリケーション（モノラル・スピーカフォン）を示す。ここでは、部屋のスピーカから再生するオーディオをマイクロフォンでキャプチャする。**H_1** はスピーカとマイクロフォン間の音響伝達関数を示す。y_n（n は時間インデックス）はマイクロフォンでの結果信号を示す。同時に、人間の会話もマイクロフォンにキャプチャされる。**H_2** は人間とマイクロフォン間の音響伝達関数を示し、s_n はマイクロフォンでの結果信号を示す。また、ノイズのソース（例えばコンピュータの冷却ファンの音）もキャプチャされる。**H_3** はノイズのソースとマイクロフォン間の音響伝達関数を示し、v_n はマイクロフォンでの結果信号を示す。信号は、マイクロフォンで加法的に混合される。AEC は、伝達関数 **H_1** と遅延 D を正確に推定することで、フィルタリングされたスピーカの信号をマイクロフォンの信号から減算して、エコーをキャンセルする。

図 8-8　一般的な音響エコー・キャンセルの例



音響エコー・キャンセラの基本要素を示す 図 8-9 では、音声信号ストリームに AEC を適用するステップについて説明する。

図 8-9　音響エコー・キャンセル・システムの基本要素



周波数領域ブロック NLMS 適応フィルタ

アルゴリズムのステップ

周波数領域の正規化された最小平均 2 乗（NLMS）適応フィルタは、簡単な計算の構造体と適度の複雑さを備えている [Ash94]。アルゴリズムは、次のステップで構成される。

1. **新規入力ブロック**。最後に取得した N/2 入力サンプル χ_n と以前に取得した N/2 入力サンプル χ_{n-1} は、新しい入力ブロック $\mathbf{x}_n$ を構成する。

$$\mathbf{x}_n = \begin{bmatrix} \chi_{n-1} \\ \chi_n \end{bmatrix}$$

2. **入力 DFT**。入力の離散フーリエ変換が計算される。

$$\mathbf{X}_n = \mathbf{F}\mathbf{x}_n$$

3. **入力ヒストリの更新**。最後に取得した L 周波数領域入力ブロックは保持される。

$$\mathrm{X}_n = [\mathbf{X}_n \mathbf{X}_{n-1} \Lambda\ \mathbf{X}_{n-L+1}]$$

4. **入力エネルギー推定の更新。**次の式で計算した平均値を使用して、各ビン $P_{xx}(k)$ におけるエネルギーを推定できる（ $0 < \beta < 1$ の場合）。

$$\hat{P}_{xx}(k) = (1-\beta)\cdot\hat{P}_{xx}(k) + \beta\cdot\left|\mathbf{X}_n(k)\right|^2 \text{ for } k = 0,\ldots,N/2$$

5. **適応ステップのサイズの計算**。適応ステップのサイズは、正規化 LMS（NLMS）アプローチを使用して計算される（ $0 < \mu << 1$ および $P_{x\min} > 0$ の場合)。

$$\mathrm{M}(k) = \begin{cases} \dfrac{\mu}{\hat{P}_{xx}(k)}, & \hat{P}_{xx}(k) > P_{x\min} \\ \dfrac{\mu}{P_{x\min}}, & \hat{P}_{xx}(k) \le P_{x\min} \end{cases} \quad \text{for } k = 0,\ldots,N/2$$

6. **フィルタ出力の計算。**適応フィルタは、[Ash94] で説明されている低レイテンシ構造体を使用する。ここでは、インパルス応答フィルタは重なり合わないセグメントとして均等に分割される。一般に、これはたたみ込み

$$y(m) = \sum_{i=0}^{I-1} h(i)\cdot x(m-i)$$

を次のたたみ込みに書き換えることに相当する。

$$y(m) = \sum_{j=0}^{J-1}\sum_{k=0}^{I/J-1} h(k + j(I/J))\cdot x(m - j(I/J) - k) = \sum_{j=0}^{J-1}\sum_{k=0}^{K-1} h_j(k)\cdot x_j(m-k)$$

周波数領域内のフィルタ出力は、次の式で計算される。

$$\hat{\mathbf{Y}}_n(k) = \sum_{i=0}^{L-1}\mathbf{X}_{n-i}(k)\cdot\mathbf{H}_i(k) \text{ for } k = 0,\ldots,N/2$$

ここで、$\mathbf{H}_i$ は、たたみ込みが線形になるように正しく抑制される（ステップ 9 を参照)。

7. **エラーの抑制**。最初に、時間領域フィルタ出力が逆方向の離散フーリエ変換で計算される。

$$\hat{\mathbf{y}}_n = \mathbf{F}^{-1}\hat{\mathbf{Y}}_n$$

音響エコー・キャンセル・アプリケーションでは、エラー $\mathbf{e}_n$ は

マイクロフォン入力 $\mathbf{y}_n$ と適応フィルタ出力の差である。

$$\mathbf{e}_n = \mathbf{y}_n - \hat{\mathbf{y}}_n$$

intel®

時間領域エラーは、次のように抑制する必要がある。

$$\mathbf{E}_n = \mathbf{F}\begin{bmatrix} \mathbf{0} \\ \boldsymbol{\varepsilon}_n \end{bmatrix}$$

これによって、周波数領域内の適応ステップ（ステップ 8 を参照）を正しく実装できる。

エコー・キャンセラ出力の n^{th} ブロックは、次のように定義される。

$$\boldsymbol{\varepsilon}_n = [\mathbf{e}_n(N/2)\Lambda\ \mathbf{e}_n(N-1)]^T$$

8. **適応フィルタの更新**。適応フィルタの更新は、周波数領域で行われる。適応フィルタ応答のセグメントは次の式で個別に更新される。

$$\mathbf{H}_i = \mathbf{H}_i + \mathbf{M}_n \bullet \mathbf{X}^*_{n-i} \bullet \mathbf{E}_n \ \ \text{for}\ \ i = 0,...,L-1$$

ここで、$\bullet$ は乗算記号を示す。また、$\mathbf{X}^*_{n-i} \bullet \mathbf{E}_n$ は傾きの i^{th} 番目のセグメントとして認識される。

9. **適応フィルタ応答の抑制**。周波数領域内で有限線形（フィルタリング）操作を正確に実装するには、抑制を適用する必要がある。フィルタ応答は次のように抑制される

$$\mathbf{h}_i = \mathbf{F}^{-1}\mathbf{H}_i$$

$$\mathbf{h}_i = \mathbf{F}^{-1}\mathbf{H}_i$$

（ $i = 0,...,L-1$ の場合）。ここで、インパルス応答

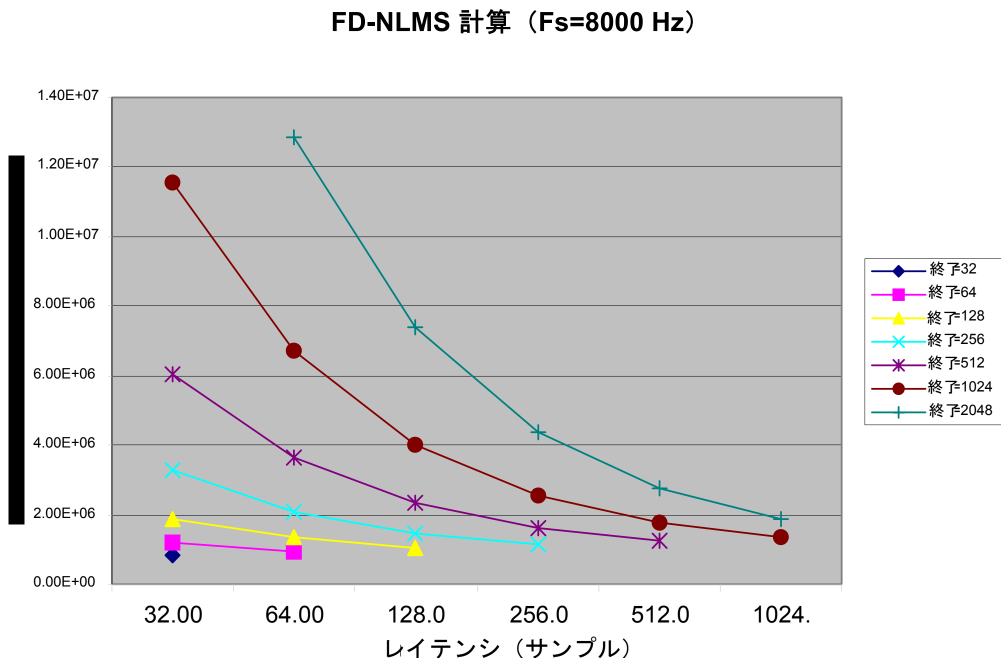
$$\boldsymbol{\eta}_i = [\mathbf{h}_i(0)\Lambda\ \mathbf{h}_i(N/2-1)]^T$$

の各セグメントの前半は保持され、残りはゼロに設定される。

計算の複雑さ

適応フィルタの計算の複雑さを図 8-10 に示す。ここでは、低レイテンシとフィルタ終了の長さにおけるトレードオフを示す。例えば、レイテンシを 8 ミリ秒で固定した場合（8000 Hz サンプル・レートで 1 ブロックにつき 64 サンプル）、128 ミリ秒の終了の長さには約 7 MOPS（Million Operations Per Second）が必要となる。レイテンシを 4 ミリ秒にすると、計算は 11.5 MOPS になる。終了の長さを 256 ミリ秒にすると、計算は 13 MOPS になる。

図 8-10　FD-NLMS 適応フィルタの計算の複雑さ



AEC コントローラ

AEC コントローラの必要性については、図 8-8 を参照する。

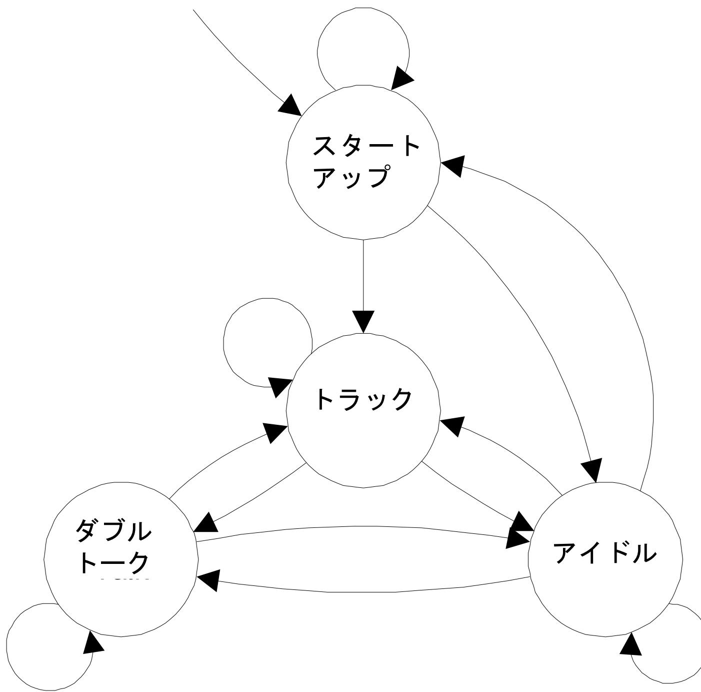
適応フィルタ $\hat{H}_1$ は、フィルタリングされたスピーカ信号 $\hat{y}_n$ が最小 2 乗でマクロフォン信号 z_n に最も一致するよう係数を変更する。

スピーカの信号がアクティブで、s_n と v_n が弱く、なおかつスピーカからマイクへの伝達関数 H_1 が線形に近い場合、$\hat{H}_1$ は正確な伝達関数 H_1 に近づく。スピーカがアクティブで、s_n または v_n もアクティブ（ダブルトーク状態）な場合、$\hat{H}_1$ は 正しいソリューションを収束しない。s_n または v_n が弱くても、スピーカの信号が弱いか無声の場合は、適応フィルタが発散する可能性がある。AEC コントローラは、適応フィルタのステップ・サイズ（ステップ 5 の μ）を調整することでこれらの問題を解決する。これにより、スピーカの信号がマイクロフォンでキャプチャされるとフィルタは瞬時に収束するが、ダブルトーク状態やマイクロフォンがキャプチャした不十分な振動には素早く発散しない。また、適応フィルタが発散すると、コントローラは再生と出力ゲインを管理してエコーをブロックする。

アルゴリズムの説明

AEC コントローラ更新プリミティブ ippsControllerUpdateAEC は、図 8-11 で示す簡単なステート図に基づいている。

図 8-11　AEC コントローラのステート図



ステート遷移は、簡単な完全バンド・エネルギー測定によって制御される。

$$E_n^e = \sum_{k=0}^{N/2-1} e_n^2(k)$$

エラー・エネルギー

$$E_n^z = \sum_{k=0}^{N/2-1} z_n^2(k)$$

マイクロフォン・エネルギー

$$E_n^x = \sum_{k=0}^{N/2-1} x_n^2(k)$$

スピーカ・エネルギー

$\overline{E}_n^z = (1-\gamma)\cdot \overline{E}_{n-1}^z + \gamma \cdot E_n^z$　　平滑化マイクロフォン・エネルギー

$\overline{E}_n^e = (1-\gamma)\cdot \overline{E}_{n-1}^e + \gamma \cdot E_n^e$　　平滑化エラー・エネルギー

$ERLE_n = \overline{E}_n^z / \overline{E}_n^e$　　エコー・リターン・ロス拡張（ERLE）

ブロックの更新レートが 8 ミリ秒の場合、平滑化した定数は $\gamma = 0.005$ となり（例 : 8000 Hz サンプル・レートで 64 サンプル）、他のブロックの更新レートでも同じレートを取得できるように調節される。エネルギー測定を使用して 4 つの条件が定義される。

収束（C）　$ERLE_n > T^{ERLE}$

レシーブ・アクティブ（R）　$E_n^x > T_n^x$

マイク・アクティブ（M）　$E_n^z > T_n^z$

ダブルトークなし（N）　$E_n^x \cdot G_{xz} < E_n^z$

しきい値は次のように設定される。ERLE しきい値 T^{ERLE} は 2.0（3 dB）。

スピーカ・アクティビティのしきい値 T_n^x は

$$6.0 \cdot \min\{E_n^x, E_{n-1}^x, \ldots, E_{n-windowsize}^x\}$$

（ローカル最小値より 8 dB 上）。マイクロフォン・アクティビティのしきい値 T_n^x は

$6.0 \cdot \min\{E_n^z, E_{n-1}^z, \ldots, E_{n-windowsize}^z\}$（ローカル最小値より 8 dB 上）。

スピーカとマイクロフォン間の外部ゲイン G_{xz} は 0.25（-6 dB）となる。

最小トラッキング窓サイズは 2 秒となる。

ステート遷移は、次のテーブルに従って 4 つの条件でトリガされる。

ステップ・サイズ、スピーカ・ゲイン、および出力ゲインは、次のように調整される。

C	R	M	N	現在のステート	次のステート	μ	α	β
F	T	T	T	スタートアップ	スタートアップ	4_c	0.0	1.0
F	T	F	X	スタートアップ	スタートアップ	$_c$	0.0	1.0
F	T	T	F	スタートアップ	スタートアップ	0.0	1.0	0.0
X	F	T	X	スタートアップ	スタートアップ	0.0	1.0	0.0
X	F	F	X	スタートアップ	スタートアップ	0.0	0.5	0.5
T	T	T	T	スタートアップ	トラック	4_c	1.0	1.0
T	T	T	F	スタートアップ	トラック	$_c$	1.0	1.0
T	T	F	T	スタートアップ	トラック	$_c$	0.0	1.0
T	T	F	F	スタートアップ	アイドル	0.0	0.5	0.5
T	T	T	X	トラック	トラック	2_c	1.0	1.0
X	T	F	X	トラック	トラック	$_c$	1.0	1.0
F	T	T	X	トラック	ダブルトーク	0.0	1.0	1.0
X	F	X	X	トラック	アイドル	0.0	0.5	0.5
X	F	T	X	アイドル	アイドル	0.0	1.0	1.0
X	F	F	X	アイドル	アイドル	0.0	0.5	0.5
T	T	T	T	アイドル	トラック	2_c	1.0	1.0
T	T	T	F	アイドル	トラック	2_c	1.0	1.0
X	T	F	X	アイドル	トラック	2_c	1.0	1.0
X	T	T	F	アイドル	ダブルトーク *	0.0	1.0	1.0
F	T	T	X	ダブルトーク	ダブルトーク	0.0	1.0	1.0
T	T	T	T	ダブルトーク	トラック	2_c	1.0	1.0
T	T	T	F	ダブルトーク	トラック	$_c$	1.0	1.0
X	T	F	X	ダブルトーク	トラック	$_c$	1.0	1.0
T	T	T	F	ダブルトーク	トラック *	0.0	1.0	1.0

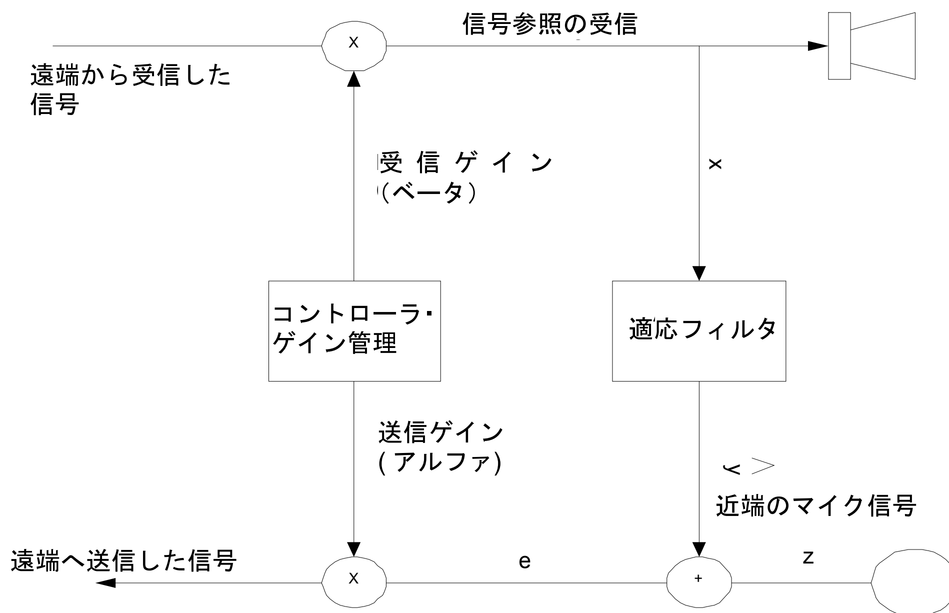
X	F	T	X	ダブルトーク	アイドル	0.0	1.0	1.0
X	F	F	X	ダブルトーク	アイドル	0.0	0.5	0.5

「*」でマークされた遷移は冗長であるが、次に示すように ERLE が「収束（C)」条件のしきい値を超えおり、dB 内のピーク ERLE が 40% 以下の場合は優先される。

$$\log\left(T^{ERLE}\right) < \log\left(ERLE_n\right) < 0.4 \cdot \log\left(\max\left\{ERLE_n, ERLE_{n-1}, \ldots, ERLE_{n-windowsize}\right\}\right)$$

テーブルの値 α と β は、それぞれ送信ゲイン・ターゲットと受信ゲイン・ターゲットに対応する。瞬時ゲインは、聞き取れるフラッタを防ぐために徐々に変更される。ゲイン・ターゲットの 1 つが変更すると、100 ミリ秒後にターゲットと同じになるように瞬時ゲインは直線的に増加または減少される。図 8-12 は、AEC システムの送信ゲインおよび受信ゲインを示す。

図 8-12　AEC コントローラによる AEC システムの送信ゲインおよび受信ゲイン



データ構造体

AEC コントローラは、多くのパラメータにアクセスする必要がある。これらのパラメータは、次の構造体に格納される。

```
typedef struct {
Ipp16s *pMicrophone;/* pointer to mic samples */
Ipp16s *pLoudspeaker;/* pointer to speaker samples */
Ipp16s *pError;        /* pointer to error samples */
Ipp32s *pAFInputPSD;/* pointer to filter input PSD */
Ipp32sc **ppAFCoefs;/* pointer to filter segment array */
Ipp32s muQ31;          /* fixed step size (Q31 value in [0,1)) */
Ipp32s AECOutGainQ30;/* AEC output gain (Q30 value in [0,1]) */
Ipp32s speakerGainQ30;/* loudspeaker gain (Q30 value in [0,1]) */
int numSegments;       /* number of segments of filter tail */
int numFFTBins;        /* number of FFT bins (FFTSize / 2 + 1) */
int numSamples;        /* mic, error, loudspeaker frame size */
int sampleRate;        /* sample rate (Hertz) */

} IppAECNLMSParam;
```

numSamples は、フレームで重なり合わない部分のサイズを示す。IppAECNLMSParam 構造体の要素で有効な範囲を次のテーブルに示す。

メンバ	範囲
pMicrophone	サンプルの範囲は $[-2^{15}, +2^{15})$。
pLoudspeaker	サンプルの範囲は $[-2^{15}, +2^{15})$。
pError	サンプルの範囲は $[-2^{15}, +2^{15})$。
pAFInputPSD	PSD 係数の範囲は $[0, 2^{31})$。
ppAFCoefs	係数の実数部と虚数部の範囲は $[-2^{31}, +2^{31})$。
muQ31	Q31 スカラの範囲は [0 , 1)。
AECOutGainQ30	Q30 スカラの範囲は [0 , 1]。
speakerGainQ30	Q30 スカラの範囲は [0 , 1]。
numSegments	範囲は [1 , 255]。
numFFTBins	範囲は {17, 33, 65, 129, 257, 513, 1025,2049, 4097} の 1 つ。
numSamples	範囲は [32 , 4096]。
sampleRate	範囲は [8000 , 48000]。

フィルタの終了の長さは最大 2 秒までサポートされている。構造体に含まれているポインタは、常に一番新しいブロックまたは係数セットを指している必要がある。コントローラは、`IppAECCtrlState` 構造体の内部状態を保持する。

`IppAECCtrlState` 構造体は内部で使用され、プログラマによって変更することはできない。

`IppAECScaled32s` 構造体は、スケーリングされた 32 ビット符号付整数を提供する。

```
typedef struct {
    Ipp32s val;
    Ipp32s sf;
} IppAECScaled32s;
```

`IppAECScaled32s` 型の変数 *x* は、`x.val` * $2^{x.sf}$ の値を示す。`IppAECScaled32s` 構造体は、格納サイズを大きくすることで、インテル® PCA プロセッサ・ファミリ上で効率的な浮動小数点演算を提供する。通常、`val` フィールドは「左寄せ」である。つまり、左にシフトすることで

$2^{30} \le x.val < 2^{31}$ または $-2^{30} < x.val \le -2^{31}$ となる。

フィルタ・プリミティブ

FilterAECNLMS

周波数領域内の適応フィルタ出力を計算する。

```
IppStatus ippsFilterAECNLMS_32sc_Sfs(const Ipp32sc **ppSrcSignalIn,
    const Ipp32sc **ppSrcCoefs, Ipp32sc *pDstSignalOut, int
    numSegments, int len, int scaleFactor);
```

引数

ppSrcSignalIn	一番新しい入力ブロック（例 : $X_n, X_{n-1}, \ldots, X_{n-L+1}$）へのポインタ配列へのポインタ。これらの複素数値ベクトルは、入力信号の FFT を格納する。引数 *ppSrcSignalIn* は、サイズが [*numSegments*][*len*] の 2 次元の複素数ベクトルである。

ppSrcCoefs	フィルタ係数ベクトルへのポインタ配列へのポインタ。これらの複素数値ベクトルは、フィルタ係数を格納する。引数 *ppSrcCoefs* は、サイズが [*numSegments*][*len*] の 2 次元の複素数ベクトルである。
pDstSignalOut	複素数値フィルタ出力ベクトルへのポインタ。
numSegments	フィルタ・セグメントの数(L)(0 < *numSegments* < 256)。
len	各フィルタ・セグメントの入力および出力ベクトルに格納された要素の数(0 < *len* ≤ 4097)。
scaleFactor	飽和固定スケール係数(-32 < *scaleFactor* < 32)。

説明

関数 ippsFilterAECNLMS は、ippsr.h ファイルで宣言される。この関数は、上記のアルゴリズムのステップ 6 を実行して、周波数領域の適応フィルタ出力を計算する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *ppSrcSignalIn*、*ppSrcCoefs*、または *pDstSignalOut* が NULL。
ippStsLengthErr	エラー。*len* の値が無効。
ippStsRangeErr	エラー。*numSegments* または *scaleFactor* が範囲外。

フィルタ更新プリミティブ

CoefUpdateAECNLMS

適応フィルタの係数を更新する。

```
IppStatus ippsCoefUpdateAECNLMS_32sc_I(const
    IppAECScaled32s*pSrcStepSize, const Ipp32sc **ppSrcFilterInput,
    const Ipp32sc *pSrcError, Ipp32sc **ppSrcDstCoefs, int numSegments,
    int len, int scaleFactorCoef);
```

引数

pSrcStepSize
: スケーリングされた整数ステップ・サイズのベクトルへのポインタ。val フィールドを左寄せにすることを推奨する。つまり、左にシフトすることで 230 ≤ *pSrcStepSize*[k].val < 231 および sf フィールドが、範囲（-64 ≤ *pSrcStepSize*[k].sf < 64）に制限される。*pSrcStepSize* の次元は [*len*] となる。

ppSrcFilterInput
: 一番新しい入力ブロック（例 : $X_n, X_{n-1}, \ldots, X_{n-L+1}$）へのポインタ配列へのポインタ。これらの複素数値ベクトルは、入力信号の FFT を格納する。
ppSrcFilterInput の次元は [*numSegments*][*len*] である。

pSrcError
: フィルタ・エラーを格納する複素数値ベクトルへのポインタ。*pSrcError* の次元は [*len*] である。

ppSrcDstCoefs
: フィルタ係数ベクトルへのポインタの配列へのポインタ。これらの複素数値ベクトルは、フィルタ係数を格納する。
ppSrcDstCoefs の次元は [*numSegments*][*len*] である。

numSegments
: フィルタ・セグメントの数 (L) (0 < *numSegments* < 256)。

len
: 各入力および出力ベクトルに格納された要素の数（0 < *len* ≤ 4097）。

scaleFactorCoef
: フィルタ係数の固定スケール係数 (0 ≤ *scaleFactor* < 32)。通常、ippsCoefUpdateAECNLMS および ippsFilterAECNLMS 関数では同じスケール係数が使用される。
フィルタ係数は、この関数に呼び出される前に $2^{scaleFactorCoef}$ で掛けられていると見なされる。
オーバーフローが発生すると、更新の結果は飽和される。

説明

関数 ippsCoefUpdateAECNLMS は、ippsr.h ファイルで宣言される。この関数は、上記のアルゴリズムのステップ 8 を実行して、適応フィルタ係数を更新する。スケール係数は、次の部分にのみ適用される。

「$+ \mathbf{M}_n \bullet \mathbf{X}^*_{n-i} \bullet \mathbf{E}_n$ 」

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *ppSrcStepSize*、*ppSrcFilterInput*、*pSrcError*、または *ppSrcDstCoefs* が NULL。
ippStsLengthErr	エラー。*len* の値が無効。
ippStsRangeErr	エラー。*pSrcStepSize*[i].val<0 か、または、*numSegments* か *scaleFactorCoef* が範囲外。

ステップ・サイズ更新プリミティブ

StepSizeUpdateAECNLMS

適応ステップのサイズを計算する。

```
IppStatus ippsStepSizeUpdateAECNLMS_32s(const Ipp32s *pSrcInputPSD,
    Ipp32s muQ31, IppAECScaled32s maxStepSize, Ipp32s minInputPSD,
    IppAECScaled32s *pDstStepSize, int len);
```

引数

pSrcInputPSD	入力パワー・スペクトル推定が格納された実数値ベクトルへのポインタ。
muQ31	Q31 で表現されるスカラ実数値 ($0.0_{Q31} \le muQ31 < 1.0_{Q31}$)。
maxStepSize	左寄せとスケーリングが行われた整数値。利用可能な最大ステップ・サイズを示す（通常、この値は $mu / minInputPSD > 0$ である）。
minInputPSD	ステップ・サイズの更新を行う入力 PSD の最小値 ($0 < minInputPSD$)。
len	入力および出力ベクトルに格納された要素の数 ($0 < len \le 4097$)。
pDstStepSize	左寄せとスケーリングが行われた整数出力ベクトルへのポインタ。

説明

関数 ippsStepSizeUpdateAECNLMS は、ippsr.h ファイルで宣言される。この関数は、上記のアルゴリズムのステップ 5 を実行して、適応ステップ・サイズを計算する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcInputPSD* または *pDstStepSize* が NULL。
ippStsLengthErr	エラー。*len* の値が無効。
ippStsRangeErr	エラー。*pSrcInputPSD*[i]< 0、*smuQ31*< 0、*minInputPSD*< 0、または *maxStepSize*.val< 0。

AEC コントローラ・プリミティブ

ControllerGetSizeAEC

AEC コントローラ・ステート構造体のサイズを返す。

```
IppStatus ippsControllerGetSizeAEC_32s(int *pDstSize);
```

引数

pDstSize	ステートのサイズを格納する変数へのポインタ (バイト単位)。

説明

関数 ippsControllerGetSizeAEC は、ippsr.h ファイルで宣言される。この関数は、AEC コントローラ・ステート構造体のサイズをバイト単位で返す。この関数を使用することで、正しいサイズのメモリを割り当てることができる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDstSize* が NULL。

ControllerInitAEC

AEC コントローラ・ステート構造体を
初期化する。

```
IppStatus ippsControllerInitAEC_32s(const IppAECNLMSParam *pSrcParams,
    IppAECCtrlState *pDstState);
```

引数

pSrcParams	AEC パラメータ構造体へのポインタ。
pDstState	AEC コントローラ・ステート構造体へのポインタ。

説明

関数 ippsControllerInitAEC は、ippsr.h ファイルで宣言される。この関数は、AEC コントローラ・ステート構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcParams* または *pSrcDstState* が NULL。
ippStsRangeErr	エラー。*pSrcParams->numSamples* または *pSrcParams->sampleRate* が範囲外（範囲に関する詳細はデータ構造体を参照のこと）。

注： *pDstState* に格納されたステート・メモリ・アドレスは、32 ビットのワード境界にアライメントする必要がある。

ControllerUpdateAEC

エネルギー・ベースの AEC コントローラを
実行する。

```
IppStatus ippsControllerUpdateAEC_32s(const IppAECNLMSParam
    *pSrcParams, IppAECCtrlState *pSrcDstState, Ipp32s *pDstMuQ31,
    Ipp32s *pDstAECOutGainQ30, Ipp32s *pDstSpeakerGainQ30);
```

引数

pSrcParams	AEC パラメータ構造体へのポインタ。
pSrcDstState	AEC コントローラ・ステート構造体へのポインタ。
pDstMuQ31	Q31 で表現されるスカラ実数値へのポインタ（$0.0_{Q31} \le pDstMuQ31 < 1.0_{Q31}$）。
pDstAECOutGainQ30	Q30 で表現されるスカラ実数値へのポインタ（$0.0_{Q30} \le pDstAECOutGainQ31 \le 1.0_{Q30}$）。
pDstSpeakerGainQ30	Q30 で表現されるスカラ実数値へのポインタ（$0.0_{Q30} \le pDstSpeakerGainQ31 \le 1.0_{Q30}$）。

説明

関数 `ippsControllerUpdateAEC` は、`ippsr.h` ファイルで宣言される。この関数は、アルゴリズムの説明の項で説明されているエネルギー・ベースの AEC コントローラを実行する。この関数は、さまざまなシステム・パラメータに基づいて、固定の適応ステップ・サイズ（*mu*）、AEC 出力ゲイン、およびスピーカ・ゲインを設定する。例えば、「ダブルトーク」状態（これは、マイクロフォンで話している最中に、スピーカで音声を再生している状態）では、適応ステップ・サイズが小さくなる。

intel®

注 :ippsControllerUpdateAEC_32s プリミティブは、基本的な AEC 制御機能を提供する。しかし、最高のパフォーマンスを得るには、特定のハードウェア専用のコントローラと任意のアプリケーションに適用できる物理デバイスを設計する必要がある。このコントローラは、プロトタイピング段階で基準の AEC パフォーマンスを提供することを目的としている。最終的な製品では、このコントローラをカスタム・コントローラと置き換える。このコントローラには次の制限がある。

- ダブルトーク状態における適応フィルタ収束を 2 秒以上管理しない。このケースは、通常の収束では滅多に発生しない。
- スピーカのエネルギーが数百ミリ秒以上一定である場合は適応しない。このケースは、テストでは滅多に発生しない。ただし、広帯域の定常ノイズをスピーカで再生しても、適応フィルタが瞬時に収束しないという予期せぬ副作用が発生する場合がある (コントローラがないときに発生する場合がある)。
- スピーカとマイクロフォンの間でおよそ 6 dB 低くなることを想定する必要がある。これは、物理的なレイアウト、スピーカとマイクロフォンの方向性、およびアナログ再生ゲインに大きく依存する。したがって、このコントローラを使用する AEC は、音量レベルの大きい外部増幅器を使用すると正常に動作しない場合がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcParams*、*pSrcDstState*、*pDstMuQ31*、*pDstAECOutGainQ30*、または *pDstSpeakerGainQ30* が NULL。

音声アクティビティ検出 (VAD)

この項では、音声アクティビティ検出 (VAD) を構成するインテル® IPP 関数を説明する。プリミティブは主に、計算に時間のかかる演算を行う。

VAD は、測定パラメータと音声モデリング・ヒューリスティックから構成される。

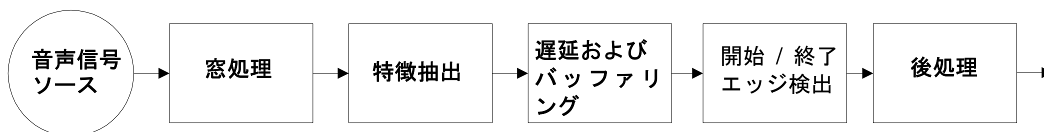
IPP VAD プリミティブは、次の機能をサポートする。

- ピーク・ピッキング
- 周期性
- ゼロ交差レート

音声アクティビティ検出アーキテクチャ

一般的な音声アクティビティ検出を下記の図に示す。音声信号はウィンドウ処理され、フレームに分けられる（重なり合ったフレーム）。次に、特徴抽出が実行される。特徴抽出は、ゼロ交差レート計算、エネルギー計算、セグメント SNR 推定、周期性検出、サンプルまたはサブバンド・ヒストグラム測定のうち 1 つ以上が行われる。測定結果はバッファされるため、音声の開始と終了を他の音声イベントと区別できる。400 ミリ秒の遅延が一般的である。音声の開始と終了エッジ検出は、ヒューリスティック・ルールに基づいて実行される。後処理（例 : メディアン・フィルタリング）は低遅延で実行され、誤った検出を防ぐ。

図 8-13　音声アクティビティ検出の基本要素



音声アクティビティ検出プリミティブ

FindPeaks

入力ベクトルのピークを識別する。

```
IppStatus ippsFindPeaks_32s8u(const Ipp32s *pSrc, Ipp8u *pDstPeaks, int
     len, int searchSize, int movingAvgSize);
```

引数

pSrc	入力ベクトルへのポインタ。

pDstPeaks	入力ベクトルのピークに 1 を、それ以外に 0 を格納する出力ベクトルへのポインタ。
len	入力および出力ベクトルに格納された要素の数（0 < *len* < 65536）。
searchSize	ピークより右または左にある要素の数（0 < *searchSize* < 128）。
movingAvgSize	ピークを選択する前に適用される、移動平均窓に含まれる右または左にある要素の数（0 ≤ *movingAvgSize* < 128）。

説明

関数 ippsFindPeaks は、ippsr.h ファイルで宣言される。この関数は、入力ベクトルのピークを識別し、出力ベクトルでピークの位置に 1 を、それ以外の場所に 0 を格納する。
ピークは、ポイント *pSrc*[i] として定義される。つまり、*searchSize* が L の場合、*pSrc*[i-L] < *pSrc*[i-L+1] < … < *pSrc*[i-1] < *pSrc*[i] > … > *pSrc*[i+L-1] > *pSrc*[i+L] となる。
movingAvgSize が 0 より大きい場合、ソース・ベクトルはピークが選択される前に移動平均によって平滑化される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDstPeaks* が NULL。
ippStsLengthErr	エラー。*len* が範囲外。
ippStsSizeErr	エラー。*SearchSize* または *movingAvgSize* が範囲外。

PeriodicityLSPE

入力音声フレームの周期性を計算する。

```
IppStatus ippsPeriodicityLSPE_16s(const Ipp16s *pSrc, int len, Ipp16s
    *pPeriodicityQ15, int *period, int maxPeriod, int minPeriod);
```

引数

pSrc	入力音声ベクトルへのポインタ。

len	入力ベクトルに格納された要素の数（$6 < len \le \min(16 * minPeriod, 1024)$）。
pPeriodicityQ15	最大周期的サンプリングの正規化された和に対応した、Q15 で表現される値へのポインタ（$0.0_{Q15} \le pPeriodicityQ15 \le 1.0_{Q15}$）。
period	LSPE コスト関数を最小限にする周期（サンプル単位）
maxPeriod	検索する最大周期（$minPeriod < maxPeriod < len$）。
minPeriod	検索する最小周期（$6 \le minPeriod < maxPeriod$）。

説明

関数 `ippsPeriodicityLSPE` は、`ippsr.h` ファイルで宣言される。この関数は、入力音声フレームの周期性を計算する。周期性は、[Tuc92] で定義されている最小 2 乗周期性推定値（LSPE）アルゴリズムに従って計算される。周期性検索は、2000 ～ 24000 Hz 間のサンプル・レートの音声信号を対象に設計されており、音楽またはその他のソースでは正しく動作しない可能性がある。周期性（有声化）のみが必要な場合、このプリミティブを呼び出す前に入力音声をダウンサンプリングすることで効率を改善できる。例えば、2000 Hz のサンプル・レートでは、*minPeriod*=10、*maxPeriod*=20、および *len*=64 で、いくつかのアプリケーションで適切な周期性を提供する。

戻り値

ippStsNoErr	エラーなし。
ippStsLengthErr	エラー。*len* が範囲外。
ippStsRangeErr	エラー。*maxPeriod* または *minPeriod* が範囲外。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pPeriodicityQ15* が NULL。

Periodicity

入力ブロックの周期性を計算する。

```
IppStatus ippsPeriodicity_32s16s(const Ipp32s *pSrc, int len, Ipp16s
    *pPeriodicityQ15, int *period, int maxPeriod, int minPeriod);
```

引数

pSrc	負でないエントリを格納した入力ベクトルへのポインタ。
len	入力ベクトルに格納された要素の数（$0 < len \le 4096$）。
pPeriodicityQ15	最大高調波サンプリングの正規化された和に対応した、Q15 で表現される値へのポインタ（$0.0_{Q15} \le pPeriodicityQ15 \le 1.0_{Q15}$）。
period	最大エネルギー高調波サンプリングを提供する周期（サンプル単位）へのポインタ。
maxPeriod	検索する最大周期（$minPeriod < maxPeriod < len$）。
minPeriod	検索する最小周期（$0 < minPeriod < maxPeriod$）。

説明

関数 `ippsPeriodicity` は、`ippsr.h` ファイルで宣言される。この関数は、入力ブロックの周期性を計算する。通常のアプリケーションでは、入力ブロックはウィンドウ処理された音声の離散フーリエ変換の大きさを 2 乗したものである。周期性は、最大のエネルギーを維持する入力ブロックの周期的なサンプリングとして定義される。

$$\max_{k,T_0} \sum_n x(k + nT_0) \quad \text{where } 0 < k \le T_0$$

バイアス除去は検索の前に実行され、正確な測定が保証される。

戻り値

ippStsNoErr	エラーなし。
ippStsLengthErr	エラー。*len* が範囲外。
ippStsRangeErr	エラー。*pSrc*[k]、*maxPeriod*、または *minPeriod* が範囲外。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*period*、または *pPeriodicityQ15* が NULL。

バージョン 1.1 との互換性

インテル® IPP バージョン 1.1 の音声認識プリミティブの一部の関数名は、インテル IPP バージョン 2 の共通の命名規則に従って非推奨となった。次の表は、左の欄に古い関数、右の欄に新しい関数を示している。インテル IPP バージョン 2 では、互換性を保証するために、これらの古い関数の使用もサポートしているが、推奨されない。

表 8-2　インテル® IPP 2.0 で名前が変更された関数

古い関数	新しい関数名
ippsAddNRow	ippsAddNRows
ippsSumCol	ippsSumColumn
ippsSumAllRow	ippsCopyColumn
ippsCopyCol	ippsCopyColumn
ippsMeanCol	ippsMeanColumn
ippsVarCol	ippsVarColumn
ippsMeanVarCol	ippsMeanVarColumn
ippsNormalizeCol	ippsNormalizeColumn
ippsAddMulCol	ippsAddMulColumn
ippsQRTransCol	ippsQRTransColumn
ippsDotProdCol	ippsDotProdColumn
ippsMulCol	ippsMulColumn
ippsSumColAbs	ippsSumColumnAbs
ippsSumColSqr	ippsSumColumnSqr
ippsDeltaW1	ippsDelta_Win1
ippsDeltaW2	ippsDelta_Win2
ippsDeltaDeltaW1	ippsDeltaDelta_Win1
ippsDeltaDeltaW2	ippsDeltaDelta_Win2
ippsRecSqrt_Th	ippsRecSqrt
ippsLMThreshold	ippsScaleLM
ippsMahDist1	ippsMahDist
ippsMahDist2	ippsMahDist_MultiMix
ippsLogGauss1	ippsLogGauss
ippsLogGauss2	ippsLogGauss_MultiMix
ippsLogGaussMax1	ippsLogGaussMax
ippsLogGaussAdd1	ippsLogGaussAdd
ippsLogGaussAdd2	ippsLogGaussAdd_MultiMix

intel®

バージョン 2.0 との互換性

インテル® IPP バージョン 2.0 の多くの関数は、拡張された機能を持つ関数によって置き換えられ、インテル IPP バージョン 3 では非推奨となった。次の表は、左の欄に古い関数、右の欄に新しい関数を示している。インテル IPP バージョン 3 では、互換性を保証するために、これらの古い関数の使用もサポートしているが、推奨されない。古い関数の機能は、インテル IPP 3.0 関数の *scaleFactor* 引数を 0 に設定した場合と同等である。

表 8-3　インテル® IPP 3.0 で置き換えられた関数

古い関数	新しい関数名
ippsLogGaussSingle_16s32f	ippsLogGaussSingle_Scaled_16s32f
ippsLogGaussSingle_DirectVar_16s32f	ippsLogGaussSingle_DirectVarScaled_16s32f
ippsLogGaussSingle_IdVar_16s32f	ippsLogGaussSingle_IdVarScaled_16s32f
ippsLogGaussSingle_BlockDVar_16s32f	ippsLogGaussSingle_BlockDVarScaled_16s32f
ippsLogGauss_16s32f_D2	ippsLogGauss_Scaled_16s32f_D2
ippsLogGauss_16s32f_D2L	ippsLogGauss_Scaled_16s32f_D2L
ippsLogGauss_IdVar_16s32f_D2	ippsLogGauss_IdVarScaled_16s32f_D2
ippsLogGauss_IdVar_16s32f_D2L	ippsLogGauss_IdVarScaled_16s32f_D2L
ippsLogGaussMultiMix_16s32f_D2	ippsLogGaussMultiMix_Scaled_16s32f_D2
ippsLogGaussMultiMix_16s32f_D2L	ippsLogGaussMultiMix_Scaled_16s32f_D2L
ippsLogGaussMax_16s32f_D2	ippsLogGaussMax_Scaled_16s32f_D2
ippsLogGaussMax_16s32f_D2L	ippsLogGaussMax_Scaled_16s32f_D2L
ippsLogGaussMax_IdVar_16s32f_D2	ippsLogGaussMax_IdVarScaled_16s32f_D2
ippsLogGaussMax_IdVar_16s32f_D2L	ippsLogGaussMax_IdVarScaled_16s32f_D2L
ippsLogGaussMaxMultiMix_16s32f_D2	ippsLogGaussMaxMultiMix_Scaled_16s32f_D2
ippsLogGaussMaxMultiMix_16s32f_D2L	ippsLogGaussMaxMultiMix_Scaled_16s32f_D2L

次のインテル IPP バージョン 2 の関数名は、インテル IPP バージョン 3 では共通の命名規則に従って変更された。インテル IPP バージョン 3 では、互換性を保証するために、これらの古い関数の使用もサポートしているが、推奨されない。

表 8-4　インテル® IPP 3.0 で名前が変更された関数

古い関数	新しい関数名
ippsLogGaussAdd_16s32f_D2	ippsLogGaussAdd_Scaled_16s32f_D2
ippsLogGaussAdd_16s32f_D2L	ippsLogGaussAdd_Scaled_16s32f_D2L
ippsLogGaussAdd_IdVar_16s32f_D2	ippsLogGaussAdd_IdVarScaled_16s32f_D2
ippsLogGaussAdd_IdVar_16s32f_D2L	ippsLogGaussAdd_IdVarScaled_16s32f_D2L

表 8-4 インテル® IPP 3.0 で名前が変更された関数

古い関数	新しい関数名
ippsLogGaussAddMultiMix_16s32f_D2	ippsLogGaussAddMultiMix_Scaled_16s32f_D2
ippsLogGaussAddMultiMix_16s32f_D2L	ippsLogGaussAddMultiMix_Scaled_16s32f_D2L
ippsLPToLSP_32f_D2	ippsLPToLSP_32f
ippsLPToLSP_16s_D2Sfs	ippsLPToLSP_16s_Sfs
ippsLSPToLP_32f_D2	ippsLSPToLP_32f
ippsLSPToLP_16s_D2Sfs	ippsLSPToLP_16s_Sfs

intel®

音声符号化関数

9

本章では、ITU-T 勧告 G.729、G.723.1、G.722.1、G.726、G.728、GSM-AMR、および GSM-FR に準拠した音声コーデックの開発に使用できるインテル® IPP 関数について説明する。

これらの音声コーデックは、正しく作成された場合、公開されたテスト・ベクトルのビットごとの正確さの仕様に適合する。

本章は、最初にインテル IPP 音声符号化関数で使用される丸めモード、表記の規則、ヘッダ・ファイルの定義、およびデータ構造体について説明する。そして、関数は機能ごとに以下の項目に分けて説明する。

- 共通の関数
- G.729 に関連する関数
- G.723.1 に関連する関数
- GSM-AMR に関連する関数
- GSM フル・レートに関連する関数
- G.722.1 に関連する関数
- G.726 に関連する関数
- G.728 に関連する関数

各項では、最初にその関数グループ専用のインテル IPP 関数のリストを表で示した後、その API の詳細を説明する。

丸めモード

多くの音声コーデックは、ビットごとの正確さの必要条件を満たす必要がある。したがって、本章で説明するインテル® IPP 関数は、汎用信号処理関数に使用されるデフォルトの丸めモードとは異なる丸めモードを使用する。

汎用信号処理関数では、デフォルトの丸めモードを「最も近い偶数」として定義できる。したがって、固定小数点数 $x = N + \alpha,\ 0 \le \alpha < 1$ （N は整数）は、次のように丸められる。

$$\lceil x \rceil = \begin{cases} N, & 0 \le \alpha < 0.5 \\ N+1, & 0.5 < \alpha < 1 \\ N, & \alpha = 0.5,\ N-even \\ N+1, & \alpha = 0.5,\ N-odd \end{cases}$$

例えば、1.5 は 2 に丸められ、2.5 は 2 に丸められる。

本章の関数には、2 種類の丸めモードがある。

デフォルトの丸めモードは「切り捨て」である。このモードでは、固定小数点数の小数部分は切り捨てられ、丸めの結果は元の値より常に小さくなる。厳密には、固定小数点数 $x = N + \alpha, 0 \le \alpha < 1$（$N$ は整数）は、常に N に丸められる。
例えば、-1.3 は -2 に丸められ、1.7 は 1 に丸められる。このデフォルトの丸めモードを使用する関数の名前には、特別なサフィックスは付けられない。

もう 1 つの丸めモードは、「四捨五入（nearest right）」である。このモードでは、固定小数点数 $x = N + \alpha, 0 \le \alpha < 1$（$N$ は整数）は次のように丸められる。

$$\lceil x \rceil = \begin{cases} N, & 0 \le \alpha < 0.5 \\ N+1, & 0.5 \le \alpha < 1 \end{cases}$$

例えば、1.5 は 2 に丸められるが（「最も近い偶数」モードと同じ）、
-1.5 は -1 に丸められる（「最も近い偶数」モードでは、-1.5 は -2 に丸められる）。

「四捨五入」の丸めモードを使用する関数の名前には、サフィックス「NR」が付けられる。

表記の規則

本章では、本書全体を通じて使用される規則以外に、次の表記規則を使用する。

- 関数の引数の説明では、引数がベクトルである場合、引数の説明の後に示される大カッコ内の式 [n] によって、そのベクトルの要素の数（長さ）を指定する。
- 多くの音声符号化関数は、整数と整数配列の引数を固定小数点数として解釈することで、正しく実行する。この値は、特定の範囲内で変化する実数値を表す。引数の説明で使用される Qn という表記は、この引数値が整数値に 2^{-n} を掛けた実数値として、関数内の整数演算で使用されることを意味する（「n」はスケール係数と呼ばれる）。
  例えば、引数値が「Q12 の 4069」と説明されている場合、これは実数値 1.0 として解釈される。値が「スケール係数 14 の 15565」と説明されている場合、実数値 0.95 を表す。引数が「Q14 の [0, 1]」と説明されている場合、[0,16386] 範囲の整数値として渡す必要がある。

定義

この項では、本章の後半で説明するインテル® IPP 音声符号化関数 API を使用するために必要なヘッダ・ファイルの定義について説明する。ビット・レート指定子の定義については、データ構造体を参照のこと。

任意の音声符号化プリミティブに対してリンクする場合、次のコード例のようにヘッダ・ファイル ippdefs.h および ippsc.h をインクルードする必要がある。

```
#include "ippdefs.h"
#include "ippsc.h"
int main()
{
    ...
    /* call GSM-AMR IPP functions */
        ippsLevinsonDurbin_GSMAMR(pSrcAutoCorr, pSrcDstLpc);
    ...
}
```

データ構造体

音声符号化関数によっては、列挙型のビット・レート・パラメータ IppSpchBitRate を使用する。次の構造体で示すように、このセットにはサポートされた各ビット・レートに対して、それぞれ 1 つの識別子が含まれている。

```
typedef enum {
  IPP_SPCHBR_4750
  IPP_SPCHBR_5150
  IPP_SPCHBR_5300
  IPP_SPCHBR_5900
  IPP_SPCHBR_6300
  IPP_SPCHBR_6700
  IPP_SPCHBR_7400
  IPP_SPCHBR_7950
  IPP_SPCHBR_9600
  IPP_SPCHBR_10200
  IPP_SPCHBR_12200
  IPP_SPCHBR_12800
  IPP_SPCHBR_16000
  IPP_SPCHBR_24000
```

```
    IPP_SPCHBR_32000
    IPP_SPCHBR_40000
    IPP_SPCHBR_DTX
} IppSpchBitRate;
```

指定子 IPP_SPCHBR_4750、IPP_SPCHBR_5150、IPP_SPCHBR_5900、IPP_SPCHBR_6700、IPP_SPCHBR_7400、IPP_SPCHBR_7950、IPP_SPCHBR_10200、IPP_SPCHBR_12200 は、GSM-AMR 関数で使用され、それぞれ 4.75、5.15、5.9、6.7、7.4、7.95、10.2、12.2 Kbps/s の送信レートに対応する。
指定子 IPP_SPCHBR_5300 および IPP_SPCHBR_6300 は、G.723.1 コーデックで使用され、5.3 (low) および 6.3 (high) Kbps/s の送信レートに対応する。
指定子 IPP_SPCHBR_9600、IPP_SPCHBR_12800、および IPP_SPCHBR_16000 は、G.728 コーデックで使用され、9.6、12.8、および 16 Kbps/s の送信レートに対応する。

指定子 IPP_SPCHBR_16000、IPP_SPCHBR_24000、IPP_SPCHBR_32000、IPP_SPCHBR_40000 は、G.726 コーデックで使用され、それぞれ 16、24、32、40 Kbps/s の送信レートに対応する。

共通の関数

この項では、各種の音声コーデックに使用される共通の関数について説明する。これらすべての関数のリストを次の表に示す。

表 9-1　IPP の共通の音声符号化関数

関数の基本名	操作
ConvPartial	ID シグナルの線形のたたみ込みを実行する。
Mul_NR	2 つのベクトルの要素を掛け合わせる。
MulC_NR	ベクトルの各要素を定数値で掛ける。
MulPowerC_NR	ベクトルの各要素に対して累乗による重み付けを実行する。
AutoScale	最大の要素によってスケーリングする。
DotProdAutoScale	自動スケーリングを使用して、2 つのベクトルの内積を計算する。
InvSqrt	ベクトル要素の逆平方根を計算する。
AutoCorr	ベクトルの自己相関を計算する。
AutoCorrLagMax	ベクトルの最大自己相関を推定する。
AutoCorr_NormE	ベクトルの自己相関をノーマルで推定する。
CrossCorr	2 つのベクトルの相互相関を推定する。
CrossCorrLagMax	2 つのベクトルの最大相互相関を推定する。
SynthesisFilter	合成フィルタ 1/A(z) によって入力音声をフィルタリングし、音声信号を計算する。

ConvPartial

ID シグナルの線形のたたみ込みを実行する。

```
IppStatus ippsConvPartial_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsConvPartial_16s32s (const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, ipp32s* pDst, int len);
```

引数

pSrc1	先頭のソース・ベクトルへのポインタ。
pSrc2	2 番目のソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ソースおよびデスティネーション・ベクトルの要素の数。
scaleFactor	出力データのスケーリングに使用されるスケール係数。

説明

関数 ippsConvPartial_16s32s は、ippsc.h ファイルで宣言される。この関数は、次の式に従って、ベクトル *pSrc1* と *pSrc2* のたたみ込みを計算する。

$$pDst[i] = \sum_{j=0}^{i} pSrc1[i] \cdot pSrc2[i-j] \quad , \; i = 0, \ldots len-1$$

関数 ippsConvPartial_16s_Sfs は、次の式に従って、ベクトル *pSrc1* と *pSrc2* のたたみ込みを計算し、出力データをスケーリングする。

$$pDst[i] = 2^{-scaleFactor} \cdot \sum_{j=0}^{i} pSrc1[i] \cdot pSrc2[i-j] \quad , \; i = 0, \ldots len-1$$

計算された結果は切り捨てられる。

部分的たたみ込み関数の出力結果は、丸めによる違いを考慮に入れない場合、汎用信号処理たたみ込み関数 ippsConv_16s_Sfs の最初の *len* 個の結果と同じになる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsScaleRangeErr	エラー。*scaleFactor* が負。

Mul_NR

2 つのベクトルの要素を掛け合わせる。

```
IppStatus ippsMul_NR_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
    Ipp16s* pDst, int len, int scaleFactor);
IppStatus ippsMul_NR_16s_ISfs (const Ipp16s* pSrc, Ipp16s* pSrcDst, int
    len, int scaleFactor);
```

引数

pSrc1, pSrc2	互いに掛け合わされるベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrc	*pSrcDst* の要素が掛けられるベクトルへのポインタ（インプレース演算用）。
pSrcDst	ソースおよびデスティネーション・ベクトル（インプレース演算用）へのポインタ。
len	各ベクトルの要素の数。
scaleFactor	出力データのスケーリングに使用されるスケール係数。

説明

関数 ippsMul_NR は、ippsc.h ファイルで宣言される。この関数は、1 番目のソース・ベクトル *pSrc1* に 2 番目のソース・ベクトル *pSrc2* を要素ごとに掛けて、結果を *pDst* に格納する。

インプレース関数型は、ベクトル *pSrc* にベクトル *pSrcDst* を要素ごとに掛けて、結果を *pSrcDst* に格納する。

いずれの関数型も、*scaleFactor* の値に従って乗算結果をスケーリングする（第 2 章の 「整数のスケーリング」を参照）。出力値がデータ範囲を超えた場合は、結果は飽和される。

関数 ippsMul_NR は、ビットごとの正確さの必要条件を満たすために「四捨五入」の丸めを実行する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、*pSrc*、*pSrcDst*、または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsScaleRangeErr	エラー。*scaleFactor* がゼロより小さい。

MulC_NR

ベクトルの各要素を定数値で掛ける。

```
IppStatus ippsMulC_NR_16s_Sfs (Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst,
    int len, int scaleFactor);
IppStatus ippsMulC_NR_16s_ISfs (Ipp16s val, Ipp16s* pSrcDst, int len,
    int scaleFactor);
```

引数

val	ソース・ベクトルの各要素に掛けられるスカラ値。
pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースおよびデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	出力データのスケーリングに使用されるスケール係数。

説明

関数 ippsMulC_NR は、ippsc.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc* にスカラ値 *val* を掛けて、結果を *pDst* に格納する。

インプレース関数型は、ベクトル *pSrcDst* に *val* を掛けて、結果を *pSrcDst* に格納する。

いずれの関数型も、*scaleFactor* の値に従って乗算結果をスケーリングする（第 2 章の 「整数のスケーリング」 を参照)。出力値がデータ範囲を超えた場合は、結果は飽和される。

関数 ippsMulC_NR は、ビットごとの正確さの必要条件を満たすために「四捨五入」の丸めを実行する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsScaleRangeErr	エラー。*scaleFactor* がゼロより小さい。

MulPowerC_NR

ベクトルの各要素に対して累乗による
重み付けを実行する。

```
IppStatus ippsMulPowerC_NR_16s_Sfs (const Ipp16s* pSrc, Ipp16s val,
    Ipp16s* pDst, int len, int scaleFactor);
```

引数

pSrc	ソース・ベクトルへのポインタ。
val	ソース・ベクトルに掛けられるスカラ値。
pDst	デスティネーション・ベクトルへのポインタ。
len	各ベクトルの要素の数。
scaleFactor	出力データのスケーリングに使用されるスケール係数。

説明

関数 ippsMulPowerC_NR は、ippsc.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc* の各要素に、スカラ値 *val* を累乗した値を掛けて、結果を *pDst* に格納する。計算は次のように行われる。

$$pDst[i] = val^{i} \cdot pSrc[i]\ ,\ 0 \le i < len$$

この関数は、*scaleFactor* の値に従って乗算結果をスケーリングする（第 2 章の「整数のスケーリング」を参照）。出力値がデータ範囲を超えた場合は、結果は飽和される。

関数 ippsMulPowerC_NR は、ビットごとの正確さの必要条件を満たすために「四捨五入」の丸めを実行する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsScaleRangeErr	エラー。*scaleFactor* がゼロより小さい。

AutoScale

最大の要素によってスケーリングする。

```
IppStatus ippsAutoScale_16s (const Ipp16s *pSrc, Ipp16s *pDst, int len,
    int *pScale);
IppStatus ippsAutoScale_16s_I(const Ipp16s *pSrcDst, int len, int
    *pScale);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	インプレース演算用のソースおよびデスティネーション・ベクトルへのポインタ。
len	各ベクトルの要素の数。
pScale	入力 / 出力スケール係数へのポインタ。

説明

関数 ippsAutoScale は、ippsc.h ファイルで宣言される。この関数は、次の式に従って入力ベクトルをスケーリングする。

$$pDst[i] = 2^{scaleFactor} \cdot pSrc[i] \quad , \; i = 0, \ldots len-1$$

ここで、*scaleFactor* = *normMax*-*pScale*[0] である。
normMax は、ベクトルの最大値の絶対値が正規化されるように計算される。値 *pScale*[0] は、入力スケール係数である。このスケール係数は、処理後に *scaleFactor* で置き換えられる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、*pSrcDst*、または *pScale* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsScaleRangeErr	エラー。*pScale* によって指定されるスケール係数がゼロより小さい。

DotProdAutoScale

自動スケーリングを使用して、2 つのベクトルの内積を計算する。

```
IppStatus ippsDotProdAutoScale_16s32s_Sfs(const Ipp16s* pSrc1, const
    Ipp16s* pSrc2, int len, Ipp32s* pDp, int* pSfs);
```

引数

pSrc1	先頭のソース・ベクトルへのポインタ。
pSrc2	2 番目のソース・ベクトルへのポインタ。
len	各ベクトルの要素の数。
pDp	出力結果へのポインタ。
pSfs	スケール係数を返すポインタ。

説明

関数 ippsDotProdAutoScale は、ippsc.h ファイルで宣言される。この関数は、2 つのベクトルの内積を計算し、計算プロセスでオーバーフローが発生しないようにスケール係数を調整して、計算中に内積を自動的にスケーリングする。

$$pDp = 2^{scaleFactor} \cdot \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i]$$

intel®

最終的なスケール係数は、*pSfs* によって返される。
ベクトル *pSrc1* と *pSrc2* は、同じ長さ（*len*）でなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、*pDp* または *pSfs* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsOverflow	警告。少なくとも 1 つの結果の値が飽和された。

InvSqrt

ベクトル要素の逆平方根を計算する。

```
IppStatus ippsInvSqrt_32s_I (Ipp32s *pSrcDst, int len );
```

引数

pSrcDst	ソースおよびデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsInvSqrt は、ippsc.h ファイルで宣言される。この関数は、次の式に従って逆平方根を計算する。

$$pDst[i] = \frac{2^{31}}{\sqrt{pSrc[i]}}, \quad i = 0, \ldots len - 1$$

この計算には、近似値テーブルが使用される。入力ベクトルと出力ベクトルは、30 ビットだけスケーリングされる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

AutoCorr

ベクトルの自己相関を計算する。

```
IppStatus ippsAutoCorr_16s32s (const Ipp16s *pSrc, int srcLen,
Ipp32s *pDst, int dstLen );
```

引数

pSrc	1 番目のソース・ベクトル [*srcLen*] へのポインタ。
srcLen	ソース・ベクトルの長さ。
pDst	デスティネーション・ベクトル [*dstLen*] へのポインタ。
dstLen	デスティネーション・ベクトルの長さ（計算する自己相関値の数)。

説明

関数 ippsAutoCorr は、ippsc.h ファイルで宣言される。この関数は、次の式で入力ベクトルの自己相関を計算する。

$$pDst[n] = \sum_{i=n}^{srcLen-1} pSrc[i-n] \cdot pSrc[i] \quad , n = 0,...,dstLen-1$$

結果を *pDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*srcLen* または *dstLen* がゼロ以下。

AutoCorrLagMax

ベクトルの最大自己相関を推定する。

```
IppStatus ippsAutoCorrLagMax_Inv_16s (const Ipp16s* pSrc, int len, int
    lowerLag, int upperLag, Ipp32s* pMax, int* maxLag);
IppStatus ippsAutoCorrLagMax_Fwd_16s (const Ipp16s* pSrc, int len, int
    lowerLag, int upperLag, Ipp32s* pMax, int* maxLag);
```

引数

pSrc	ソース・ベクトルへのポインタ。
len	自己相関の長さ。
lowerLag	入力ラグの下限値。
upperLag	入力ラグの上限値。
pMax	相関の最大出力値へのポインタ。
maxLag	相関の最大値を保持する出力ラグへのポインタ。

説明

これらの関数は、ippsc.h ファイルで宣言される。ippsAutoCorrLagMax_Inv は、次の式に従って、指定されたラグ範囲内の最大自動相関を求める。

$$pMax = \max_n \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i-n] \quad , \; lowerLag \le n \le upperLag$$

関数 ippsAutoCorrLagMax_Fwd は、次の式に従って、指定されたラグ範囲内の最大自動相関を求める。

$$pMax = \max_n \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n] \quad , \; lowerLag \le n \le upperLag$$

複数の最大値が存在する場合は、関数は最初の最大値を返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pMax*、または *maxLag* が NULL。

ippStsSizeErr	エラー。*len* がゼロ以下。

AutoCorr_NormE

ベクトルの自己相関をノーマルで
推定する。

```
IppStatus ippsAutoCorr_NormE_16s32s (const Ipp16s* pSrc, int len,
    Ipp32s* pDst, int lenDst, int *pNorm);
IppStatus ippsAutoCorr_NormE_NR_16s (const Ipp16s* pSrc, int len,
    Ipp16s* pDst, int lenDst, int* pNorm);
```

引数

pSrc	Q12 のソース・ベクトルへのポインタ。
len	ソース・ベクトル内の要素の数。
pDst	ソース・ベクトルの推定された自己相関を格納するデスティネーション・ベクトルへのポインタ。
lenDst	デスティネーション・ベクトル内の要素の数。
pNorm	出力スケール係数へのポインタ。

説明

関数 ippsAutoCorr_NormE は、ippsc.h ファイルで宣言される。この関数は、ソース・ベクトル *pSrc* の自己相関を計算する。結果は最初の自己相関係数（エネルギー）の値に従ってスケーリングされる。厳密には、自己相関係数に係数 2^{norm} が掛けられる。ここで、最初の係数が正規化されるように、0 以上の *norm* が計算される。

$$pDst[n] = 2^{norm} \cdot \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n] \quad , \ 0 \le n < lenDst \ , \ 0 \le norm$$

ここで、

$$pSrc[i] = \begin{cases} pSrc[i] \ , & 0 \le i < len \\ 0 \ , & otherwise \end{cases}$$

対応するスケール係数 *norm* は、*pNorm* によって返される。

NR サフィックスが付いた関数は、「四捨五入」の丸めを実行する（丸めモードを参照）。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pNorm* が NULL。
ippStsSizeErr	エラー。*len* または *lenDst* がゼロ以下。
ippStsOverflow	警告。少なくとも 1 つの結果の値が飽和された。

CrossCorr

2 つのベクトルの相互相関を推定する。

```
IppStatus ippsCrossCorr_16s32s_Sfs (const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32s* pDst, int scaleFactor);
IppStatus ippsCrossCorr_NormM_16s (const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp16s* pDst);
```

引数

pSrc1	先頭のソース・ベクトルへのポインタ。
pSrc2	2 番目のソース・ベクトルへのポインタ。
len	ソースおよびデスティネーション・ベクトル内の要素の数。
pDst	デスティネーション・ベクトルへのポインタ。
scaleFactor	デスティネーション・ベクトルのスケール係数。

説明

関数 ippsCrossCorr は、ippsc.h ファイルで宣言される。この関数は、次の式に従って、ベクトル *pSrc1* とベクトル *pSrc2* の間の相互相関を推定する。

$$corr[n] = 2^{scaleFactor} \cdot \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i+n] \,,\; 0 \le n \le len \;,$$

ここで、

$$pSrc2[i] = \begin{cases} pSrc2[i] \;, & 0 \le i < len \\ 0 \;, & otherwise \end{cases}$$

結果はベクトル *pDst* に格納される。相関の和は飽和される。*scaleFactor* の値に従ってスケーリングが実行される。

NormM サフィックスが付いた関数は、相関の和の最大値の絶対値を正規化するスケール係数を使用する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pDst* が NULL。
ippStsScaleRangeErr	エラー。*scaleFactor* が負。
ippStsSizeErr	エラー。*len* がゼロ以下。

CrossCorrLagMax

2 つのベクトルの最大相互相関を推定する。

```
IppStatus ippsCrossCorrLagMax_16s (const Ipp16s *pSrc1, const
Ipp16s *pSrc2, int len, int lag, Ipp32s *pMax, int *maxLag);
```

引数

pSrc1	1 番目のソース・ベクトル [*len*] へのポインタ。
pSrc2	2 番目のソース・ベクトル [*len*+*lag*+1] へのポインタ。
len	相互相関の長さ。
lag	最大ラグ値。
pMax	相互相関の最大出力値へのポインタ。
maxLag	相互相関の最大値を保持する出力ラグへのポインタ。

説明

関数 ippsCrossCorrLagMax は、ippsc.h ファイルで宣言される。この関数は、次の式に従って 2 つの入力ベクトルから相互相関の最大値を検索する。

$$pMax = \max_{0 \le n \le lag} \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i+n]$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、*pMax*、または *maxLag* が NULL。
ippStsRangeErr	エラー。*lag* がゼロより小さい。
ippStsSizeErr	エラー。*len* がゼロ以下。

SynthesisFilter

合成フィルタ 1/A(z) によって入力音声をフィルタリングし、音声信号を計算する。

```
IppStatus ippsSynthesisFilter_NR_16s_Sfs(const Ipp16s * pLPC, const
    Ipp16s * pSrc, Ipp16s * pDst, int len, int scaleFactor, const
    Ipp16s *pMem);
IppStatus ippsSynthesisFilterLow_NR_16s_ISfs(const Ipp16s * pLPC,
    Ipp16s * pSrcDst, int len, int scaleFactor, const Ipp16s *pMem);
IppStatus ippsSynthesisFilter_NR_16s_ISfs(const Ipp16s * pLPC, Ipp16s *
    pSrcDst, int len, int scaleFactor, const Ipp16s *pMem);
```

引数

pLPC	入力フィルタ係数 $a_0, a_1,...,a_{10}$ へのポインタ。
pSrc	入力ベクトルへのポインタ。
pSrcDst	ヒストリ入力 / フィルタ出力ベクトルへのポインタ。
pDst	フィルタリングされた出力へのポインタ。
len	ベクトル内の要素の数。
scaleFactor	デスティネーション・ベクトルに対するに対するスケール係数。
pMem	フィルタリングに使用されるメモリへのポインタ。このポインタは、デスティネーション・ベクトルの最後の 10 個の値を使用して、関数の外部で更新する必要がある。

説明

関数 ippsSynthesisFilter は、ippsc.h ファイルで宣言される。この関数は、次の式に従って、合成フィルタによって入力信号をフィルタリングする。

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i) \quad , n = 0,1,...,len-1$$

関数型 ippsSynthesisFilterLow_NR_16s_ISfs は、飽和を実行しない。この関数型は、入力データが十分に小さいために正しい計算が行われることを前提にして、オーバーフロー状態のチェックを行わない。この動作モードは、関数名のサフィックス Low に示される。

すべての関数型は、ビットごとの正確さの必要条件を満たすために「四捨五入」の丸めを実行する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pLPC*、*pSrcDst*、*pDst*、または *pMem* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsScaleRangeErr	エラー。scaleFactor が 12 または 13 ではない。
ippStsOverflow	警告。少なくとも 1 つの結果の値が飽和された。

G.729 に関連する関数

この項で説明するインテル® IPP 関数は、ITU-T 勧告 G.729（[ITU729]、[ITU729A]、[ITU729B] を参照）に準拠した音声コーデックの作成に使用できるビルディング・ブロックである。

これらすべての関数のリストを次の表に示す。

表 9-2　G.729 に関連するインテル® IPP 関数

関数の基本名	操作
基本関数	
DotProd_G729	2 つのベクトルの内積を計算する。
Interpolate_G729	2 つのベクトルの重み付けされた和を計算する。
線形予測分析関数	
AutoCorr_G729	ベクトルの自己相関を推定する。
LevinsonDurbin_G729	自己相関係数から LP 係数を計算する。
LPCToLSP_G729	LP 係数を LSP 係数に変換する。
LSFToLSP_G729	線スペクトル周波数を LSP 係数に変換する。
LSFQuant_G729	LSF 係数を量子化する。
LSFDecode_G729	量子化された LSF をデコードする。
LSFDecodeErased_G729	フレームが消去された場合に、量子化された LSF を再穀 z する。
LSPToLPC_G729	LSP 係数を LP 係数に変換する。
LSPQuant_G729	LSP 係数を量子化する。
LSPToLSF_G729	LSP 係数を LSF 係数に変換する。
LagWindow_G729	60Hz 帯域幅の拡張を適用する。
コードブック検索関数	
OpenLoopPitchSearch_G729	最適なピッチ値を検索する。
AdaptiveCodebookSearch_G729	整数部分の遅延と小数部分の遅延を検索し、アダプティブ・ベクトルを計算する。
DecodeAdaptiveVector_G729	過去の励振を補間して、アダプティブ・コードブック・ベクトルを復元する。
FixedCodebookSearch_G729	固定コードブック・ベクトルを検索する。
ToeplizMatrix_G729	固定コードブック検索用の Toepliz 行列の 616 個の要素を計算する。
コードブック・ゲイン関数	
DecodeGain_G729	アダプティブ・コードブック・ゲインと固定コードブック・ゲインをデコードする。
GainControl_G729	アダプティブ・ゲイン・コントロールを計算する。
GainQuant_G729	2 段階の共役構造を持つコードブックを使用して、コードブック・ゲインを量子化する。
AdaptiveCodebookContribution_G729	アダプティブ・コードブックの影響を差し引いて、コードブック検索のターゲット・ベクトルを更新する。

表 9-2　　G.729 に関連するインテル® IPP 関数 （続き）

関数の基本名	操作
AdaptiveCodebookGain_G729	アダプティブ・コードブック・ベクトルのゲインとフィルタリングされたコードブック・ベクトルを計算する。
フィルタ関数	
ResidualFilter_G729	逆 LP フィルタを実行し、残留信号を得る。
SynthesisFilter_G729	LP 係数と残留信号から音声信号を再構築する。
LongTermPostFilter_G729	元の音声信号から長期的な情報を復元する。
ShortTermPostFilter_G729	残留信号から音声信号を復元する。
TiltCompensation_G729	短期フィルタ内のティルトを補正する。
HarmonicFilter	高調波フィルタを計算する。
HighPassFilterSize_G729	G729 ハイパス・フィルタのサイズを返す。
HighPassFilterInit_G729	ハイパス・フィルタを初期化する。
HighPassFilter_G729	G729 ハイパス・フィルタを実行する。
IIR16s_G729	IIR フィルタリングを実行する。
PhaseDispersionGetStateSize_G729D	フェーズ分散フィルタのメモリ・サイズをクエリする。
PhaseDispersionInit_G729D	フェーズ分散フィルタのメモリを初期化する。
PhaseDispersionUpdate_G729D	フェーズ分散フィルタ・ステートを更新する。
PhaseDispersion_G729D	フェーズ分散フィルタリングを実行する。
Preemphasize_G729A	ポスト・フィルタのプリエンファシスを計算する。
WinHybridGetStateSize_G729E	ハイブリッド窓モジュールのメモリ・サイズをクエリする。
WinHybridInit_G729E	ハイブリッド窓モジュールのメモリを初期化する。
WinHybrid_G729E	ハイブリッド窓を適用し、自己相関係数を計算する。
RandomNoiseExcitation_G729B	ガウス分布を持つランダム・ベクトルを初期化する。

基本関数

これらの関数は、エンコード・プロセスとデコード・プロセスの両方に適用できる。

DotProd_G729

2 つのベクトルの内積を計算する。

```
IppStatus ippsDotProd_G729A_16s32s (const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, int len, Ipp32s* pDp);
```

引数

pSrc1　　先頭のソース・ベクトルへのポインタ。

pSrc2	2 番目のソース・ベクトルへのポインタ。
len	各ベクトルの要素の数。
pDp	出力結果へのポインタ。

説明

関数 ippsDotProd_G729A は、ippsc.h ファイルで宣言される。この関数は、次の式に従って、2 つのソース・ベクトル pSrc1 と *pSrc2* の内積を計算する。

$$pDp = 2 \cdot \sum_{i=0}^{len/2} pSrc1[2 \cdot i] \cdot pSrc2[2 \cdot i]$$

ベクトル *pSrc1* と *pSrc2* は、同じ長さ（*len*）でなければならない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pDp* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsOverflow	警告。少なくとも 1 つの結果の値が飽和された。

Interpolate_G729

2 つのベクトルの重み付けされた和を
計算する。

```
IppStatus ippsInterpolate_G729_16s (const Ipp16s* pSrc1, const Ipp16s*
    pSrc2, Ipp16s* pDst, int len);
IppStatus ippsInterpolateC_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s
    val1, const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int
    scaleFactor);
IppStatus ippsInterpolateC_NR_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s
    val1, const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int
    scaleFactor);
```

引数

pSrc1	先頭のソース・ベクトルへのポインタ。

val1	1 番目のソース・ベクトルに掛けられる係数。
pSrc2	2 番目のソース・ベクトルへのポインタ。
val2	2 番目のソース・ベクトルに掛けられる係数。
pDst	(補間された) デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。
scaleFactor	デスティネーション・ベクトルに対するに対するスケール係数。

説明

これらの関数は、ippsc.h ファイルで宣言される。関数 ippsInterpolate_G729 は、次の式に従って、重み付けされた和を計算する。

$$pDst[i] = (pSrc1[i] + \mathrm{sign}(pSrc1[i])) >> 1 + (pSrc2[i] + \mathrm{sign}(pSrc2[i])) >> 1\ ,$$

$$i = 0, \ldots len - 1$$

関数 ippsInterpolateC_G729 と ippsInterpolateC_NR_G729 は、いずれも同じ公式を使用して、重み付けされた和を計算する。

$$pDst[i] = (val1 \cdot pSrc1[i] + val2 \cdot pSrc2[i]) >> scaleFactor\ ,$$
$$i = 0, \ldots len - 1$$

ただし、これらの関数は、出力結果に対して異なる丸めモードを適用する (丸めモードを参照)。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsScaleRangeErr	エラー。scaleFactor がゼロより小さい。

線形予測分析関数

この項で説明する関数は、LSP のコード化（量子化）およびデコードと、LPC 係数、LSP 係数、LSF 係数間の変換を実行する。

AutoCorr_G729

ベクトルの自己相関を推定する。

```
IppStatus ippsAutoCorr_G729B(const Ipp16s * pSrcSpch,
    Ipp16s * pResultAutoCorrExp, Ipp32s * pDstAutoCorr);
```

引数

pSrcSpch	入力音声信号ベクトル [240] へのポインタ。
pResultAutoCorrExp	自己相関係数の指数へのポインタ。
pDstAutoCorr	自己相関係数ベクトル [13] へのポインタ。

説明

関数 `ippsAutoCorr_G729` は、`ippsc.h` ファイルで宣言される。この関数は、入力音声信号の 11 個の自己相関係数とそれらの指数を計算する。この関数は、240 個の音声サンプルのベクトルに適用される。これらのサンプルは、過去の音声フレームから 120 サンプル、現在の音声フレームから 80 サンプル、将来のフレームから 40 サンプルで構成される。関数の機能は次のとおりである。

1. 最初に、次の式によって得られる非対称窓を音声サンプルに適用する。

$$w_{lp}(n) = \begin{cases} 0.54 - 0.46\cos\left(\dfrac{2n\pi}{399}\right), & n = 0, 1, \ldots, 199 \\ \cos\left(\dfrac{2(n-200)\pi}{159}\right), & n = 200, 201, \ldots, 239 \end{cases}$$

2. 次に、次の公式を使用して、窓処理された音声サンプル $s(i), i = 0, 1, \ldots, 239$ の自己相関を計算する。

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k), \quad k = 0, 1, \ldots, 11$$

3. 最後に、最初の自己相関係数（エネルギー）の値に従って、自己相関係数をスケーリングする。厳密には、自己相関係数に係数 2^{norm} が掛けられる。ここで、最初の係数が正規化されるように、0 より小さい *norm* が計算される。対応するスケール係数 *norm* は、*pResultAutoCorrExp* によって返される。

注 : 関数 ippsAutoCorr_G729B は、実際には、ippsMul_NR 関数と ippsAutoCorr_NormE 関数を組み合わせたものである。次のコードは、詳しい対応関係を示している。

```
{
      short sig_win[240];
      ippsMul_NR_16s_Sfs(pSrcSpch,window,sig_win,240,15);
      ippsAutoCorr_NormE_16s32s(sig_win,240,pDstAutoCorr,11,
      &pResultAutoCorrExp);
}
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcSpch*、*pResultAutoCorrExp*、または *pDstAutoCorr* が NULL。

LevinsonDurbin_G729

自己相関係数から LP 係数を計算する。

```
IppStatus ippsLevinsonDurbin_G729_32s16s(const Ipp32s* pSrcAutoCorr,
    int order, Ipp16s* pDstLPC, Ipp16s* pDstRc, Ipp16s*
    pResultResidualEnergy);
IppStatus ippsLevinsonDurbin_G729B(const Ipp32s * pSrcAutoCorr, Ipp16s
    * pDstLPC, Ipp16s * pDstRC, Ipp16s * pResultResidualEnergy);
```

引数

order	LP の次数。
pSrcAutoCorr	自己相関係数ベクトル [*order*+1] へのポインタ。
pDstLPC	出力 LP 係数 [*order*+1] へのポインタ。
pDstRC	出力反射係数ベクトル [*order*] へのポインタ。

pResultResidualEnergy 残留エネルギーへのポインタ。

説明

関数 `ippsLevinsonDurbin_G729` と `ippsLevinsonDurbin_G729B` は、`ippsc.h` ファイルで宣言される。これらの関数は、Levinson-Durbin アルゴリズムを使用して、自己相関係数から指定した次数の LP フィルタの線形予測（LP）係数を計算する。関数 `ippsLevinsonDurbin_G729` は、パラメータ *order* を使用する。関数 `ippsLevinsonDurbin_G729B` は、パラメータ *order* のデフォルト値（10）を使用する。

LP 係数 a_i, $i = 1, 2, \ldots, order$ を求めるには、次の一連の方程式を解く必要がある。

$$\sum_{i=1}^{order} a_i \times r(|i-k|) = -r(k), \quad k = 1,2,\ldots,order$$

これらの関数は次の手順を実行する。

1. Levinson-Durbin アルゴリズムを適用して、上の一連の方程式を解く。このアルゴリズムは、次の漸化式を使用する。

$$E^{[0]} = r(o)$$

for $i = 1$ to $order$

$$a_0^{[i-1]} = 1$$

$$k_i = -\left[\sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j)\right] / E^{[i-1]}$$

$$a_i^{[i]} = k_i$$

for $j = 1$ to $i-1$

$$a_j^{[i]} = a_j^{[i-1]} + k_i \times a_{i-j}^{[i-1]}$$

end

$$E^{[i]} = E^{[i-1]} - k_i^2 E^{[i-1]}$$

end

2. E を出力残留エネルギーとして設定し、k_i を反射係数として設定する。

3. このアルゴリズムに使用される LPC フィルタが不安定である（つまり、漸化式の計算中に、いくつかの $|k_i|$ が 1.0 に非常に近くなる）場合は、入力ベクトル *pSrcDstLPC* と *pSrcDstRC* の RC 係数の要素は変更されず、残留エネルギーはゼロに設定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcAutoCorr*、*pDstRC*、*pResultResidualEnergy*、または *pDstLPC* が NULL。

LPCToLSP_G729

LP 係数を LSP 係数に変換する。

```
IppStatus ippsLPCToLSP_G729_16s(const Ipp16s * pSrcLPC, const Ipp16s *
    pSrcPrevLSP, Ipp16s * pDstLSP);
IppStatus ippsLPCToLSP_G729A_16s(const Ipp16s * pSrcLPC, const Ipp16s *
    pSrcPrevLSP, Ipp16s * pDstLSP);
```

引数

pSrcLPC	Q12 の LP 係数ベクトル [11] へのポインタ。
pSrcPrevLSP	Q15 の直前の LP 係数ベクトル [10] へのポインタ。
pDstLSP	計算された LSP 係数ベクトル [10] へのポインタ。

説明

これらの関数は、ippsc.h ファイルで宣言される。これらの関数は、10 次 LP 係数を LSP 係数に変換する。
1 番目の関数 ippsLPCToLSP_G729 は、G.729/B コーデック向けに設計され、2 番目の関数 ippsLPCToLSP_G729A は G.729A コーデック向けに設計されている。
これらの関数は、次の手順を実行する。

1. 次の再帰的関係を使用して、$F_1(z)$ と $F_2(z)$ の多項式係数を計算する。

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

intel®

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i),\ i = 0,1,...,4,$$

ここで $f_1(0) = f_2(0) = 1.0$ である。

2. Chebyshev 多項式を使用して $F_1(z)$ と $F_2(z)$ を求める。Chebyshev 多項式は、次の式から得られる。

$$C_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + f_1(5)/2$$

$$C_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + f_2(5)/2$$

G.729/B, では、0 と π の間の等間隔の 60 個の点で $F_1(z)$ と $F_2(z)$ を求め、符号の変化がないかどうかチェックする。符号の変化は、根が存在することを示す。符号が変化している場合は、符号の変化の間隔を 4 回割って、根を求める。
G.729A では、0 と π の間の等間隔の 50 個の点で $F_1(z)$ と $F_2(z)$ を求め、符号の変化がないかどうかチェックする。符号の変化は、根が存在することを示す。符号が変化している場合は、符号の変化の間隔を 2 回割って、根を求める。

3. LSP 係数を求めるのに必要な 10 個の根が見つからない場合は、直前の LSP 係数を使用する。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcLPC*、*pSrcPrevLSP* または *pDstLSP* が NULL。

LSFToLSP_G729

線スペクトル周波数を LSP 係数に変換する。

```
IppStatus ippsLSFToLSP_G729_16s (const Ipp16s *pLSF, Ipp16s *pLSP);
```

引数

pLSF	[0,π] の範囲の Q13 の LSF ベクトル [10] へのポインタ。
pLSP	[-1;1] の範囲の Q15 の LSP ベクトル [10] へのポインタ。

説明

関数 ippsLSPToLSF_G729 は、ippsc.h ファイルで宣言される。この関数は、次の式に従って、線スペクトル周波数（LSF）を LSP 係数に変換する。

$$pLSP[i] = \cos(pLSF[i]),\ i = 1,...,10$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pLSP* または *pLSF* が NULL。
ippStsOverflow	警告。少なくとも 1 つの結果の値が飽和された。

LSFQuant_G729

LSF 係数を量子化する。

```
IppStatus ippsLSFQuant_G729_16s (const Ipp16s* pLSF, Ipp16s*
    pQuantLSFTable, Ipp16s* pQuantLSF, Ipp16s* quantIndex);
IppStatus ippsLSFQuant_G729B_16s (const Ipp16s* pLSF, Ipp16s*
    pQuantLSFTable, Ipp16s* pQuantLSF, Ipp16s* quantIndex);
```

引数

pLSF	[0,π] の範囲の Q13 の LSF ベクトル [10] へのポインタ。
pQuantLSFTable	以前に量子化された LSF を格納する、4 行 10 列の行列へのポインタ
pQuantLSF	[0,π] の範囲の Q13 の量子化された LSF ベクトルへのポインタ。
quantIndex	出力される複合コードブック・インデックス　*L*0、*L*1、*L*2、および *L*3 へのポインタ。

説明

これらの関数は、ippsc.h ファイルで宣言される。
関数 ippsLSFQuant_G729_16s は、スイッチド移動平均（MA）予測子を使用して、入力 LSF 係数と以前に量子化された LSF 係数の差を量子化する。
量子化は、2 段階のベクトル量子化機構（VQ）を使用して実行される。2 段階の VQ

は、コードブック **L1**（128 エントリ）を使用する 10 次元 VQ と、2 つのコードブック **L2** と **L3**（それぞれ 32 エントリ）を使用する 10 ビット分割型 VQ（それぞれ 5 次元）で構成される。

関数 `ippsLSFQuant_G729B_16s` は、（5 ビットと 4 ビットの）2 段階の分割型 VQ を使用して、LSF 係数を量子化する。SID-LPC 量子化手順に使用される 2 次 MA 予測子は、1 番目と 2 番目の MA 予測子を、それぞれ 0.6 と 0.4 の重み値を指定して線形補間したものとして計算される。量子化の第 1 段階は G.729 の場合と同じであるが、量子化テーブル（32 エントリ）の一部だけが使用される。第 2 段階は、G.729 の場合とは異なる。分割は行われず、2 番目のテーブル（16 エントリ）の一部だけが使用される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pLSF*、*pQuantLSFTable*、*pQuantLSF*、または *quantIndex* が NULL。
`ippStsLSFLow`	警告。対応するフィルタが低い安定性を持つ。
`ippStsLSFHigh`	警告。対応するフィルタが高い安定性を持つ。
`ippStsLSFLowAndHigh`	警告。対応するフィルタが低い安定性と高い安定性の両方を持つ。

LSFDecode_G729

量子化された LSF をデコードする。

```
IppStatus ippsLSFDecode_G729_16s (const Ipp16s *quantIndex, Ipp16s*
    pQuantLSFTable, Ipp16s* pQuantLSF);
IppStatus ippsLSFDecode_G729B_16s(const Ipp16s *quantIndex, Ipp16s
    *pQuantLSFTable, Ipp16s * pQuantLSF);
```

引数

quantIndex	インデックス *L*0、*L*1、*L*2、*L*3（[ITU729] の表 8/G.729 を参照）または *L*0、*L*1、*L*2（[ITU729B] の 4.3 項を参照）のベクトルへのポインタ。
pQuantLSFTable	Q13 の以前に量子化された 4 つの LSF ベクトルの入力 / 出力テーブル [4][10] へのポインタ。

pQuantLSF	Q13 の量子化された LSF 出力ベクトル [10] へのポインタ。

説明

これらの関数は、ippsc.h ファイルで宣言される。
関数 ippsLSFDecode_G729 は、量子化テーブルとインデックスから、量子化された LSF 係数を取得する。

関数 ippsLSFDecode_G729B は、SID フレームの量子化された LSF 係数を取得する。*L*0、*L*1、*L*2 インデックスだけが使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *quantIndex*、*pQuantLSFTable* または *pQuantLSF* が NULL。
ippStsLSFLow	警告。LSF が低い安定性だけを持つ。
ippStsLSFHigh	警告。LSF が高い安定性だけを持つ。
ippStsLSFLowAndHigh	警告。LSF が高い安定性と低い安定性の両方を持つ。

LSFDecodeErased_G729

フレームが消去された場合に、量子化された LSF を再構築する。

```
IppStatus ippsLSFDecodeErased_G729_16s (Ipp16s maIndex, Ipp16s
    *pQuantLSFTable, Ipp16s *pQuantLSF);
```

引数

maIndex	スイッチド MA 予測子（*L*0）。
pQuantLSFTable	Q12 の以前に量子化された 4 つの LSF ベクトルの入力 / 出力テーブル [4][10] へのポインタ。
pQuantLSF	Q12 の量子化された LSF 出力ベクトル [10] へのポインタ。

説明

関数 ippsLSFDecodeErased_G729 は、ippsc.h ファイルで宣言される。この関数は、以前にデコードされたスイッチド MA 予測子インデックス（*L*0）を使用して、量子化された LSF 係数を取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pQuantLSFTable*、または *pQuantLSF* が NULL。

LSPToLPC_G729

LSP 係数を LP 係数に変換する。

```
IppStatus ippsLSPToLPC_G729_16s(const Ipp16s * pSrcLSP, Ipp16s * pDstLPC);
```

引数

pSrcLSP	Q15 の LSP 係数ベクトル [10] へのポインタ。
pDstLPC	Q12 の LP 係数ベクトル [11] へのポインタ。

説明

関数 ippsLSPToLPC_G729 は、ippsc.h ファイルで宣言される。この関数は、一連の 10 次 LSP 係数を LP 係数に変換する。
この関数は、次の手順を実行する。

1. 次の再帰的関係を使用して、$F_1(z)$ と $F_2(z)$ の多項式係数を計算する。

for i = 1 to 5

$$f_1(i) = -2q_{2i-1} \times f_1(i-1) + 2f_1(i-2)$$

for j = i-1 down to 1

$$f_1^{[i]}(j) = f_1^{[i-1]}(j) - 2q_{2i-1} \times f_1^{[i-1]}(j-1) + f_1^{[i-1]}(j-2)$$

end

end

ここで、初期値は $f_1(0) = 1$ と $f_1(-1) = 0$ に設定される。
係数 $f_2(i)$ は、q_{2i-1} を q_{2i} で置き換えて、同じように計算される。

2. 次に、$F_1(z)$ と $F_2(z)$ にそれぞれ $1 + z^{-1}$ と $1 - z^{-1}$ を掛けて、次の式に従って $F_1'(z)$ と $F_2'(z)$ を求める。

$$f_1'(i) = f_1(i) + f_1(i-1) \ , i = 1,2,...5$$

$$f_2'(i) = f_2(i) - f_2(i-1) \ , i = 1,2,...5$$

3. 最後に、この関数は、$f_1'(i)$ と $f_2'(i)$ から次のように LP 係数を計算する。

$$a_i = \begin{cases} 0.5 \times f_1'(i) + 0.5 \times f_2'(i) \ , \ i = 1, 2, \ldots, 5 \\ 0.5 \times f_1'(11-i) - 0.5 \times f_2'(11-i) \ , \ i = 6, 7, \ldots, 10 \end{cases}$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcLSP* または *pDstLPC* が NULL。

LSPQuant_G729

LSP 係数を量子化する。

```
IppStatus ippsLSPQuant_G729_16s(const Ipp16s * pSrcLSP, Ipp16s
    *pSrcDstPrevFreq, Ipp16s * pDstQLSP, Ipp16s * pDstQLSPIndex);
```

引数

pSrcLSP	Q15 の LSP 係数ベクトル [10] へのポインタ。
pSrcDstPrevFreq	Q13 の直前および更新された 4 つの量子化周波数ベクトル [40] へのポインタ。要素 0 ～ 9 は最新のフレーム、要素 30 ～ 39 は最も古いフレームを示す。
pDstQLSP	Q15 の LSP 係数ベクトル [10] へのポインタ。
pDstQLSPIndex	長さが 2 の複合コードブック・インデックス *L0*、*L1*、*L2*、*L3* へのポインタ。最初の要素は、次の組み合わせでインデックス *L0* と *L1* を表す。すなわち、*L1* が最下位ビットから 7 ビットを占め、*L0* は *L1* に隣接する次の 1 ビットに置かれる。2 番目の要素は、次の組み合わせ

でインデックス *L2* と *L3* を表す。すなわち、*L3* は最下位ビットから 5 ビットを占め、*L2* は *L3* に隣接する次の 5 ビットに置かれる。

説明

関数 `ippsLSPQuant_G729` は、`ippsc.h` ファイルで宣言される。この関数は、量子化された LSP 係数とコードブック・インデックスを得る。この関数は、次の演算を実行する。

1. 正規化された周波数領域 [0, π] 内で、次の式に従って LSP 係数 q_i を LSF 係数 ω_i に変換する。

$$\omega_i = \arccos(q_i)\ ,\ i = 1,...,10$$

2. スイッチド 4 次 MA 予測を使用して、現在のフレームの LSF 係数を予測する。計算された係数と予測された係数の差は、2 段階のベクトル量子化機構を使用して量子化される。第 1 段階は、128 エントリ（7 ビット）を格納するコードブック **L1** を使用する 10 次元の VQ である。第 2 段階は、それぞれ 32 エントリ（5 ビット）を格納する 2 つの 5 次元コードブック **L2** と **L3** を使用する分割型 VQ として開発された 10 ビット VQ である。この量子化プロセスについて説明するには、最初にデコード・プロセスについて説明する方がわかりやすい。各係数は、次のように、2 つのコードブックの和から得られる。

$$\hat{l} = \begin{cases} \mathbf{L1}_i(L1) + \mathbf{L2}_i(L2)\ ,\ i = 1, \ldots, 5 \\ \mathbf{L1}_i(L1) + \mathbf{L3}_{i-5}(L3)\ ,\ i = 6, \ldots, 10 \end{cases}$$

L1、*L2*、*L3* はコードブック・インデックスである。

3. 現在のフレーム *m* の量子化されるベクトルは、次の式から得られる。

$$l_i = \left[\omega_i^{(m)} - \sum_{k=1}^{4} \hat{p}_{i,k}\hat{l}_i^{(m-k)}\right] \Big/ \left(1 - \sum_{k=1}^{4} \hat{p}_{i,k}\right)\ ,\ i = 1,...,10\ ,$$

$\hat{p}_{i,k}$ はスイッチド MA 予測子の係数である。最初のコードブック **L1** が検索され、（重み付けされていない）平均 2 乗誤差が最小限になるようなエントリ *L1* が選択される。続いて、2 番目のコードブック **L2** が検索され、第 2 段階の下位部分が定義される。量子化された LP 合成フィルタ内で強い共鳴が起こらないように、部分ベクトル $\hat{l}_i$, i = 1,…,5 は、隣接するベクトル間の最小距離が *J* になるように再調整される。この再調整の手順を以下に示す。

for i = 2,…,10

if $\hat{l}_{i-1} > \hat{l}_i - j$

$\hat{l}_{i-1} = (\hat{l}_i + \hat{l}_{i-1} - j)/2$

$\hat{l}_i = (\hat{l}_i + \hat{l}_{i-1} + j)/2$

end

end

ここで、最初のパスでは J は 0.0012 である。選択された第 1 段階のベクトル $L1$ と第 2 段階の下位部分 $L2$ を使用して、コードブック **L3** から第 2 段階の上位部分が検索される。ここでも、再調整手順を使用して、0.0012 の最小距離を保証する。結果のベクトル $\hat{l}_i$, i = 1, …,10 は再調整され、0.0006 の最小距離が保証される。現在のフレーム m の量子化された LSF 係数 $\hat{\omega}_i^{(m)}$ は、次のように、量子化機構の以前の出力 $\hat{l}_i^{(m-k)}$ と量子化機構の現在の出力 $\hat{l}_i^{(m)}$ の重み付けされた和から得られる。

$$\hat{\omega}_i^{(m)} = \left(1 - \sum_{k=1}^{4} \hat{p}_{i,k}\right)\hat{l}_i^{(m)} + \sum_{k=1}^{4} \hat{p}_{i,k}\hat{l}_i^{(m-k)}\quad,\ i = 1,...,10$$

2 つの MA 予測子が存在するが、どちらの MA 予測子を使用するかは、単一のビット $L0$ で指定される。重み付けされた平均 2 乗誤差が最小になるような予測子が選択される。

$$E_{lsf} = \sum_{i=1}^{10} w_i(\omega_i - \hat{\omega}_i)^2$$

重み w_i は、量子化されていない LSF 係数の関数としてアダプティブにされる。

$$w_1 = \begin{cases} 1.0 & ,\ if\ \omega_2 - 0.04\pi - 1 > 0 \\ 10(\omega_2 - 0.04\pi - 1)^2 + 1\ , & otherwise \end{cases}$$

$$w_i = \begin{cases} 1.0 & ,\ if\ \omega_{i+1} - \omega_{i-1} - 1 > 0 \\ 10(\omega_{i+1} - \omega_{i-1} - 1)^2 + 1\ , & otherwise \end{cases}\qquad ,2 \le i \le 9$$

$$w_{10} = \begin{cases} 1.0 & ,\ if\ -\omega_9 + 0.92\pi - 1 > 0 \\ 10(-\omega_9 + 0.92\pi - 1)^2 + 1\ , & otherwise \end{cases}$$

さらに、重み w_5 と w_6 には、それぞれ 1.2 が掛けられる。

4. LP フィルタ内で強い共鳴が起こらないように、結果のベクトル $\hat{l}_i$ に再調整手順が 2 回適用される（$J = 0.0012$ と $J = 0.0006$ を順に使用する）。

5. $\hat{\omega}_i$ の計算後、対応するフィルタの安定性がチェックされる。この処理は次のように行われる。

1）係数 $\hat{\omega}_i$ を値の小さい方から順番に並べる。

2）$\hat{\omega}_i < 0.005$ の場合は、$\hat{\omega}_i = 0.005$;

3) $\hat{\omega}_{i+1} - \hat{\omega}_i < 0.0391$ の場合は、$\hat{\omega}_{i+1} = \hat{\omega}_i + 0.0391$, i = 1, …,9;

4）$\hat{\omega}_{10}$ >3.315 の場合は、$\hat{\omega}_{10}$ =3.315

6. 量子化された LSF ベクトルを LSP ベクトルに変換する。

注： 関数 ippsLSPQuant は、実際には ippsLSPToLSF、ippsLSFQuant、および ippsLSFToLSP 関数を組み合わせたものである。次のコードは、詳しい対応関係を示している。

```
IppStatus ippsLSPQuant_G729_16s(const Ipp16s * pSrcLSP, Ipp16s
* pSrcDstPrevFreq, Ipp16s * pDstQLSP, Ipp16s * pDstQLSPIndex) {
   __ALIGN(8) short lsf[LP_ORDER];
   __ALIGN(8) short lsp_q[LP_ORDER];
   short q_index[4];
   IppStatus sts=ippsLSPToLSF_G729_16s(pSrcLSP, lsf);
   if (sts != ippStsNoErr) return sts;
   sts = ippsLSFQuant_G729_16s(lsf, pSrcDstPrevFreq, lsf_q,
q_index );
   if(sts != ippStsNoErr) return sts;
   pDstQLSPIndex[0] = (q_index[0]<<G729_NC0_B) | q_index[1];
   pDstQLSPIndex[1] = (q_index[2]<<G729_NC1_B) | q_index[3];
   ippsLSFToLSP_G729_16s(lsp_q, pDstQLSP)
   return ippStsNoErr;
}
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcLSP*、*pSrcDstPrevFreq*、*pDstQLSP*、または *pDstQLSPIndex* が NULL。

LSPToLSF_G729

LSP 係数を LSF 係数に変換する。

```
IppStatus ippsLSPToLSF_Norm_G729_16s (const Ipp16s *pLSP, Ipp16s
    *pLSF);
IppStatus ippsLSPToLSF_G729_16s (const Ipp16s *pLSP, Ipp16s *pLSF);
```

引数

pLSP	[-1:1] の範囲の Q15 の LSP ベクトルへのポインタ。
pLSF	LSF ベクトルへのポインタ。値は、（ippsLSPToLSF_G729_16s 関数の場合は）[0:π] の範囲で 13 ビット、（ippsLSPToLSF_Norm_G729_16s 関数の場合は）[0:0.5] の範囲で 15 ビットだけスケーリングされなければならない。

説明

これらの関数は、ippsc.h ファイルで宣言される。

関数 ippsLSPToLSF_Norm_G729 は、次に示すように LSP 係数を LSF 係数に変換する。

$$pLSF[i] = 2^{scaleFactor} \cdot \arccos(pLSP[i]) \quad , 0 \le i < 10$$

スケール係数には、1/π に掛けることで最初の LSF 係数が [0:0.5] の区間で正規化されるような値が選択される。

関数 ippsLSPToLSF_G729 は、結果が [0:π] の区間を超えないようにする。いずれの関数も、同じ近似値テーブルを使用して逆余弦を計算する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pLSP* または *pLSF* が NULL。

intel®

LagWindow_G729

60Hz 帯域幅の拡張を適用する。

```
IppStatus ippsLagWindow_G729_32s_I (Ipp32s* pSrcDst, int len);
```

引数

pSrcDst	自己相関ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsLagWindow_G729 は、ippsc.h ファイルで宣言される。この関数は、次のように、自己相関ベクトルに 60Hz 帯域幅の拡張を適用する。

$$r[0] = 1.0001 \cdot r[0]$$

$$r[i] = w_{lag}[i] \cdot r[i] \text{ , } i = 0, \ldots len - 1 \text{ ,}$$

ここで、

$$w_{lag}[i] = \exp\left(-\frac{1}{2} \cdot \left(2\pi f_0 \cdot \frac{i}{f_s}\right)^2\right)$$

また、次の式が使用される。
$f_0 = 60\text{Hz}$, $f_s = 8000\text{Hz}$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcDst* ポインタが NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsRangeErr	エラー。*len* > 12。

コードブック検索関数

コードブック検索関数は、アダプティブ・コードブックを使用してオープン・ループ・ピッチの推定を実行し、ACELP 励振（固定）コードブック内でパルスの最適な符号と位置を検索する。

OpenLoopPitchSearch_G729

最適なピッチ値を検索する。

```
IppStatus ippsOpenLoopPitchSearch_G729_16s(const Ipp16s * pSrc, Ipp16s
     * bestLag);
IppStatus ippsOpenLoopPitchSearch_G729A_16s(const Ipp16s * pSrc, Ipp16s
     * bestLag);
```

引数

pSrc	知覚的に重み付けされた音声信号ベクトル [170] へのポインタ。
bestLag	オープン・ループ・ピッチの検索結果へのポインタ。

説明

これらの関数は、ippsc.h ファイルで宣言される。これらの関数は、オープン・ループ法を使用してピッチ値を求める。最良のアダプティブ・コードブック遅延を検索する手順の複雑さを軽減するために、検索範囲は、候補となる遅延の周囲に制限される。候補となる遅延には、知覚的に重み付けされる音声の相関係数が最大になるような遅延が選択される。
関数 ippsOpenLoopPitchSearch_G729A は、G.729A コーデック向けに設計され、次のアルゴリズムを実行する。
1. オーバーフローとアンダーフローのチェックを実行する。

$$nSumTemp = \sum_{j=-71}^{40} sw(2j-1) \times sw(2j-1) ;$$

$nSumTemp \geq 2^{30}$ の場合は、オーバーフローが発生し、$sw'(i) = sw(i) >> 3$, i = -143,...,79 になる。

$nSumTemp < 2^{19}$ の場合は、アンダーフローが発生し、$sw'(i) = sw(i) << 3$, i= -143,...,79 になる。

それ以外の場合は、$sw'(i) = sw(i)$, i = -143,...,79 になる。

2. 相関を計算する。

$$R(k) = \sum_{n=0}^{39} sw'(2n) \times sw'(2n-k) \quad , k = 20,...,143$$

3. [20, 39] の範囲内で、$R(k)$ の最大値と最適な k_1 を求める。[40, 79] の範囲内で、$R(k)$ の最大値と最適な k_2 を求める。[80, 143] の範囲内で、偶数の k の中から、$R(k)$ の最大値と次善の k_3 を求める。$k_3' = k_3$ とする。

4. $R(k_3'+1) > R(k_3)$ の場合は、$k_3 = k_3'+1$ である。

5. $R(k_3'-1) > R(k_3)$ の場合は、$k_3 = k_3'-1$ である。

6. 3 つの最適な k_i, $i = 1,2,3$ について、相関の 3 つのピーク値を正規化する。

$$R'(k_i) = \frac{R(k_i)}{\sqrt{\sum_{n=0}^{39} sw'(2n-k_i) \times sw'(2n-k_i)}}$$

7. $R'(k_1)$ と $R'(k_2)$ を後処理する。

$$\begin{aligned}
&if\ (|2 \times k_2 - k_3| < 5)\ \{ \\
&\quad R'(k_2) = R'(k_2) + 0.25 \times R'(k_3)\ ; \\
&\} \\
&if\ (|3 \times k_2 - k_3| < 7)\ \{ \\
&\quad R'(k_2) = R'(k_2) + 0.25 \times R'(k_3)\ ; \\
&\} \\
&if\ (|2 \times k_1 - k_2| < 5)\ \{ \\
&\quad R'(k_1) = R'(k_1) + 0.2 \times R'(k_2)\ ; \\
&\} \\
&if\ (|3 \times k_1 - k_2| < 7)\ \{ \\
&\quad R'(k_1) = R'(k_1) + 0.2 \times R'(k_2)\ ; \\
&\}
\end{aligned}$$

8. k_1 ～ k_3 から、次の条件を満たす最適な遅延 k_{opt} を求める。

$$R'(k_{opt}) = \max\{R'(k_i),\ i = 1,...,3\}$$

関数 `ippsOpenLoopPitchSearch_G729` は、G.729/B コーデック向けに設計され、次のアルゴリズムを実行する。

1. オーバーフローとアンダーフローのチェックを実行する。

$$nSumTemp = \sum_{i=-143}^{79} sw(i) \times sw(i)$$

$nSumTemp \geq 2^{30}$ の場合は、オーバーフローが発生し、$sw'(i) = sw(i) >> 3$, i = -143,...,79 になる。

$nSumTemp < 2^{19}$ の場合は、アンダーフローが発生し、$sw'(i) = sw(i) << 3$, i= -143,...,79 になる。

それ以外の場合は、$sw'(i) = sw(i)$, i = -143,...,79 になる。

2. 相関を計算する。

$$R(k) = \sum_{i=0}^{79} sw'(i) \times sw'(i-k) \quad , k = 20,...,143$$

3. [20, 39] の範囲内で、$R(k)$ の最大値と最適な $k = k_1$ を求める。[40,79] の範囲内で、$R(k)$ の最大値と最適な $k = k_2$ を求める。[80,143] の範囲内で、$R(k)$ の最大値と最適な $k = k_3$ を求める。

4. 3 つの最適な k_i, i = 1,2,3 について、相関の 3 つのピーク値を正規化する。

$$R'(k_i) = \frac{R(k_i)}{\sqrt{\sum_{n=0}^{79} sw'(n-k_i) \times sw'(n-k_i)}}$$

2. 次の関係に従って、k_1 ～ k_3 から最適な遅延 k_{opt} を求める。

$$k_{opt} = k_1, \quad R'(k_{opt}) = R'(k_1)$$

$$if\ (R'(k_2) \geq 0.85R'(k_{opt}))\ 、\ k_{opt} = k_2,\ R'(k_{opt}) = R'(k_2)$$

$$if\ (R'(k_3) \geq 0.85R'(k_{opt}))\ 、\ k_{opt} = k_3,\ R'(k_{opt}) = R'(k_3)$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrc* または *bestLag* が NULL。

AdaptiveCodebookSearch_G729

整数部分の遅延と小数部分の遅延を検索し、
アダプティブ・ベクトルを計算する。

```
IppStatus ippsAdaptiveCodebookSearch_G729_16s(Ipp16s valOpenDelay, const
    Ipp16s * pSrcAdptTarget, const Ipp16s * pSrcImpulseResponse, Ipp16s *
    pSrcDstPrevExcitation, Ipp16s * pDstDelay, Ipp16s * pDstAdptVector,
    Ipp16s subFrame);
IppStatus ippsAdaptiveCodebookSearch_G729A_16s(Ipp16s valOpenDelay, const
    Ipp16s * pSrcAdptTarget, const Ipp16s * pSrcImpulseResponse, Ipp16s *
    pSrcDstPrevExcitation, Ipp16s * pDstDelay, Ipp16s * pDstAdptVector,
    Ipp16s subFrame);
IppStatus ippsAdaptiveCodebookSearch_G729D_16s (Ipp16s valOpenDelay,
    const Ipp16s* pSrcAdptTarget, const Ipp16s * pSrcImpulseResponse,
    Ipp16s* pSrcDstPrevExcitation, Ipp16s * pDstDelay, Ipp16s
    subFrame);
```

引数

valOpenDelay	[18,145] の範囲のオープン・ループ遅延。
pSrcAdptTarget	アダプティブ・コードブック検索ベクトル [40] のターゲット信号へのポインタ。
pSrcImpulseResponse	Q12 の重み付けされた合成フィルタ・ベクトル [40] のインパルス応答へのポインタ。
pSrcDstPrevExcitation	直前および更新された励振ベクトル [194] へのポインタ。
pDstDelay	整数部分の遅延および小数部分の遅延ベクトル [2] へのポインタ。
pDstAdptVector	アダプティブ・ベクトル [40] へのポインタ。
subFrame	サブフレーム番号、0 または 1。

説明

これらの関数は、ippsc.h ファイルで宣言される。これらの関数は、ターゲット信号と過去のフィルタリングされた励振を使用して、整数部分の遅延 T と小数部分の遅延 t を検索し、アダプティブ・ベクトルを計算する。これらの関数はサブフレームに適用される。

関数 **ippsAdaptiveCodebookSearch_G729A** は、G.729A コーデック向けに設計され、次のアルゴリズムを実行する。

1. 次の式の値が最大になるような、最適な遅延 k を検索する。

$$R_N(k) = \sum_{n=0}^{39} x(n)y_k(n) = \sum_{n=0}^{39} x(n)\sum_{i=0}^{n} u_k(i)h(n-i) = \sum_{i=0}^{39}\sum_{n=i}^{39} u_k(i)x(n)h(n-i)$$

$$= \sum_{n=0}^{39} u_k(n)x_d(n), \quad k = t_{min}, \ldots, t_{max}$$

$$x_d(n) = \sum_{i=n}^{39} x(i)h(i-n), \; n = 0,1,\ldots,39$$

最適な遅延 k を T として示す。

2. 検索結果 T が第 1 サブフレーム内で 85 より小さいか、第 2 サブフレーム内にある場合は、以下の式の値が最大になるような小数部分の遅延を検索する。

$$R_{Nt}(k) = \sum_{n=0}^{39} x_d(n)u_{kt}(n), \; t = -1,0,1, \; k = T$$

$$u_{kt}(n) = \sum_{i=0}^{9} u_k(n-k_p-i)b_{30}(t_p+3i) + \sum_{i=0}^{9} u_k(n-k_p+1+i)b_{30}(3-t_p+3i),$$

$n = 0,1,\ldots,39$

$$k_p = \begin{cases} T, & t = 0,-1 \\ T+1, & t = 1 \end{cases}, \qquad t_p = \begin{cases} -t, & t = 0,-1 \\ 2, & t = 1 \end{cases}$$

intel®

3. 小数部分の結果を t とすると、新しいアダプティブ・ベクトルは、次のように計算される。

$$v(n) = u_{kt}(n), n = 0,1,...,39$$

関数 **ippsAdaptiveCodebookSearch_G729** は、G.729/B コーデック向けに設計され、次のアルゴリズムを実行する。

1. 次の式の値が最大になるような、整数部分の遅延 k を検索する。

$$R(k) = \frac{\sum_{n=0}^{39} x(n)y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n)y_k(n)}}$$

ここで、y_k は、遅延 k の過去のフィルタ励振である。最適な k を T として示す。

2. 検索結果 T が第 1 サブフレーム内で 85 より小さいか、第 2 サブフレーム内にある場合は、以下の式の値が最大になるような小数部分の遅延 t を検索する。

$$R_t(T) = \sum_{i=0}^{3} R(T-i)b_{12}(t+3i) + \sum_{i=0}^{3} R(T+1+i)b_{12}(3-t+3i) \quad , t = 0,1,2$$

3. 最後に、過去の励振を補間して、アダプティブ・ベクトルを得る。
補間方法は、ippsAdaptiveCodebookSearch_G729A 関数で新しいアダプティブ・ベクトルを得るのに使用したのと同じ方法である。

関数 **ippsAdaptiveCodebookSearch_G729D** は ippsAdaptiveCodebookSearch_G729 とは異なり、第 2 サブフレームのラグ数が 32 から 16 に減少する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcAdptTarget*、*pSrcImpulseResponse*、*pSrcDstPrevExcitation*、*pDstDelay* または *pDstAdptVector* が NULL。

IppStsRangeErr	エラー。*valOpenDelay* が [18, 145] の範囲内にないか、*subFrame* が 0 または 1 でない。

DecodeAdaptiveVector_G729

過去の励振を補間して、アダプティブ・コードブック・ベクトルを復元する。

```
IppStatus ippsDecodeAdaptiveVector_G729_16s(const Ipp16s * pSrcDelay,
    Ipp16s * pSrcDstPrevExcitation, Ipp16s * pDstAdptVector);
IppStatus ippsDecodeAdaptiveVector_G729_16s_I(const Ipp16s * pSrcDelay,
    Ipp16s * pSrcDstPrevExcitation);
```

引数

pSrcDelay	各サブフレームの整数部分および小数部分の遅延へのポインタ。
pSrcDstPrevExcitation	過去および更新された励振ベクトル [194] へのポインタ。
pDstAdptVector	アダプティブ・コードブック・ベクトル [40] へのポインタ。

説明

関数 ippsDecodeAdaptiveVector_G729 は、ippsc.h ファイルで宣言される。この関数は、過去の励振を補間して、アダプティブ・コードブック・ベクトルを復元する。
この関数は、次の手順を実行する。

1. ピッチ遅延の整数部分 T と小数部分 t をデコードする。

2. 次のように、過去の遅延を補間して、現在のアダプティブな励振を得る。

$$v(n) = \sum_{i=0}^{9} u(n-T-i) \times b_{30}(t+3i) + \sum_{i=0}^{9} u(n-T+1+i) \times b_{30}(3-t+3i),\ n = 0,1,...,39$$

戻り値

ippStsNoErr	エラーなし。

intel®

ippStsNullPtrErr	エラー。ポインタ *pSrcDelay*、*pSrcDstPrevExcitation* または *pDstAdptVector* が NULL。
ippStsRangeErr	エラー。*pSrcDelay*[0] が [18, 145] の範囲内にないか、*pSrcDelay*[1] が 0 または 1 でない。

FixedCodebookSearch_G729

固定コードブック・ベクトルを検索する。

```
IppStatus ippsFixedCodebookSearch_G729_16s(const Ipp16s *
    pSrcFixedCorr, Ipp16s * pSrcDstMatrix, Ipp16s * pDstFixedVector,
    Ipp16s * pDstFixedIndex, Ipp16s * pSearchTimes, Ipp16s subFrame);
IppStatus ippsFixedCodebookSearch_G729_32s16s(const Ipp16s *
    pSrcFixedCorr, Ipp32s * pSrcDstMatrix, Ipp16s * pDstFixedVector,
    Ipp16s * pDstFixedIndex, Ipp16s * pSearchTimes, Ipp16s subFrame);
IppStatus ippsFixedCodebookSearch_G729A_16s(const Ipp16s *
    pSrcFixedCorr, const Ipp16s * pSrcDstMatrix, Ipp16s *
    pDstFixedVector, Ipp16s * pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729A_32s16s(const Ipp16s *
    pSrcFixedCorr, const Ipp32s * pSrcDstMatrix, Ipp16s *
    pDstFixedVector, Ipp16s * pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729E_16s(int mode, const Ipp16s*
    pSrcFixedTarget, const Ipp16s* pSrcLtpResidual, const Ipp16s*
    pSrcImpulseResponse, Ipp16s* pDstFixedVector, Ipp16s*
    pDstFltFixedVector, Ipp16s* pDstFixedIndex);
IppStatus ippsFixedCodebookSearch_G729D_16s(Ipp16s *pSrcFixedCorr,
    Ipp16s *pSrcDstMatrix, Ipp16s *pSrcImpulseResponse, Ipp16s
    *pDstFixedVector, Ipp16s *pDstFltFixedVector, Ipp16s
    *pDstFixedIndex);
```

引数

mode	逆方向 LP 分析モード（*mode* = 1)、順方向 LP 分析モード（*mode* = 0）を示す。
pSrcFixedCorr	Q13 の固定コードブック検索ベクトル [40] の相関信号へのポインタ。
pSrcDstMatrix	長さが 616（Q9 の Ipp16s 型または Q25 の Ipp32s 型の要素数）の Toepliz 行列へのポインタ。
pSrcFixedTarget	更新された入力ターゲット音声ベクトルへのポインタ。

pSrcLtpResidual	長期予測の後の入力残留信号ベクトルへのポインタ。
pSrcImpulseResponse	LP 合成フィルタ [40] へのポインタ。
pDstFixedVector	Q13 の固定コードブック・ベクトル [40] へのポインタ。
pDstFltFixedVector	フィルタリングされた出力固定コードブック・ベクトルへのポインタ。
pDstFixedIndex	固定コードブック・インデックス [2] へのポインタ。最初の要素は符号コードワード *S*、2 番目の要素は位置コードワード *C* である。
pSearchTimes	残りのサブフレームの最大検索時間へのポインタ。
subFrame	サブフレーム番号、0 または 1。

説明

これらの関数は、`ippsc.h` ファイルで宣言される。これらの関数は、固定コードブック・ベクトルとそれに対応するベクトル・インデックスを検索する。これらの関数はサブフレームに適用される。

1 番目の関数 `ippsFixedCodebookSearch_G729` は、G.729/B コーデック向けに設計され、ネスト・ループ検索法を使用する。この関数内で Toepliz 行列が更新される。この関数は、次の手順を実行する。

1. 最初に、Toepliz 行列の 616 個の要素を次のように更新する。

$$\Phi'(i, j) = \mathrm{sign}[d(i)]\mathrm{sign}[d(j)]\Phi(i, j)$$

ここで、$\Phi(i, j)$ は Toepliz 行列の要素、$d(n)$ は相関信号である。

2. 次に、次の式を使用して、しきい値 thr_3 を計算する。

$$thr_3 = av_3 + 0.4(max_3 - av_3)$$

ここで、max_3 と av_3 は、$d(n)$ 内の最初の 3 つのパルスの最大値の絶対値と平均値の絶対値である。

3. 次に、項 C^2/E が最小になるような固定コードブック・ベクトルを検索し、ベクトル・インデックスを求める。ここで、*C* は次の式で指定される相関である。

$$C = |d(m_0)| + |d(m_1)| + |d(m_2)| + |d(m_3)| \quad (9\text{-}1)$$

また、*E* は次の式から得られるエネルギーである。

$$E = 2(\Phi'(m_0, m_0) + \Phi'(m_1, m_1) + \Phi'(m_0, m_1) + \Phi'(m_2, m_2) + \Phi'(m_0, m_2) + \Phi'(m_1, m_2)) \quad (9\text{-}2)$$

$$+\Phi'(m_3, m_3) + \Phi'(m_0, m_3) + \Phi'(m_1, m_3) + \Phi'(m_2, m_3))$$

検索手順は 4 ループ検索の一種である。それまでに計算された相関がしきい値 thr_3 を超えた場合にのみ、最後のループが実行される。また、コードブックの低いパーセンテージが検索されるように、ループを反復できる最大回数は 180 に固定されている。

インデックスの符号コードワード *S* は、次のように計算される。

$$S = s_0 + 2s_1 + 4s_2 + 8s_3 \qquad (9\text{-}3)$$

インデックスの位置コードワード *C* は、次のように計算される。

$$C = (m_0/5) + 8(m_1/5) + 64(m_2/5) + 512(2(m_3/5) + jx), \qquad (9\text{-}4)$$

ここで、$m3 = 3, 8, \ldots, 38$ の場合は $jx = 0$、$m3 = 4, 9, \ldots, 39$ の場合は $jx = 1$ である。

4. 最後に、固定コードブック・ベクトルを計算する。

関数 `ippsFixedCodebookSearch_G729A` は、G.729A コーデック向けに設計され、反復深さ優先のツリー検索法を使用する。
この関数の機能は次のとおりである。

1. 最初に、次の公式を使用して、古い Toepliz 行列 $\Phi'(i, j)$ から、新しい Toepliz 行列 Φ の 616 個の要素を計算する。

$$\Phi'(i, j) = \mathrm{sign}[d(i)]\mathrm{sign}[d(j)]\Phi(i, j) \quad ,$$

ここで、$\Phi(i, j)$ は Toepliz 行列の要素、$d(n)$ は相関信号である。

2. 次に、項 C^2/E が最小になるような固定コードブック・ベクトルを検索し、ベクトル・インデックスを求める。ここで、*C* は（9-1）で指定される相関、*E* は（9-2）から得られるエネルギーである。

検索には、反復深さ優先のツリー検索法を使用する。

インデックスの符号コードワード *S* は、（9-3）に従って計算される。インデックスの位置コードワード *C* は、（9-4）から得られる。

3. 最後に、固定コードブック・ベクトルを計算する。

関数 ippsFixedCodebookSearch_G729 と ippsFixedCodebookSearch_G729A は、次の表で示す位置から 4 個の符号付きパルスを検索する。

パルス	位置
i_0	0,5,10,15,20,25,30,35
i_1	1,6,11,16,21,26,31,36
i_2	2,7,12,17,22,27,32,37
i_3	3,8,13,18,23,28,33,38 4,9,14,19,24,29,34,39

関数 ippsFixedCodebookSearch_G729E は、G.729E コーデック向けに設計され、5 つのトラックから 10 個の符号付きパルスを次の表で示す位置から検索する。

パルス	位置
i_0,i_1	0,5,10,15,20,25,30,35
i_3,i_4	1,6,11,16,21,26,31,36
i_5,i_6	2,7,12,17,22,27,32,37
i_7,i_8	3,8,13,18,23,28,33,38
i_9,i_{10}	4,9,14,19,24,29,34,39

各トラックの 2 個のパルスがオーバーラップした場合、振幅が 2 重の 1 個のパルスとなる。

関数 ippsFixedCodebookSearch_G729D は、G.729D コーデック向けに設計され、次の表で示す 2 つの重複したトラックから 2 個の符号付きパルスを検索する。

パルス	位置
i_0	1, 3, 6, 8, 11, 13, 16, 18, 21, 23, 26, 28, 31, 33, 36, 38
i_1	0, 1, 2, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 16, 17, 19, 20, 21, 22, 24, 25, 26, 27, 29, 30, 31, 32, 34, 35, 36, 37, 39

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意のポインタが NULL。
IppStsRangeErr	エラー。*subFrame* が 0 または 1 ではない。

ToeplizMatrix_G729

固定コードブック検索用の Toepliz 行列の
616 個の要素を計算する。

```
IppStatus ippsToeplizMatrix_G729_16s(const Ipp16s *
    pSrcImpulseResponse, Ipp16s * pDstMatrix);
IppStatus ippsToeplizMatrix_G729_16s32s(const Ipp16s *
    pSrcImpulseResponse, Ipp32s * pDstMatrix);
```

引数

pSrcImpulseResponse	Q12 のインパルス応答ベクトル [40] へのポインタ。
pDstMatrix	長さが 616 の Toepliz 行列の要素へのポインタ（Q9 の `Ipp16s` 型または Q25 の `Ipp32s` 型）。

説明

関数 `ippsToeplizMatrix_G729` は、`ippsc.h` ファイルで宣言される。この関数は、固定コードブック検索用の Toepliz 行列の 616 個の要素を計算する。これらの要素は、次のように表現できる。

$$\Phi(i, j) = \sum_{n=j}^{39} h(n-i) \times h(n-j) \ , 0 \le i \le 39 \ , 0 \le j \le 39$$

ここで、$h(i)$, $i = 0, 1, \ldots, 39$ はインパルス応答である。

この関数は、計算された 616 個の要素を、次の順番で *pDstMatrix* に格納する。

1. $\Phi(m_i, m_i)$ ($i = 0, 1, 2, 3$), 3x8+16 = 40 個の要素、開始位置 : 0

   $\Phi(0, 0), \Phi(5, 5), \ldots, \Phi(35, 35)$, $\Phi(1, 1), \Phi(6, 6), \ldots, \Phi(36, 36)$,

   $\Phi(2, 2), \Phi(7, 7), \ldots, \Phi(37, 37)$, $\Phi(3, 3), \Phi(8, 8), \ldots, \Phi(39, 39)$

2. $\Phi(m_0, m_1)$, 8x8 = 64 個の要素、開始位置 : 40

   $\Phi(0, 1), \ldots, \Phi(0, 36)$, $\Phi(5, 1), \ldots, \Phi(5, 36)$, $\Phi(10, 1), \ldots, \Phi(35, 36)$

3. $\Phi(m_0, m_2)$, 8x8 = 64 個の要素、開始位置 : 104

   $\Phi(0, 2), \ldots, \Phi(0, 37)$, $\Phi(5, 2), \ldots, \Phi(5, 37)$, $\Phi(10, 2), \ldots, \Phi(35, 37)$

4. $\Phi(m_0, m_3)$, 8x16 = 128 個の要素、開始位置 : 168

$\Phi(0, 3), \ldots, \Phi(0, 38)$ ， $\Phi(5, 3), \ldots, \Phi(5, 38)$ ， $\Phi(10, 3), \ldots, \Phi(35, 38)$

$\Phi(0, 4), \ldots, \Phi(0, 39)$ ， $\Phi(5, 4), \ldots, \Phi(5, 39)$ ， $\Phi(10, 4), \ldots, \Phi(35, 39)$

5. $\Phi(m_1, m_2)$, 8x8 = 64 個の要素、開始位置 : 296

   $\Phi(1, 2), \ldots, \Phi(1, 37)$ ， $\Phi(6, 2), \ldots, \Phi(6, 37)$ ， $\Phi(11, 2), \ldots, \Phi(36, 37)$

6. $\Phi(m_1, m_3)$, 8x16 = 128 個の要素、開始位置 : 360

   $\Phi(1, 3), \ldots, \Phi(1, 38)$ ， $\Phi(6, 3), \ldots, \Phi(6, 38)$ ， $\Phi(11, 3), \ldots, \Phi(36, 38)$

   $\Phi(1, 4), \ldots, \Phi(1, 39)$ ， $\Phi(6, 4), \ldots, \Phi(6, 39)$ ， $\Phi(11, 4), \ldots, \Phi(36, 39)$

7. $\Phi(m_2, m_3)$, 8x16 = 128 個の要素、開始位置 : 488

   $\Phi(2, 3), \ldots, \Phi(2, 38)$ ， $\Phi(7, 3), \ldots, \Phi(7, 38)$ ， $\Phi(12, 3), \ldots, \Phi(37, 38)$

   $\Phi(2, 4), \ldots, \Phi(2, 39)$ ， $\Phi(7, 4), \ldots, \Phi(7, 39)$ ， $\Phi(12, 4), \ldots, \Phi(37, 39)$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcImpulseResponse* または *pDstMatrix* が NULL。

コードブック・ゲイン関数

コードブック・ゲイン関数は、G.729 のエンコード / デコード手順における固定コードブック・ゲインとアダプティブ・コードブック・ゲインの予測、量子化、デコード、制御に使用できる。

DecodeGain_G729

アダプティブ・コードブック・ゲインと固定コードブック・ゲインをデコードする。

```
IppStatus ippsDecodeGain_G729_16s (Ipp32s energy, Ipp16s *pPastEnergy,
    const Ipp16s *pQuaIndex, Ipp16s *pGain);
IppStatus ippsDecodeGain_G729I_16s(Ipp32s energy,  Ipp16s
    valGainAttenuation, Ipp16s *pPastEnergy, const Ipp16s *pQuaIndex,
    Ipp16s *pGain);
```

引数

energy	コードワードの入力エネルギー。
pPastEnergy	直前の 4 つのサブフレームの固定コードブックの影響を示すログ・エネルギーの入力 / 出力ベクトル（Q14）へのポインタ。
pQuaIndex	フレームが消去された場合は NULL。それ以外の場合は、コードブック・インデックスのベクトルへのポインタ。 *pQuaIndex* [0] - 第 1 段階のコードブック・インデックス *pQuaIndex* [1] - 第 2 段階のコードブック・インデックス
valGainAttenuation	フレームが消去された場合におけるゲインの減衰係数（*pQuaIndex* = NULL）。
pGain	デコードされた入力 / 出力ゲインへのポインタ。 *pGain*[0] - アダプティブ（ピッチ）ゲイン。 *pGain*[1] - 固定コードブック・ゲイン。

説明

関数 `ippsDecodeGain_G729` は、`ippsc.h` ファイルで宣言される。この関数は、アダプティブ・コードブック・ゲインと固定コードブック・ゲインをデコードする。固定コードブック・ゲイン g_c は、次のように表現される。

$g_c = \gamma\ g_c'$,

ここで、 g_c' は直前の固定コードブック・エネルギーに基づいて予測されるゲインであり、γ は補正係数である。予測されるゲイン g_c' は、係数 [0.68, 0.58, 0.34, 0.19] を持つ 4 次 MA 予測子を使用して、直前の固定コードブックの影響のログ・エネルギーから現在の固定コードブックの影響のログ・エネルギーを予測することによって得られる。
[0.68, 0.58, 0.34, 0.19]

アダプティブ・コードブック・ゲインと係数は、構造を持つコードブックを使用してベクトル量子化される。各コードブックの最初の要素は、量子化されたアダプティブ・コードブック・ゲインを表す。2 番目の要素は、量子化された固定コードブック補正係数を表す。フレームが消去された場合は、ゲインは直前のゲインの減衰版になる。

ippsDecodeGain_G729_16s。第 1 段階が 3 ビット、第 2 段階が 4 ビットの 2 段階の共役構造を持つ 2 次元コードブックが使用される。フレームが消去された場合は、アダプティブ・コードブック・ゲインと固定コードブック・ゲインの減衰にはそれぞれ Q15 の係数 0.9 と 0.98 が使用される。

ippsDecodeGain_G729I_16s。新しい 6 ビットの共役構造を持つコードブックが使用される。フレームが消去された場合は、アダプティブ・コードブック・ゲインと固定コードブック・ゲインの両方の減衰に *valGainAttenuation* が提供する係数が使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pPastEnergy* または *pGain* が NULL。

GainControl_G729

アダプティブ・ゲイン・コントロールを計算する。

```
IppStatus ippsGainControl_G729_16s_I (const Ipp16s* pSrc, Ipp16s*
    pSrcDst, Ipp16s* pGain);
IppStatus ippsGainControl_G729A_16s_I (const Ipp16s* pSrc, Ipp16s*
    pSrcDst, Ipp16s* pGain);
```

引数

pSrc	再構築されたソース音声ベクトル [40] へのポインタ。
pSrcDst	ポスト・フィルタリングされたソース信号とゲイン・スケーリングおよびポスト・フィルタリングされたデスティネーション信号のベクトル [40] へのポインタ。
pGain	出力アダプティブ・ゲインへのポインタ。

説明

これらの関数は、ippsc.h ファイルで宣言される。関数 ippsGainControl_G729 は、（*pSrc* から得られる）再構築された音声信号 *sr* と（*pSrcDst* から得られる）フィルタリングされた信号 *sf* のゲインの差を補正する。最初に、次の式に従ってゲイン係数 *G* が計算される。

$$G = \frac{\sum_{i=0}^{39} |sr[i]|}{\sum_{i=0}^{39} |sf[i]|}$$

ポスト・フィルタリングされた信号 *pSrcDst* の出力ゲイン *spf* は、次のように計算される。

$$spf(n) = g^{(n)} \cdot sf(n)\ ,\ n = 0,..., 39\ ,$$

ここで、

$$g^{(n)} = 0.85 \cdot g^{(n-1)} + 0.15 \cdot G\ ,\ n = 0,..., 39\ ,\ g^{(-1)} = 1.0$$

この関数は *pGain* に $g^{(39)}$ を返す。

関数 `ippsGainControl_G729A_16s` では、ゲイン係数 G と係数 $g^{(n)}$ は、次の式に従って計算される。

$$G = \sqrt{\frac{\sum_{i=0}^{39} |sr[i]^2|}{\sum_{i=0}^{39} |spf[i]^2|}}$$

$$g^{(n)} = 0.9 \cdot g^{(n-1)} + 0.1 \cdot G$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pSrcDst* または *pGain* が NULL。

GainQuant_G729

2 段階の共役構造のコードブックを使用して、
コードブック・ゲインを量子化する。

```
IppStatus ippsGainQuant_G729_16s(const Ipp16s * pSrcAdptTarget, const
    Ipp16s * pSrcFltAdptVector, const Ipp16s * pSrcFixedVector, const
    Ipp16s *pSrcFltFixedVector, Ipp16s *pSrcDstEnergyErr, Ipp16s *
    pDstQGain, Ipp16s * pDstQGainIndex, Ipp16s tameProcess);
IppStatus ippsGainQuant_G729D_16s(const Ipp16s* pSrcAdptTarget, const
    Ipp16s* pSrcFltAdptVector, const Ipp16s* pSrcFixedVector, const
    Ipp16s  *pSrcFltFixedVector, Ipp16s* pSrcDstEnergyErr, Ipp16s
    *pDstQGain, Ipp16s *pDstQGainIndex, Ipp16s tameProcess)
```

引数

pSrcAdptTarget	Q11 のアダプティブ・コードブック検索ベクトル [40] のターゲット信号へのポインタ。
pSrcFltAdptVector	Q11 のフィルタリングされたアダプティブ・コードブック・ベクトル [40] へのポインタ。
pSrcFixedVector	Q12 の事前に重み付けされた固定コードブック・ベクトル [40] へのポインタ。
pSrcFltFixedVector	Q12 のフィルタリングされた固定コードブック・ベクトル [40] へのポインタ。
pSrcDstEnergyErr	Q10 の直前の予測エネルギー誤差ベクトル [4] へのポインタ。
pDstQGain	量子化されたゲイン（G_p と G_c）へのポインタ。
pDstQGainIndex	ゲイン・コードブック・インデックス（*GA* と *GB*）へのポインタ。
tameProcess	テイミング・プロセス・インジケータ。

説明

関数 `ippsGainQuant_G729` と `ippsGainQuant_G729D` は、`ippsc.h` ファイルで宣言される。これらの関数は、2 段階の共役構造を持つコードブックを使用する。関数 `ippsGainQuant_G729_16s` は、7 ビットのコードブックを使用してゲインを量子化するのに対して、関数 `ippsGainQuant_G729D_16s` は 6 ビットのコードブックを使用する。ゲイン・コードブックの検索では、元の音声と再構築された音声の間の重み付けされた平均 2 乗誤差が最小になるようなゲインが検索される。

$$E = \mathbf{x}^t\mathbf{x} + g_p^2\mathbf{y}^t\mathbf{y} + g_c^2\mathbf{z}^t\mathbf{z} - 2g_p\mathbf{x}^t\mathbf{y} - 2g_c\mathbf{x}^t\mathbf{z} + 2g_pg_c\mathbf{y}^t\mathbf{z},$$

ここで、$\mathbf{x}$ はアダプティブ・ターゲット、$\mathbf{y}$ はフィルタリングされたアダプティブ・ベクトル、$\mathbf{z}$ はフィルタリングされた固定ベクトルである。g_p はピッチ・ゲイン、g_c は固定ゲインである。

この関数の機能は次のとおりである。

1. 平均エネルギーから固定コードブックの影響の平均エネルギーを引いた値を計算する。

$$E = 10\log\left(\frac{1}{40}\sum_{n=0}^{39} c(n)^2\right)$$

次に、$Eb - E = 30(dB) - E$ を計算する。

2. 直前のサブフレームの予測の誤差から、予想されるエネルギーを計算する。

$$\tilde{E}^m = \sum_{i=1}^{4} b_i\hat{U}^{(m-i)}$$

$\hat{U}^m$ は、次の式で計算される。$U^m = E - \tilde{E}^m$

3. 次の式によって g_c を計算する。$g_c = 10^{(\tilde{E}^m + Eb - E)/20}$

4. 次の要素を計算する。

$G0 = \mathbf{y}^t\mathbf{y}$, $G1 = -2\mathbf{x}^t\mathbf{y}$, $G2 = \mathbf{z}^t\mathbf{z}$, $G3 = -2\mathbf{x}^t\mathbf{z}$, $G4 = 2\mathbf{y}^t\mathbf{z}$

5. 最適なゲイン G_p と G_c を次のように計算する。

$$G_p = -\frac{2\times G1\times G2 - G3\times G4}{4\times G0\times G2 - G4\times G4} \quad , \quad G_c = -\frac{2\times G0\times G3 - G1\times G4}{4\times G0\times G2 - G4\times G4}$$

6. G_p と G_c が存在する可能性が最も高い領域をあらかじめ選択する。出力は *index*0 と *index*1 である。

7. コードブック *GA* と *GB* 内のゲイン・ベクトルを選択する。検索範囲は *index*0 と *index*1 によって制限される。検索基準として、次の式が最小になるようなゲインが検索される。

$$E = G0 \times G_p^2 + G2 \times G_c^2 + G1 \times G_p + G3 \times G_c + G4 \times G_p \times G_c$$

8. U^m を更新する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcAdptTarget*、*pSrcFltAdptVector*、*pSrcFixedVector*、*pSrcFltFixedVector*、*pSrcDstEnergyErr*、*pDstQGain* または *pDstQGainIndex* が NULL。
IppStsRangeErr	エラー。*tameProcess* が 0 または 1 ではない。

AdaptiveCodebookContribution_G729

アダプティブ・コードブックの影響を差し引いて、コードブック検索のターゲット・ベクトルを更新する。

```
IppStatus ippsAdaptiveCodebookContribution_G729_16s (Ipp16s gain, const
    Ipp16s* pFltAdptVector, const Ipp16s* pSrcAdptTarget, Ipp16s*
    pDstAdptTarget);
```

引数

gain	アダプティブ・コードブック・ゲイン g_p。
pFltAdptVector	フィルタリングされた入力アダプティブ・コードブック・ベクトル $y(n)$ へのポインタ。
pSrcAdptTarget	入力ターゲット・ベクトル $x(n)$ へのポインタ。
pDstAdptTarget	更新された出力ターゲット・ベクトル $x'(n)$ へのポインタ。

説明

関数 ippsAdaptiveCodebookContribution_G729 は、ippsc.h ファイルで宣言される。この関数は、ターゲット信号からアダプティブ・コードブック・ベクトルの影響を差し引いて、更新されたターゲット・ベクトルを格納する。

$$x'(n) = x(n) - g_p \cdot y(n) \quad , n = 0,..., 39$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pFltAdptVector*、*pSrcAdptTarget* または *pDstAdptTarget* が NULL。

AdaptiveCodebookGain_G729

アダプティブ・コードブック・ベクトルのゲインと、フィルタリングされたコードブック・ベクトルを計算する。

```
IppStatus ippsAdaptiveCodebookGain_G729_16s(const Ipp16s * pSrcAdptTarget,
    const Ipp16s * pSrcImpulseResponse, const Ipp16s * pSrcAdptVector,
    Ipp16s * pDstFltAdptVector, Ipp16s * pResultAdptGain);
IppStatus ippsAdaptiveCodebookGain_G729A_16s(const Ipp16s * pSrcAdptTarget,
    const Ipp16s * pSrcLPC, const Ipp16s * pSrcAdptVector, Ipp16s *
    pDstFltAdptVector, Ipp16s * pResultAdptGain);
```

引数

pSrcAdptTarget	アダプティブ・ターゲット信号ベクトル [40] へのポインタ。
pSrcImpulseResponse	Q12 の知覚重み付けフィルタ・ベクトル [40] のインパルス応答へのポインタ。
pSrcAdptVector	Q12（ippsAdaptiveCodebookGain_G729_16s の場合）または Q10（ippsAdaptiveCodebookGain_G729A_16s の場合）のアダプティブ・コードブック・ベクトル [40] へのポインタ。
pSrcLPC	Q12 の合成フィルタ・ベクトル [11] の LPC 係数へのポインタ。
pDstFltAdptVector	フィルタリングされたアダプティブ・コードブック・ベクトル [40] へのポインタ。
pResultAdptGain	Q14 のアダプティブ・コードブック・ゲインへのポインタ。

説明

これらの関数は、ippsc.h ファイルで宣言される。これらの関数は、それぞれ、アダプティブ・コードブック・ベクトルのゲインの計算と、フィルタリングされたアダプティブ・コードブック・ベクトルの計算を実行する。いずれの関数もサブフレーム内で適用される。
1 番目の関数 ippsAdaptiveCodebookGain_G729 は、G.729/B コーデック向けに設計され、次の機能を持つ。

1. 最初に、次の公式を使用して、インパルス応答 $h(n)$ を使用してアダプティブ・コードブック・ベクトル $v(n)$ をたたみ込み、フィルタリングされたアダプティブ・コードブック・ベクトル $y(n)$ を得る。

$$y(n) = \sum_{i=0}^{n} v(i)h(n-i), \; n = 0,1,...,39$$

2. 次に、次の式に従ってアダプティブ・コードブック・ゲイン g_p を計算する。

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)},$$

ここで、$x(n)$ はアダプティブ・ターゲット信号である。

3. 最後に、得られたアダプティブ・コードブック・ゲイン g_p を [0, 1.2] の範囲に制限する。

2 番目の関数 ippsAdaptiveCodebookGain_G729A は、G.729A コーデック向けに設計され、次の機能を持つ。

1. 最初に、次の公式を使用して、アダプティブ・コードブック・ベクトル $v(n)$ に合成フィルタ $1/A_q(z/\gamma)$ を適用し、フィルタリングされたアダプティブ・コードブック・ベクトル $v(n)$ を得る。

$$y(n) = v(n) - \sum_{i=1}^{10} a_i y(n-i), \; n = 0,1,...,39$$

2. 次に、アダプティブ・コードブック・ゲイン g_p を計算する。

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)},$$

ここで、$x(n)$ はアダプティブ・ターゲット信号である。

3. 最後に、得られたアダプティブ・コードブック・ゲイン g_p を [0, 1.2] の範囲に制限する。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcAdptTarget*、*pSrcImpulseResponse*、*pSrcAdptVector*、*pSrcLPC*、*pDstFltAdptVector* または *pResultAdptGain* が NULL。

フィルタ関数

フィルタ関数を使用して、エンコードの前処理段階およびデコードの後処理段階のハイパス・フィルタリングや、デコード時の長期および短期ポストフィルタなど、各種のフィルタリングを実行できる。
残留信号フィルタ（FIR）、単純合成フィルタ（IIR）、高調波フィルタ、プリエンファシス・フィルタ関数を組み合わせて、さらに複雑な合成フィルタリングを実行できる。これらのフィルタ関数は、異なるエンコード段階とデコード段階で使用できる。

ResidualFilter_G729

逆 LP フィルタを実行し、
残留信号を得る。

```
IppStatus ippsResidualFilter_G729_16s(const Ipp16s * pSrcSpch, const
    Ipp16s * pSrcLPC, Ipp16s * pDstResidual);
IppStatus ippsResidualFilter_G729E_16s(const Ipp16s *pCoeffs, int
    order, const Ipp16s *pSrc, Ipp16s *pDst, int len)
```

引数

pSrcSpch	入力音声信号へのポインタ。要素 *pSrcSpch*[0...39] は現在の音声信号、*pSrcSpch*[-10... -1] は使用されるヒストリである。
pSrcLPC	Q12 の LP 係数ベクトル [11] へのポインタ。
pDstResidual	LP 残留信号へのポインタ。
pCoeffs	フィルタ係数のベクトル [*order*+1] へのポインタ。
order	フィルタの次数。
pSrc	ソース・ベクトル [*len*] へのポインタ。値 *pSrc* [-*order*,..,-1] も提供する必要がある。
pDst	フィルタリングされた出力残留信号ベクトル [*len*] へのポインタ。
len	ソースおよびデスティネーション・ベクトルの長さ。

説明

関数 `ippsResidualFilter_G729` と `ippsResidualFilter_G729E` は、`ippsc.h` ファイルで宣言される。

ippsResidualFilter_G729。この関数は、次のように、逆 LP フィルタによって入力音声信号をフィルタリングし、残留信号を得る。

$$r(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i \times s(n-i) \ , n = 0,1,...,39$$

ippsResidualFilter_G729E。この関数は、フィルタの次数が格納された変数を使用して、上記の関数と同じ演算を実行する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcSpch*、*pSrcLPC*、*pDstResidual*、*pSrC*、*pCoeffs*、または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

SynthesisFilter_G729

LP 係数と残留信号から音声信号を再構築する。

```
IppStatus ippsSynthesisFilter_G729_16s(const Ipp16s * pSrcResidual,
    const Ipp16s * pSrcLPC, Ipp16s * pSrcDstSpch);
IppStatus ippsSynthesisFilterZeroStateResponse_NR_16s(const Ipp16s*
    pSrcLPC, Ipp16s* pDstImp, int len, int scaleFactor);
IppStatus ippsSynthesisFilter_G729E_16s(const Ipp16s *pLPC, int order,
    const Ipp16s *pSrc, Ipp16s *pDst, int len, const Ipp16s *pMem);
IppStatus ippsSynthesisFilter_G729E_16s_I(const Ipp16s *pLPC, int
    order, Ipp16s *pSrcDst, int len, const Ipp16s *pMem);
```

引数

pSrcResidual	LP 残留信号ベクトル [40] へのポインタ。
pSrcLPC	Q12 の LP 係数ベクトル [11] へのポインタ。
pSrcDstSpch	合成および更新された音声へのポインタ。要素 *pSrcDstSpch*[0...39] は現在の合成音声、*pSrcDstSpch*[-10...-1] は使用されるヒストリである。
pLPC	LP 係数ベクトル [*order*+1] へのポインタ。
order	フィルタの次数（*pLPC*[*order*+1] ベクトルを入力時に提供する必要がある）。
pSrc	ソース・ベクトル [*len*] へのポインタ。
pDst	デスティネーション音声ベクトル [*len*] へのポインタ。

pSrcDst	ソースおよびデスティネーション・ベクトル [*len*] へのポインタ。
pDstImp	デスティネーション・ゼロ・ステートのインパルス応答ベクトル [*len*] へのポインタ。
len	ソースおよびデスティネーション・ベクトルの長さ。
pMem	フィルタリングに使用されるフィルタ・メモリ・ベクトル [*order*] へのポインタ。
scaleFactor	入力 LP 係数と出力インパルス応答のスケール係数。

説明

これらの関数は、ippsc.h ファイルで宣言される。

関数 ippsSynthesisFilter_G729_16s は、デフォルト LP フィルタ（10）を使用して、残留信号から音声信号を再構築する。

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i) \quad , n = 0,1,...,39$$

関数 ippsSynthesisFilterZeroStateResponse_NR_16s はメモリを使用せずに、ippsSynthesisFilter_NR_16s と同じパラメータで同じ演算を行い、次のように設定されたインパルス入力ベクトル *pSrc* を処理する。

$pSrc[0] = 2^{scaleFactor}$,
$pSrc[i] = 0$, $i = 1,... len - 1$

関数 ippsSynthesisFilter_G729E_16s と ippsSynthesisFilter_G729E_16s_I は、パラメータ *order* で指定した順序の合成フィルタリングを実行する。これらの関数は、デフォルト順序（10）を指定した関数 ippsSynthesisFilter_G729_16s のように動作する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。任意のポインタが NULL。
ippStsOverflow	警告。オーバーフローが発生した。
ippStsSizeErr	エラー。*len* がゼロ以下。

LongTermPostFilter_G729

古い音声信号から長期的な
情報を復元する。

```
IppStatus ippsLongTermPostFilter_G729_16s (Ipp16s gammaFactor, int
    valDelay, const Ipp16s *pSrcDstResidual, Ipp16s *pDstFltResidual,
    Ipp16s *pResultVoice);
IppStatus ippsLongTermPostFilter_G729A_16s(Ipp16s valDelay, const
    Ipp16s * pSrcSpch, const Ipp16s * pSrcLPC, Ipp16s *
    pSrcDstResidual, Ipp16s * pDstFltResidual);
IppStatus ippsLongTermPostFilter_G729B_16s(Ipp16s valDelay, const
    Ipp16s * pSrcSpch, const Ipp16s * pSrcLPC, Ipp16s *
    pSrcDstResidual, Ipp16s * pDstFltResidual, Ipp16s * pResultVoice,
    Ipp16s frameType);
```

引数

gammaFactor	Q15 のポスト・フィルタ係数 y_p。
valDelay	ピッチ遅延。
pSrcSpch	Q15 の再構築された音声 $\hat{s}(n)$ へのポインタ。要素 *pSrcSpch*[0...39] は現在の音声信号、要素 *pSrcSpch*[-10...-1] は使用されるヒストリである。
pSrcLPC	Q12 の LP 係数 $\hat{a}_i$ ベクトル [11] へのポインタ。
pSrcDstResidual	長期予測の後の入力残留信号ベクトル [40] へのポインタ。要素 [-152...-1] は使用される長期予測ヒストリである。
pDstFltResidual	フィルタリングされた出力残留信号ベクトル [40] へのポインタ。
pResultVoice	音声情報へのポインタ。
frameType	フレームのタイプ。 0 - 未送信フレーム 1 - 通常の音声フレーム 2 - SID フレーム

説明

これらの関数は、ippsc.h ファイルで宣言される。これらの 2 つの FIR フィルタを使用して、ピッチの長さで古い音声から長期的な関係を復元できる。これらの関数はサブフレームに適用される。

1 番目の関数 **ippsLongTermPostFilter_G729** は、次の機能を持つ。

1. 最初に、次の公式を使用して、$\hat{r}(n)$ の自己相関 $R(k)$ を計算し、[int(T1)-1, int(T1)+1] の範囲内で、$R(k)$ が最大になるような最も良い整数 T0 を見つける。

$$R(k) = \sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}(n-k) \qquad (9\text{-}6)$$

ここで、int(T1) は、最初のサブフレーム内のピッチ遅延 T1 の整数部分である。

2. 次に、T0 の周囲で、解像度 1/8 を使用して、最良の遅延の小数部分 T を探す。疑似正規化相関 $R'(k)$ が最大になるような遅延を見つける。

$$R'(k) = \frac{\sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}_k(n)}{\sqrt{\sum_{n=0}^{39} \hat{r}_k(n) \cdot \hat{r}_k(n)}},$$

$\hat{r}_k(n)$ は、遅延 k の残留信号である。

3. 最適な遅延 T が見つかったら、次の条件を使用して、長期フィルタを無効にするかどうかを決定する。

$$\frac{R'(T)^2}{\sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}(n)} < 0.5 \text{ の場合は、} g_l = 0 \text{ とする。} \qquad (9\text{-}7)$$

intel®

それ以外の場合は、

$$g_1 = \frac{\sum_{n=0}^{39} \hat{r}(n) \cdot \hat{r}_k(n)}{\sum_{n=0}^{39} \hat{r}_k(n) \cdot \hat{r}_k(n)}$$ とし、[0, 1.0] の範囲に制限する。 (9-8)

4. 最後に、次の FIR フィルタを使用する。

$$y(n) = \frac{1}{1 + \gamma_p \cdot g_1}(\hat{r}(n) + \gamma_p \cdot g_1 \cdot \hat{r}(n - T)) \quad , \qquad (9\text{-}9)$$

ここで、$\hat{r}(n)$ は残留信号である。係数 γ_p は、長期ポスト・フィルタの量を制御する。

2 番目の関数 **`ippsLongTermPostFilter_G729A`** は、G.729A コーデック向けに設計され、次の演算を実行する。

1. 最初に、次の公式を使用して、合成音声 $\hat{s}(n)$ から残留信号 $\hat{r}(n)$ を得る。

$$\hat{r}(n) = \hat{s}(n) + \sum_{i=1}^{10} \gamma_n^i \cdot \hat{a}_i \cdot \hat{s}(n - i) \quad , \qquad (9\text{-}5)$$

ここで、$\hat{s}(n)$ は合成音声、$\hat{a}_i$ は量子化された LP 係数である。

$\gamma_n = 0.55$

2. 次に、公式 (9-6) を使用して、$\hat{r}(n)$ の自己相関 $R(k)$ を計算し、[int(T1)-3, int(T1)+3] の範囲内で、$R(k)$ が最大になるような最も良い整数 T を見つける。

ここで、int(T1) は、最初のサブフレーム内のピッチ遅延 T1 の整数部分である。

3. 最適な遅延 T が見つかったら、関係（9-7）と（9-8）を使用して、長期フィルタを無効にするかどうかを決定する。

4. 最後に、FIR フィルタ（9-9）を使用する。

3 番目の関数 **`ippsLongTermPostFilter_G729B`** は、実際には、次のコードで示す他の関数を組み合わせたものである。

```
{
  const short g_pst[11] ={32767,18022,9912,5451,2998,1649,907,499,274,151,83};
      /* yn = 0.55 powered by i=0,1,…,10 in Q15 */
   short coeffs[LP_ORDER+1];      /* temporary nominator coefficients  */
   short vc;

   ippsMul_NR_16s_Sfs(g_pst, pSrcLpc, coeffs, LP_11, 15);
   ippsResidualFilter_G729_16s(pSrcSpch, coeffs, pSrcDstResidual+154);
   if (1 == frameType){
     ippsLongTermPostFilter_G729_16s (16384,valDelay, pSrcDstResidual + 154,
               pDstFltResidual, &vc); /* Harmonic filtering : gp=0.5 in Q15*/
      *pResultVoice = (vc != 0);
   }else{
      ippsCopy_16s(pSrcDstResidual + 154,pDstFltResidual,40);
      *pResultVoice = 0;
   }
}
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcSpch*、*pSrcLPC*、*pSrcDstResidual*、*pDstFltResidual* または *pResultVoice* が NULL。
ippStsRangeErr	エラー。*valDelay* が [18, 145] の範囲内にないか (ippsLongTermPostFilter_G729A 関数の場合)、または *valDelay* が [0, 143] の範囲内にないか (ippsLongTermPostFilter_G729B 関数の場合)、*frameType* が [0, 2] の範囲内にない。

ShortTermPostFilter_G729

残留信号から音声信号を復元する。

```
IppStatus ippsShortTermPostFilter_G729_16s(const Ipp16s * pSrcLPC,
    const Ipp16s * pSrcFltResidual, Ipp16s * pSrcDstSpch, Ipp16s *
    pDstImpulseResponse);
```

```
IppStatus ippsShortTermPostFilter_G729A_16s(const Ipp16s * pSrcLPC,
    const Ipp16s * pSrcFltResidual, Ipp16s * pSrcDstSpch);
```

引数

pSrcLPC	Q12 の量子化された LP 係数ベクトル [11] へのポインタ。
pSrcFltResidual	Q15 の残留信号 $x(n)$ ベクトル [40] へのポインタ。
pSrcDstSpch	Q15 の短期フィルタリングされた音声 $y(n)$ へのポインタ。要素 *pSrcDstSpch*[0...39] は現在の短期フィルタリングされた音声信号、要素 *pSrcDstSpch*[-10...-1] は使用されるヒストリである。
pDstImpulseResponse	Q12 の生成されたインパルス応答 $h_f(n)$ ベクトル [20] へのポインタ。

説明

これらの関数は、`ippsc.h` ファイルで宣言される。これらの 2 つの IIR フィルタは、残留信号から音声を復元する。いずれの関数もサブフレーム内で適用される。

1 番目の関数 `ippsShortTermPostFilter_G729` は、G.729/B コーデック向けに設計され、次の機能を持つ。

1. 最初に、逆フィルタ のインパルス応答 $h_f(n)$ を計算する。$\hat{A}(z)$

2. 次に、次の公式を使用して、ゲインの項 g_f を計算する。

$$g_f = \sum_{0}^{10} |h_f(n)|$$

3. 最後に、次のように、逆フィルタ $\hat{A}(z)$ によって残留信号をフィルタリングする。

$$y(n) = \frac{1}{g_f}x(n) - \sum_{i=1}^{10} \hat{a}_i y(n-i), n = 0,1,...,39 ,$$

ここで、$x(n)$ は残留信号、$\hat{a}_i$ は重み付けされた量子化 LP 係数である。

2 番目の関数 `ippsShortTermPostFilter_G729A` は、G.729A コーデック向けに設計されている。この関数は、次の公式を使用して、逆フィルタ $\hat{A}(z)$ によって残留信号をフィルタリングする。

$$y(n) = x(n) - \sum_{i=1}^{10} \hat{a}_i y(n-i), n = 0,1,...,39,$$

ここで、$x(n)$ は残留信号、$\hat{a}_i$ は重み付けされた量子化 LP 係数である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcLPC*、*pSrcFltResidual*、*pSrcDstSpch* または *pDstImpulseResponse* が NULL。

TiltCompensation_G729

短期フィルタ内のティルトを補正する。

```
IppStatus ippsTiltCompensation_G729_16s(const Ipp16s *
    pSrcImpulseResponse, Ipp16s * pSrcDstSpch);
IppStatus ippsTiltCompensation_G729A_16s(const Ipp16s * pSrcLPC, Ipp16s
    * pSrcDstFltResidual);
IppStatus ippsTiltCompensation_G729E_16s(Ipp16s val, const Ipp16s
    *pSrc, Ipp16s *pDst);
```

引数

pSrcImpulseResponse	Q12 のインパルス応答 $h_f(n)$ ベクトル [20] へのポインタ。
pSrcLPC	Q12 のガンマ係数で重み付けされた LP 係数ベクトル [22] へのポインタ。最初の 11 個の要素は γ_n、次の 11 個の要素は γ_d を示す。
pSrcDstSpch	Q15 の現在およびティルト補正された音声 $x(n)$ へのポインタ。要素 *pSrcDstSpch*[0...39] は現在の音声信号、要素 *pSrcDstSpch*[-1] は使用されるヒストリである。

pSrcDstFltResidual	Q15 の長期フィルタリングされた LP 残留信号へのポインタ。要素 *pSrcDstFltResidual*[0...39] は現在の長期フィルタリングされた LP 残留信号、要素 *pSrcDstFltResidual*[-1] は使用されるヒストリである。
val	入力ティルト係数。
pSrc	ソース・ベクトル [41] へのポインタ。
pDst	フィルタリングされた出力残留信号ベクトル [40] へのポインタ。

説明

これらの関数は、ippsc.h ファイルで宣言される。これらの FIR フィルタは、短期フィルタ内のティルトを補正する。これらの関数はサブフレームに適用される。

関数 ippsTiltCompensation_G729 は G.729/B コーデック向けに設計され、次の機能を持つ。

1. 最初に、インパルス応答 $h_f(i)$ の相関 $r_h(i)$ を計算する。

$$r_h(i) = \sum_{j=0}^{19-i} h_f(j)h_f(j+i)$$

次に、次の関係を使用して、ティルト係数 k_1' を求める。

$$k_1' = -\frac{r_h(1)}{r_h(0)}$$

2. 次に、次の公式を使用して、ゲインの項 g_t を求める。

$$g_t = 1 - |\gamma_t \cdot k_1'| \quad ,$$

ここで、k_1' が負の場合は $\gamma_t = 0.9$ 、k_1' が正の場合は $\gamma_t = 0.2$ である。

3. 最後に、次の公式を使用して、音声をフィルタリングする。

$$y(n) = \frac{1}{g_t}(x(n) + \gamma_t \cdot k_1' \cdot x(n-1)) \quad ,$$

ここで、$x(n)$ は入力音声信号である。

関数 ippsTiltCompensation_G729E は、入力ティルト係数 *val* に対してステージ 2 と 3 のみ実行し、*pSrc*[0] で指定したメモリをフィルタする。

関数 ippsTiltCompensation_G729A は、G.729A コーデック向けに設計され、次の機能を持つ。

1. 最初に、逆フィルタ $\hat{A}(z)$ のインパルス応答 $h_f(n)$ を計算する。

2. 次に、インパルス応答 $h_f(i)$ の相関 $r_h(i)$ を計算する。

$$r_h(i) = \sum_{j=0}^{21-i} h_f(j)h_f(j+i)$$

次に、次の関係を使用して、ティルト係数 k_1' を求める。

$$k_1' = -\frac{r_h(1)}{r_h(0)}$$

3. 最後に、次の公式を使用して、残留信号をフィルタリングする。

$$y(n) = x(n) + \gamma_t \cdot k_1' \cdot x(n-1) \quad ,$$

ここで、$x(n)$ は残留信号、k_1' が負の場合は $\gamma_t = 0.8$ 、k_1' が正の場合は $\gamma_t = 0$ である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDstSpch*、*pSrcImpulseResponse*、*pSrcLPC*、*pSrc*、*pDst*、または *pSrcDstFltResidual* が NULL。

HarmonicFilter

高調波フィルタを計算する。

```
IppStatus ippsHarmonicFilter_16s_I (Ipp16s beta, int T, Ipp16s*
    pSrcDst,
    int len);
IppStatus ippsHarmonicFilter_NR_16s (Ipp16s beta, int T, const Ipp16s *
    pSrc, Ipp16s * pDst, int len);
```

引数

beta	Q15 のベータ係数。
T	遅延の整数部分。
pSrc	ソース・ベクトルへのポインタ。要素 *pSrc*[-*T*,...,0,...*len*-1] が入力として使用される。
pDst	デスティネーション・ベクトルへのポインタ。
pSrcDst	ソースおよびデスティネーション・ベクトルへのポインタ。要素 *pSrcDst*[-*T*,...,0,...,*len*-1] が入力として使用され、要素 *pSrcDst*[0,...,*len*-1] が計算される。
len	ソースおよびデスティネーション・ベクトルの要素の数。

説明

これらの関数は、ippsc.h ファイルで宣言される。

関数 ippsHarmonicFilter_16s_I は、次のアダプティブ・プレフィルタを実行する。

$$H(z) = \frac{1}{1 - \beta \cdot z^{-T}} ,$$

このフィルタを使用して、音声の高調波成分を強化できる。

次のようにする。

$$pSrcDst[n] = pSrcDst[n] + beta \cdot pSrcDst[n-T] ,\ 0 \le n < len$$

関数 ippsHarmonicFilter_NR_16s は、次の高調波ノイズ・シェーピング・フィルタを実行する。

$$H(z) = 1 + \beta \cdot z^{-T}$$

次のように計算する。

$$pDst[n] = pSrc[n] + beta \cdot pSrc[n-T] ,\ 0 \le n < len$$

いずれのフィルタも、入力信号の高調波成分の強化に使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pSrcDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

intel®

HighPassFilterSize_G729

G729 ハイパス・フィルタのサイズを返す。

```
IppStatus ippsHighPassFilterSize_G729 (int *pSize);
```

引数

pSize	メモリ・サイズの出力値へのポインタ。

説明

関数 ippsHighPassFilterSize_G729 は、ippsc.h ファイルで宣言される。この関数は、ippsHighPassFilterInit_G729 関数の実行に必要なメモリのサイズを返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSize* ポインタが NULL。

HighPassFilterInit_G729

ハイパス・フィルタを初期化する。

```
IppStatus ippsHighPassFilterInit_G729 (Ipp16s *pCoeff,
    char*pMemUpdated);
```

引数

pCoeff	値 a_0, a_1, a_2, b_0, b_1, b_2, を格納する、Q12 または Q13 の長さ 6 の配列へのポインタ。
pMemUpdated	フィルタに割り当てられたメモリへのポインタ。

説明

関数 ippsHighPassFilterInit_G729 は、ippsc.h ファイルで宣言される。この関数は、ハイパス・フィルタの内部データを初期化する。a_0 は（Q12 または Q13 で）スケーリングされた 1 に等しくなければならない。フィルタのヒストリ・データ x_{-2}, x_{-1}, y_{-2}, y_{-1} はゼロに設定される。ippsHighPassFilter_G729_16s_ISfs 関数は、このメモリを使用してフィルタリングを実行する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCoeff* または *pMemUpdated* が NULL。

HighPassFilter_G729

G729 ハイパス・フィルタを実行する。

```
IppStatus ippsHighPassFilter_G729_16s_ISfs (Ipp16s* pSrcDst, int len,
    int scaleFactor, char* pMemUpdated);
```

引数

pSrcDst	ソースおよびデスティネーション・ベクトル [*len*] へのポインタ。
len	ソースおよびデスティネーション・ベクトルの要素の数。
scaleFactor	出力データのスケーリングに使用されるスケール係数。
pMemUpdated	フィルタに割り当てられたメモリへのポインタ。

ê‡ñæ

関数 ippsHighPassFilter_G729 は、ippsc.h ファイルで宣言される。この関数は、次のハイパス・フィルタを使用して、入力信号の前処理または出力信号の後処理を実行する。

$$H_h(z) = \frac{b_0 + b_1 \cdot z^{-1} + b_2 \cdot z^{-2}}{a_0 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2}}$$

現在のところ、a_0 = 1 （Q12 または Q13）だけが使用できる。この関数は、スケール係数の値として、入力データのプレフィルタリングには 12、出力データのポストフィルタリングには 13 を使用する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst* または *pMemUpdated* が NULL。
ippStsScaleRangeErr	エラー。*scaleFactor* が 12 または 13 でない。
ippStsSizeErr	エラー。*len* がゼロ以下。

IIR16s_G729

IIR フィルタリングを実行する。

```
IppStatus ippsIIR16sLow_G729_16s(const Ipp16s *pCoeffs, const Ipp16s
    *pSrc, Ipp16s *pDst, Ipp16s *pMem);
IppStatus ippsIIR16s_G729_16s(const Ipp16s *pCoeffs, const Ipp16s
    *pSrc, Ipp16s *pDst, Ipp16s *pMem);
```

引数

pCoeffs	フィルタ係数ベクトル [20] (b_1,…, b_{10}, a_1, …,a_{10}) の入力ベクトルへのポインタ。
pSrc	入力信号ベクトル [40] へのポインタ。
pDst	フィルタリングされた出力ベクトル [40] へのポインタ。
pMem	フィルタ・メモリ・ベクトル [20] へのポインタ。このベクトルは、最初はゼロで埋められていなければならない。

説明

関数 ippsIIR16sLow_G729s_16s と ippsIIR16s_G729_16s は、ippsc.h ファイルで宣言される。

これらの関数は、次の伝達関数を使用して、無限インパルス応答（IIR）フィルタリングを実行する。

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

関数 ippsIIR16sLow_G729_16s は ippsIIR16s_G729_16s とは異なり、オーバーフローと結果飽和をチェックしない。

フィルタ・メモリは更新される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCoeffs*、*pSrc*、*pSrcDst*、*pSrc*、*pDst* または *pMem* が NULL。

PhaseDispersionGetStateSize_G729D

フェーズ分散フィルタのメモリ・サイズをクエリする。

```
IppStatus ippsPhaseDispersionGetStateSize_G729D_16s (int *pSize);
```

引数

pSize	出力フェーズ分散フィルタのメモリ・サイズへのポインタ（バイト単位）。

説明

関数 ippsPhaseDispersionGetStateSize_G729D_16s は、フェーズ分散フィルタの操作で必要なメモリのサイズを提供する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSize* ポインタが NULL。

PhaseDispersionInit_G729D

フェーズ分散フィルタのメモリを
初期化する。

```
IppStatus ippsPhaseDispersionInit_G729D_16s
    (IppsPhaseDispersion16s_State_G729D* pPhDMem);
```

引数

pPhDMem	フェーズ分散フィルタのメモリへのポインタ。

説明

関数 ippsPhaseDispersionInit_G729D_16s は、バッファの内容をフェーズ分散フィルタのメモリの初期状態として設定する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pPhDMem* が NULL。

PhaseDispersionUpdate_G729D

フェーズ分散フィルタ・ステートを
更新する。

```
IppStatus ippsPhaseDispersionUpdate_G729D_16s(Ipp16s valPitchGain,
    Ipp16s valCodebookGain, IppsPhaseDispersion16s_State_G729D
    *pPhDMem);
```

引数

valPitchGain	長期ピッチ・ゲイン。
valCodebookGain	コードブック・ゲイン。
pPhDMem	フェーズ分散フィルタのメモリへのポインタ。

説明

関数 ippsPhaseDispersionUpdate_G729D_16s は、フェーズ分散フィルタ・メモリの状態を、任意のピッチ・ゲインおよびコードブック・ゲインで更新する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pPhDMem* が NULL。

PhaseDispersion_G729D

フェーズ分散フィルタリングを実行する。

```
IppStatus ippsPhaseDispersion_G729D_16s(Ipp16s valCodebookGain, Ipp16s
    valPitchGain,  const Ipp16s *pSrcExcSignal, Ipp16s
    *pDstFltExcSignal, Ipp16s *pSrcDstInnovation,
    IppsPhaseDispersion16s_State_G729D *pPhDMem)
```

引数

valPitchGain	入力長期ピッチ・ゲイン。
valCodebookGain	入力コードブック・ゲイン。
pSrcExcSignal	入力励振ベクトル [40] へのポインタ。
pDstFltExcSignal	フィルタリングされた出力励振ベクトル [40]。
pSrcDstInnovation	入力 / 出力改変コードブック・ベクトル [40]。
pPhDMem	フェーズ分散フィルタのメモリへのポインタ。

説明

関数 ippsPhaseDispersion_G729D_16s は、任意のピッチ・ゲインとコードブック・ゲインに対してフェーズ分散フィルタリングを実行する。フィルタは、改変信号（主にフェーズ）を変更することで、エネルギーがサブフレームに広がる新しい改変信号を作成する。フィルタリングは、異なる拡散量に基づいて 3 つの「セミランダム」なインパルス応答から 1 つを使用して、円状たたみ込みによって実行される。

戻り値

ippStsNoErr	エラーなし。

`ippStsNullPtrErr`	エラー。ポインタ *pSrcExcSignal*、*pDstFltExcSignal*、*pSrcDstInnovation*、または *pPhDMem* が NULL。

Preemphasize_G729A

ポスト・フィルタのプリエンファシスを計算する。

```
IppStatus ippsPreemphasize_G729A_16s (Ipp16s gamma, const Ipp16s *pSrc,
    Ipp16s *pDst, int len, Ipp16s* pMem);
IppStatus ippsPreemphasize_G729A_16s_I (Ipp16s gamma, Ipp16s* pSrcDst,
    int len, Ipp16s* pMem);
```

引数

gamma	Q15 のフィルタ係数。
pSrc	Q0 のソース・ベクトルへのポインタ。
pDst	Q0 のデスティネーション・ベクトルへのポインタ。
pSrcDst	Q0 のソースおよびデスティネーション・ベクトルへのポインタ。
len	ソースおよびデスティネーション・ベクトルの要素の数。
pMem	フィルタ・メモリ [1] へのポインタ。

説明

関数 `ippsPreemphasize_G729A` は、`ippsc.h` ファイルで宣言される。この関数は、ポスト・フィルタのプリエンファシスを計算する。

次の差分信号のプリエンファシスの式に従って、計算が実行される。

$$H(z) = 1 - gamma \cdot z^{-1}$$

フィルタ・メモリ *pMem* はゼロに初期化され、フィルタリング中に常に更新される。

関数 `ippsPreemphasize_G729A` は、丸め（クリッピング）を実行しない。

戻り値

`ippStsNoErr`	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、*pSrcDst* または *pMem* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

WinHybridGetStateSize_G729E

ハイブリッド窓モジュールのメモリ・サイズをクエリする。

```
IppStatus ippsWinHybridGetStateSize_G729E_16s (int *pSize);
```

引数

pSize	出力ハイブリッド窓モジュールのメモリ・サイズへのポインタ（バイト単位）。

説明

関数 ippsWinHybridGetStateSize_G729E は、ハイブリッド窓モジュールで必要なメモリのサイズを提供する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSize* ポインタが NULL。

WinHybridInit_G729E

ハイブリッド窓モジュールのメモリを初期化する。

```
IppStatus ippsWinHybridInit_G729E_16s (IppsWinHybridState_G729E_16s*
    pHybWinMem);
```

引数

pHybWinMem	ハイブリッド窓モジュールのメモリへのポインタ。

説明

関数 ippsPhaseDispersionInit_G729D_16s は、バッファの内容をハイブリッド窓モジュールのメモリの初期状態として設定する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pHybWinMem* が NULL。

WinHybrid_G729E

ハイブリッド窓を適用し、自己相関係数を計算する。

```
IppStatus ippsWinHybrid_G729E_16s32s (const Ipp16s* pSrcSynthSpeech,
    Ipp32s* pDstInvAutoCorr, IppsWinHybridState_G729E_16s*
    pHybWinMem);
```

引数

pSrcSynthSpeech	入力合成音声ベクトル [144] へのポインタ。
pDstInvAutoCorr	出力自己相関ベクトル [31] へのポインタ。
pHybWinMem	ハイブリッド窓モジュールのメモリへのポインタ。

説明

関数 ippsWinHybrid_G729E は、ハイブリッド窓を入力合成音声に適用し、自己相関係数を次のように計算する。

1. 最初に、入力音声ベクトルにハイブリッド窓が掛けられる。

$$s_{w+I}(k) = s_w(k) \cdot w(144 - k), k = 0,\ldots,144$$

ここで、

$$w(k) = \begin{cases} \sin((k+1) * 0.047783), k = 0,\ldots,34 \\ \sin(36 * 0.047783) * a^{(35-k)}, k = 35,\ldots,144 \end{cases}$$

$a = 0.9928337491$ とし、$a^{40}=0.75$ および $a^{160} = 0.75^4 = 0.31640625$ となる。

2. 次に、窓処理された音声の自己相関を次の式で計算する。

$$R_{rec}(n) = \sum_{k=0}^{k=79} s(k+30) \cdot s(k+30-n), n = 0,\ldots,30$$

$$R(n) = \sum_{k=0}^{k=34} s(k+110) \cdot s(k+110-n), n = 0,\ldots,30$$

3. 最後に、出力自己相関が計算され、メモリを次のように更新する。

$$\begin{cases} mem(k) = 0.31640625 * mem(k) + R_{rec}(k) \\ pDstAutoCorr(k) = mem(k) + R(k) \end{cases}, k = 0,\ldots,30$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcSynthSpeech*、*pDstInvAutoCorr*、または *pHybWinMem* が NULL。

RandomNoiseExcitation_G729B

ガウス分布を持つランダム・ベクトルを初期化する。

```
IppStatus ippsRandomNoiseExcitation_G729B_16s (Ipp16s *pSeed, Ipp16s
    *pExc, int len);
```

引数

pSeed	ランダム・ノイズ・ジェネレータの入力 / 出力シードへのポインタ。
pExc	デスティネーション・ベクトルへのポインタ。
len	デスティネーション・ベクトルの長さ。

説明

関数 `ippsRandomNoiseExcitation_G729B_16s` は、`ippsc.h` ファイルで宣言される。この関数は、次のようにランダム・ノイズの励振を発生させる。

$$pEcx[n] = \frac{\sum_{i=1}^{12} \xi_{(12n+i)}}{128}, n = 0,..., len-1$$

ここで、 $\xi_{k+1} = 31821 \cdot \xi_k + 13849 \quad \xi_0 = *pSeed$

ランダム・ノイズ・ジェネレータ ξ_k のシードは更新される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pExc* または *pSeed* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

G.723.1 に関連する関数

この項で説明するインテル® IPP 関数は、ITU-T 勧告 G.723.1（[ITU723]、[ITU723A] を参照）に準拠した音声コーデックの作成に使用できるビルディング・ブロックである。これらすべての関数のリストを次の表に示す。

表 9-3　G.723.1 に関連する IPP 関数

関数の基本名	操作
線形予測分析関数	
AutoCorr_G723	ベクトルの自己相関を推定する。
AutoCorr_NormE_G723	ベクトルの自己相関をノーマルで推定する。
LevinsonDurbin_G723	自己相関係数から LP 係数を計算する。
LPCToLSF_G723	LP 係数を LSF 係数に変換する。
LSFToLPC_G723	LSF 係数を LP 係数に変換する。
LSFDecode_G723	LSF の逆方向の量子化を実行する。
LSFQuant_G723	LSF 係数を量子化する。
コードブック検索関数	
OpenLoopPitchSearch_G723	最適なピッチ値を検索する。
ACELPFixedCodebookSearch_G723	励振の ACELP 固定コードブックを検索する。
AdaptiveCodebookSearch_G723	閉ループ・ピッチとアダプティブ・ゲイン・インデックスを検索する。
MPMLQFixedCodebookSearch_G723	励振のための MP-MLQ 固定コードブックを検索する。
ToeplizMatrix_G723	Calculates 固定コードブック検索用の Toepliz 行列の 416 個の要素を計算する。
ゲイン量子化関数	
GainQuant_G723	MP-MLQ ゲインの推定と量子化を実行する。
GainControl_G723	ディレイ・ピッチの影響を抽出する。

intel®

表 9-3　G.723.1 に関連する IPP 関数

関数の基本名	操作
フィルタ関数	
HighPassFilter_G723	入力信号のハイパス・フィルタリングを実行する。
IIR16s_G723	IIR フィルタリングを実行する。
SynthesisFilter_G723	合成フィルタ 1/A(z) によって入力音声をフィルタリングし、音声信号を計算する。
TiltCompensation_G723	ティルト補正フィルタを計算する。
HarmonicSearch_G723	高調波ノイズ・シェーピング・フィルタの高調波遅延と高調波ゲインを検索する。
HarmonicNoiseSubtract_G723	高調波ノイズ・シェーピングを実行する。
DecodeAdaptiveVector_G723	励振、ピッチ、アダプティブ・ゲインから、アダプティブ・コードブック・ベクトルを復元する。
PitchPostFilter_G723	ピッチ・ポスト・フィルタの係数を計算する。

線形予測分析関数

線形予測分析関数は、LPC 分析、LSP のコード化（量子化）とデコード、LPC 係数と LSF 係数の間の変換を実行する。

AutoCorr_G723

ベクトルの自己相関を推定する。

```
IppStatus ippsAutoCorr_G723_16s(const Ipp16s *pSrcSpch,  Ipp16s
    *pResultAutoCorrExp, Ipp16s *pDstAutoCorr);
```

引数

pSrcSpch	入力音声信号ベクトル [180] へのポインタ。
pResultAutoCorrExp	自己相関係数の指数へのポインタ。
pDstAutoCorr	自己相関係数ベクトル [11] へのポインタ。

説明

関数 ippsAutoCorr_G723 は、ippsc.h ファイルで宣言される。この関数は、入力音声信号の最初の 11 個の相関係数を計算する。この関数は、現在のサブフレームを中心とする 180 個の音声サンプルに適用される。関数の機能は次のとおりである。

1. 最初に、次の公式から得られる Hamming（ハミング）窓係数を音声サンプルに適用する。

$$w(i) = 0.54 - 0.46\cos\left[\frac{(2i-1)}{399}\pi\right], i = 0,1,...,179$$

2. 次に、自己相関を次のように計算する。

$$r(k) = \sum_{i=k}^{179} s(i) \times s(i-k), k = 0,1,...,10$$

3. 最後に、次の式から得られる二項窓係数を自己相関に適用する。

$$b(0) = 1025/1024$$

$$b(i) = \exp\left[-0.5\left(\frac{2\pi f_0 i}{f_s}\right)^2\right], i = 1,...,10$$

注： 関数 ippsAutoCorr_G723 は、実際には、ippsAutoScale 関数、ippsMul_NR 関数、ippsAutoCorr_NormE_G723 関数を組み合わせたものである。次のコードは、詳しい対応関係を示している。

```
{
      int autoScale=3, corrScale;
      short vect[180];
      ippsAutoScale_16s(pSrcSpch, Vect,180, &autoScale );
      /* Apply the Hamming window */
      ippsMul_NR_16s_ISfs(HammWindow,Vect,180,15);
      /* Compute the autocorrelation coefficients */
      ippsAutoCorr_NormE_G723_16s(Vect,pDstAutoCorr,&corrScale);
      *pResultAutoCorrExp = corrScale+(autoScale<<1);
}
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcSpch* または *pDstAutoCorr* が NULL。

AutoCorr_NormE_G723

ベクトルの自己相関をノーマルで推定する。

```
IppStatus ippsAutoCorr_NormE_G723_16s(const Ipp16s *pSrc, Ipp16s *pDst,
    int *pNorm);
```

引数

pSrc	ソース・ベクトル [180] へのポインタ。
pDst	ソース・ベクトルの推定された自己相関の結果を格納する、デスティネーション・ベクトルへのポインタ。
pNorm	出力される正規化スケール係数へのポインタ。

説明

関数 ippsAutoCorr_NormE_G723_16s は、ippsc.h ファイルで宣言される。この関数は、入力信号から 11 個の自己相関係数を計算する。最初の相関係数（エネルギー）に、ホワイト・ノイズ補正係数 $b(0) = 1025/1024$ が掛けられ、それ以外の 10 個の相関係数に、リファレンス C コード内で定義される二項窓係数 $b_n, n = 1,...,10$ が掛けられる（[ITU723] を参照）。

これらの自己相関係数に、さらに係数 2^{norm0} が掛けられる。ここで、$norm0 \geq 0$ は、最初の係数（エネルギー）が正規化されるように計算される。したがって、結果のベクトル *pDst* は、次のように計算される。

$$pDst[n] = b_n \cdot 2^{norm0} \cdot \sum_{i=0}^{179-n} pSrc[i] \cdot pSrc[i+n] \quad , 0 \leq n \leq 10$$

関数 ippsAutoCorr_NormE は、正規化スケール係数 *norm0* を *pNorm* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pNorm* が NULL。

LevinsonDurbin_G723

自己相関係数から LP 係数を計算する。

```
IppStatus ippsLevinsonDurbin_G723_16s(const Ipp16s * pSrcAutoCorr,
    Ipp16s *pValResultSineDtct, Ipp16s *pResultResidualEnergy, Ipp16s
    *pDstLPC);
```

引数

pSrcAutoCorr	自己相関係数ベクトル [11] へのポインタ。
pValResultSineDtct	正弦波検出器の入力 / 出力パラメータへのポインタ。
pResultResidualEnergy	Q15 の出力残留エネルギーへのポインタ。
pDstLPC	Q13 の出力 LP 係数ベクトル [10] へのポインタ。

説明

関数 `ippsLevinsonDurbin_G723` は、`ippsc.h` ファイルで宣言される。この関数は、Levinson-Durbin アルゴリズムを使用して、自己相関係数から 10 次 LP 係数を計算する。また、この関数は、正弦波の検出も実行する。

LP 係数 a_i, $i = 1, 2, \ldots, 10$ を求めるには、次の一連の方程式を解く必要がある。

$$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k)\ ,\ k = 1,2,\ldots,10$$

関数 `ippsLevinsonDurbin_G723` は、次の手順を実行する。

1. Levinson-Durbin アルゴリズムを適用して、上の一連の方程式を解く。このアルゴリズムは、`ippsLevinsonDurbin_G729` 関数で詳しく説明した漸化式を使用する。

2. このアルゴリズムに使用される LPC フィルタが不安定である（つまり、漸化式の計算中に が 1.0 に非常に近い値になる）場合は、それ以外の LP 係数をゼロに設定する。

3. 漸化式の計算中に $i = 1$ になったときは、正弦波検出器のパラメータが次のように更新される。

$SineDtct = SineDtct \ll 1$

$k > 0.95$ の場合は、$SineDtct = SineDtct + 1$ になる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcAutoCorr*、*pDstLPC*、*pValResultSineDtct* または *pResultResidualEnergy* が NULL。

LPCToLSF_G723

LP 係数を LSF 係数に変換する。

```
IppStatus ippsLPCToLSF_G723_16s (const Ipp16s *pSrcLPC, const Ipp16s
    *pSrcPrevLSF, Ipp16s *pDstLSF);
```

引数

pSrcLPC	Q13 の LPC 入力ベクトル [10] (*a1*,…,a*10*) へのポインタ。
pSrcPrevLSF	Q15 の直前の正規化された LSF 係数ベクトル [10] へのポインタ。
pDstLSF	Q15 の正規化された LSF 係数ベクトル [10] へのポインタ。

説明

関数 ippsLPCToLSF_G723 は、ippsc.h ファイルで宣言される。この関数は、次の演算を実行して、一連の 10 次 LP 係数を LSF 係数に変換する。

1. 7.5Hz 帯域幅の拡張を適用する。

2. 次の漸化式を使用して、$F_1(z)$ と $F_2(z)$ の多項式係数を計算する。

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i)$$

$i = 0...4$,

ここで、$f_1(0) = f_2(0) = 1.0$

3. 次の Chebyshev 多項式を使用して $F_1(z)$ と $F_2(z)$ の根を求め、LSF 係数を得る。

$$c_1(\omega) = \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + f_1(5)/2$$

$$c_2(\omega) = \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + f_2(5)/2$$

3. LSF 係数を求めるのに必要な 10 個の根が見つからない場合は、直前の一連の LSF 係数を使用する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcLPC*、*pSrcPrevLSF* または *pDstLSF* が NULL。

LSFToLPC_G723

LSF 係数を LP 係数に変換する。

```
IppStatus ippsLSFToLPC_G723_16s(const Ipp16s *pSrcLSF, Ipp16s *pDstLPC);
IppStatus ippsLSFToLPC_G723_16s_I (Ipp16s *pSrcLSFDstLPC);
```

引数

pSrcLSF	Q15 の入力 LSF 係数ベクトル [10] へのポインタ。
pDstLPC	Q13 の出力 LP 係数ベクトル [10] へのポインタ。
pSrcLSFDstLPC	Q15 の入力 LSF 係数および Q13 の出力 LP 係数ベクトル [10] へのポインタ。

説明

関数 ippsLSFToLPC_G723 は、ippsc.h ファイルで宣言される。この関数は、次の手順に従って、一連の 10 次 LSF 係数を LP 係数に変換する。

1. LSF 係数を LSP 係数に変換する。
$q_i = \cos(\omega_i)$, $i = 0...9$

2. 次の漸化式を使用して、$F_1(z)$ と $F_2(z)$ の多項式係数を計算する。

$f_1(0) = f_1(-1) = 0$

for i = 1 to 5

$f_1(i) = -2q_{2i-1} * f_1(i-1) + 2f_1(i-2)$

$f_2(i) = -2q_{2i} * f_2(i-1) + 2f_2(i-2)$

for j = i-1 down to 1

$$f_1^{[i]}(j) = f_1^{[i-1]}(j) - 2q_{2i-1} f_1^{[i-1]}(j-1) + f_1^{[i-1]}(j-2)$$

$$f_2^{[i]}(j) = f_2^{[i-1]}(j) - 2q_{2i} f_2^{[i-1]}(j-1) + f_2^{[i-1]}(j-2)$$

ここで、$f_1(0) = f_2(0) = 1.0$

3. 多項式 $F_1(z)$ に $1 + z^{-1}$ を掛けて、$F_1'(z)$ を得る。また、多項式 $F_2(z)$ に $1 - z^{-1}$ を掛けて、を得る。$F_2'(z)$

4. 次の公式を使用して、多項式 $F_1'(z)$ と $F_2'(z)$ から LP 係数を計算する。

$$\alpha_{i = 1 \ldots 5} = 0.5 f_1'(i) + 0.5 f_2'(i)$$

$$\alpha_{i = 6 \ldots 10} = 0.5 f_1'(11 - i) - 0.5 f_2'(11 - i)$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcLSF*、*pDstLPC* または *pSrcLSFDstLPC* が NULL。

LSFDecode_G723

LSF の逆方向の量子化を実行する。

```
IppStatus ippsLSFDecode_G723_16s (const Ipp16s *quantIndex, const
    Ipp16s *pPrevLSF, int erase, Ipp16s *pQuantLSF);
```

引数

quantIndex	入力 LSP VQ インデックス・ベクトル [3] へのポインタ。
pPrevLSF	直前のサブフレームの入力量子化 LSF へのポインタ。
erase	フレームの消去を示す。
pQuantLSF	直前のサブフレームの出力量子化 LSF へのポインタ。

説明

関数 `ippsLSFDecode_G723_16s` は、`ippsc.h` ファイルで宣言される。この関数は、LSF の逆方向の量子化を実行する（つまり、LSP VQ インデックスをデコードする）。最初に、送信されるインデックスに対応する 3 つの VQ テーブル・エントリが

検出される。検出されたベクトルと DC ベクトルに、予想されるベクトルが追加され、3 つの周波数帯域にデコードされたベクトルが別々に作成される。安定性チェックが実行され、それぞれの差が 31.25 Hz 以上であることが保証される。

消去されたフレームについては、ゼロとして予測されたベクトルが選択され、62.5 Hz の差を条件とする安定性チェックが適用される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *quantIndex*、*pPrevLSF* または *pQuantLSF* が NULL。
`ippStsLSFLow`	警告。安定性の条件を満たしていない。

LSFQuant_G723

LSF 係数を量子化する。

```
IppStatus ippsLSFQuant_G723_16s32s(const Ipp16s* pSrcLSF, const Ipp16s*
    pSrcPrevLSF, Ipp32s* pResultQLSFIndex);
```

引数

pSrcLSF	Q15 の LSF 係数ベクトル [10] へのポインタ。
pSrcPrevLSF	Q15 の直前の LSF 係数ベクトル [10] へのポインタ。
pResultQLSFIndex	量子化された LSF 係数の複合インデックスへのポインタ。複合インデックスを作成するには、最初のコードブック・インデックスを 16 ビット左にシフトして、2 番目のコードブック・インデックスを 8 ビット左にシフトし、シフトされた 2 つのインデックスと 3 番目のコードブック・インデックスを合計する。

説明

関数 `ippsLSFQuant_G723` は、`ippsc.h` ファイルで宣言される。この関数は、PSVQ を使用して、LSF 係数を量子化し、コードブック・インデックスを得る。この関数は、次の演算を実行する。

1. 量子化されていない LSF 係数から得られる、対角重み付け行列 *w* を計算する。

$$w_{1,1} = 1/(\omega_2 - \omega_1)$$

$$w_{10,10} = 1/(\omega_{10} - \omega_9)$$

$$w_{j,j} = 1/\min(\omega_j - \omega_{j-1},\ \omega_{j+1} - \omega_j),\ j = 2...9$$

2. 次の式から得られる予想 LSF 行列を計算する。

$$\omega_p = (\omega - \omega_{DC}) - 0.375(\omega_{-1} - \omega_{DC})$$

ここで、ω は現在の LSF 係数、ω_{-1} は直前の LSF 係数、ω_{DC} は DC LSF 係数である。

3. 次の誤差が最小になるようなコードブック・ベクトルを検索する。

$$E_l = (\omega_p - \omega_l)^T W (\omega_p - \omega_l)、$$

ここで、ω_l はコードブック内の l 番目のコード・ベクトルである。量子化には 3-3-4 の分割が使用される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcLSF*、*pSrcPrevLSF* または *pResultQLSFIndex* が NULL。

コードブック検索関数

コードブック検索関数は、アダプティブ・コードブックを使用してオープン・ループ・ピッチの推定を実行し、ACELP 励振（固定）コードブック内でパルスの最適な符号と位置を検索する。

OpenLoopPitchSearch_G723

最適なピッチ値を検索する。

```
IppStatus ippsOpenLoopPitchSearch_G723_16s(const Ipp16s * pSrcWgtSpch,
    Ipp16s * pResultOpenDelay);
```

引数

pSrcWgtSpch	Q12 の知覚的に重み付けされた音声信号へのポインタ。信号の長さは 265 で、ポインタ *pSrcWgtSpch* は 146 番目の要素を指す。
pResultOpenDelay	オープン・ループ・ピッチの検索結果へのポインタ。

説明

関数 `ippsOpenLoopPitchSearch_G723` は、`ippsc.h` ファイルで宣言される。この関数は、重み付けされた音声信号からオープン・ループ・ピッチを抽出する。この関数は、次のようにハーフフレーム内で適用される。

1. 相互相関が最大になるようなオープン・ループ・ピッチが選択される。

$$C_{ol}(j) = \frac{\left[\sum_{i=0}^{119} s(i)s(i-j)\right]^2}{\sum_{i=0}^{119} s(i-j)s(i-j)}, \quad j = 18,...,142$$

2. 検索中に、ピッチの倍数が選択されるのを避けるために、小さめのピッチ周期が使用される。見つかった各ピッチ値は、それまでの最良の値と比較される。その差が 18 より小さく、$C_{ol}(j) > C_{ol}(j')$ の場合は、プロセスは終了する。それ以外の場合は、$C_{ol}(j)$ が $C_{ol}(j')$ より 1.25db 大きい場合にのみ、そのピッチが選択される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcWgtSpch* または *pResultOpenDelay* が NULL。

ACELPFixedCodebookSearch_G723

励振の ACELP 固定コードブックを
検索する。

```
IppStatus ippsACELPFixedCodebookSearch_G723_16s(const Ipp16s
    *pSrcFixedCorr, const Ipp16s *pSrcMatrix, Ipp16s *pDstFixedSign,
    Ipp16s *pDstFixedPosition, Ipp16s *pResultGrid, Ipp16s
    *pDstFixedVector, Ipp16s *pSearchTimes);
IppStatus ippsACELPFixedCodebookSearch_G723_32s16s(const Ipp16s
    *pSrcFixedCorr, Ipp32s *pSrcDstMatrix, Ipp16s *pDstFixedSign,
    Ipp16s *pDstFixedPosition, Ipp16s *pResultGrid, Ipp16s
    *pDstFixedVector, Ipp16s *pSearchTimes);
```

引数

pSrcFixedCorr	残留信号とインパルス応答ベクトル [60] の間の相関へのポインタ。
pSrcMatrix	Toepliz 行列 [416] の要素へのポインタ。
pSrcDstMatrix	Toepliz 行列 [416] の入力および出力要素へのポインタ。
pDstFixedSign	固定ベクトル [4] の符号へのポインタ。
pDstFixedPosition	固定ベクトル [4] の位置へのポインタ。
pResultGrid	開始グリッド位置へのポインタ。
pDstFixedVector	固定ベクトル [60] の位置へのポインタ。
pSearchTimes	入力および出力最大検索時間へのポインタ。

説明

関数 `ippsACELPFixedCodebookSearch_G723` は、`ippsc.h` ファイルで宣言される。この関数は、5.3Kbps エンコーダ内の励振の ACELP 固定コードブックを検索する。この関数はサブフレーム内で次のように適用される。

1. 次の式から得られる、4 つのゼロでないパルスが固定ベクトル内に存在する。

$$c(n) = \sum_{k=0}^{3} \alpha_k \delta(n - m_k) \quad , \; n = 0,...,59$$

ここで、α_k , $k = 0...3$ は固定ベクトルの符号、m_k , $k = 0...3$ は固定ベクトルの位置である。

2. 次の誤差が最小になるような、パラメータ $\alpha_k, k = 0...3$ と $m_k, k = 0...3$ を検索する。

$$\delta = \Phi(m_0, m_0) + \Phi(m_1, m_1) + 2\alpha_0\alpha_1\Phi(m_0, m_1) +$$
$$\Phi(m_2, m_2) + 2[\alpha_0\alpha_2\Phi(m_0, m_2) + \alpha_1\alpha_2\Phi(m_1, m_2)] +$$
$$\Phi(m_3, m_3) + 2[\alpha_0\alpha_3\Phi(m_0, m_3) + \alpha_1\alpha_3\Phi(m_1, m_3) + \alpha_2\alpha_3\Phi(m_2, m_3)]$$

ここで、$\Phi(i, j)$ は Toepliz 行列である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcFixedCorr*、*pSrcMatrix*、*pSrcDstMatrix*、*pSearchTimes*、*pDstFixedSign*、*PDstFixedPosition*、*pResultGrid* または *pDstFixedVector* が NULL。

AdaptiveCodebookSearch_G723

閉ループ・ピッチとアダプティブ・ゲイン・インデックスを検索する。

```
IppStatus ippsAdaptiveCodebookSearch_G723(Ipp16s valBaseDelay,
    const Ipp16s * pSrcAdptTarget, const Ipp16s * pSrcImpulseResponse,
    Ipp16s * pSrcPrevExcitation, const Ipp32s *pSrcPrevError,
    Ipp16s * pResultCloseLag, Ipp16s * pResultAdptGainIndex, Ipp16s
    subFrame, Ipp16s sineDtct, IppSpchBitRate bitRate);
```

引数

valBaseDelay	[18,145] の範囲のベース遅延。
pSrcAdptTarget	アダプティブ・ターゲット信号ベクトル [60] へのポインタ。
pSrcImpulseResponse	インパルス応答ベクトル [60] へのポインタ。
pSrcPrevExcitation	直前の励振ベクトル [145] へのポインタ。
pSrcPrevError	直前の誤差ベクトル [5] へのポインタ。
subFrame	サブフレーム番号、0 ～ 3。

bitRate	送信ビット・レート、IPP_SPCHBR_6300 または IPP_SPCHBR_5300 に等しい。
sineDtct	正弦波検出器のパラメータ。
pResultCloseLag	閉ループ・ピッチのラグへのポインタ。
pResultAdptGainIndex	アダプティブ・ゲインのインデックスへのポインタ。

説明

関数 ippsAdaptiveCodebookSearch_G723 は、ippsc.h ファイルで宣言される。この関数は、閉ループ・ピッチとアダプティブ・ゲイン・インデックスを検索する。この関数はサブフレーム内で次のように適用される。

1. サブフレーム 0 と 2 では、適切なオープン・ループ・ピッチの周囲の [-1,1] の範囲内で閉ループ・ピッチのラグが選択される。サブフレーム 1 と 3 では、オープン・ループ・ピッチの周囲の [-1, 3] の範囲内で閉ループ・ピッチのラグが選択される。

2. 閉ループ・ピッチを L_i, i =0...3 として示す。ピッチ予測子ゲインはベクトル量子化される。6.3Kbps のビット・レートでは、85 エントリと 170 エントリの 2 つのコードブックが使用される。サブフレーム 0 と 1 で L_0 が 58 より小さいか、サブフレーム 2 と 3 で L_2 が 58 より小さい場合は、85 エントリのコードブックがピッチ・ゲインの量子化に使用される。それ以外の場合は、170 エントリのコードブックが使用される。5.3Kbps のビット・レートでは、170 エントリのコードブックが量子化に使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcAdptTarget*、*pSrcImpulseResponse*、*pSrcPrevExcitation*、*pSrcPrevError*、*pResultCloseLag* または *pResultAdptGainIndex* が NULL。
ippStsRangeErr	エラー。*valBaseDelay* が [18, 145] の範囲内にないか、サブフレームが [0, 3] の範囲内にないか、*bitRate* が列挙型 IppSpchBitRate の有効な要素でない。

MPMLQFixedCodebookSearch_G723

励振の MP-MLQ 固定コードブックを
検索する。

```
IppStatus ippsMPMLQFixedCodebookSearch_G723(Ipp16s valBaseDelay, const
    Ipp16s *pSrcImpulseResponse, const Ipp16s *pSrcResidualTarget,
    Ipp16s *pDstFixedVector, Ipp16s *pResultGrid, Ipp16s
    *pResultTrainDirac, Ipp16s *pResultAmpIndex, Ipp16s
    *pResultAmplitude, Ipp32s *pResultPosition, Ipp16s subFrame);
```

引数

valBaseDelay	[18,145] の範囲のベース遅延。
pSrcImpulseResponse	インパルス応答ベクトル [60] へのポインタ。
pSrcResidualTarget	残留ターゲット信号ベクトル [60] へのポインタ。
pDstFixedVector	固定コードブック・ベクトル [60] へのポインタ。
pResultGrid	開始グリッド位置（0 または 1 ）へのポインタ。
pResultTrainDirac	トレイン Dirac 関数を使用するかどうかを示すフラグへのポインタ。0- 使用しない、1- 使用する。
pResultAmpIndex	量子化された大きさのインデックスへのポインタ。
pResultAmplitude	量子化されたコードブック・ベクトルの大きさへのポインタ。
pResultPosition	大きさがゼロでない固定コードブック・ベクトルの位置へのポインタ。
subFrame	サブフレーム番号、0 〜 3。

説明

関数 `ippsMPMLQFixedCodebookSearch_G723` は、`ippsc.h` ファイルで宣言される。この関数は、6.3Kbps エンコーダ内の励振の MP-MLQ 固定コードブックを検索する。この関数はサブフレーム内で次のように適用される。

1. 次の計算によって誤差を求める。

$$err(n) = res(n) - G\sum_{k=0}^{M-1}\alpha_k h(n-m_k)$$

ここで、G はゲイン係数、α_k は固定ベクトルの符号、m_k は固定ベクトルの位置である。

2. 誤差信号 $err(n)$ の平均 2 乗が最小になるような、パラメータ G、α_k、m_k を検索する。

3. 次の公式を使用して、固定コードブック・ゲインを求める。

$$d(j) = \sum_{n=j}^{59} c(n) \cdot h(n-j)$$

$$Gm = \frac{\max\{|d(j)|\}_{j=0\ldots59}}{\sum_{n=0}^{59} h(n) \cdot h(n)},$$

ここで、$c(n)$ は固定ベクトルである。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcImpulseResponse*、*pSrcResidualTarget*、*pDstFixedVector*、*pResultGrid*、*pResultTrainDirac*、*pResultAmpIndex*、*pResultAmplitude* または *pResultPosition* が NULL。
IppStsRangeErr	エラー。*SubFrame* が 0、1、2、または 3 のいずれでもないか、*valBaseDelay* が [18,145] の範囲内にない。

ToeplizMatrix_G723

固定コードブック検索用の Toepliz 行列の
416 個の要素を計算する。

```
IppStatus ippsToeplizMatrix_G723_16s(const Ipp16s *
    pSrcImpulseResponse, Ipp16s * pDstMatrix);
IppStatus ippsToeplizMatrix_G723_16s32s(const Ipp16s *
    pSrcImpulseResponse, Ipp32s * pDstMatrix);
```

引数

pSrcImpulseResponse	インパルス応答ベクトル [60] へのポインタ。
pDstMatrix	Toepliz 行列ベクトル [416] の要素へのポインタ。

説明

関数 `ippsToeplizMatrix_G723` は、`ippsc.h` ファイルで宣言される。この関数は、固定コードブック検索用の Toepliz 行列の 416 個の要素を計算する。Toepliz 行列の要素は、次のように表現される。

$$\Phi(i, j) = \sum_{n=j}^{59} h(n-i) \times h(n-j) \ , i \le j \ , 0 \le i \le 59 \ ,$$

ここで $h(i)$, $i = 0, 1, \ldots, 59$, はインパルス応答である。

この関数は、計算された 416 個の要素を次の順序で *pDstMatrix* に格納する。

1. $\Phi(m_i, m_i)$ 、(i = 0, 1, 2, 3)、4 × 8=32 個の要素、開始位置 : 0
2. $\Phi(m_0, m_1)$ 、8 × 8= 64 個の要素、開始位置 : 32
3. $\Phi(m_0, m_2)$ 、8 × 8= 64 個の要素、開始位置 : 96
4. $\Phi(m_0, m_3)$ 、8 × 8= 64 個の要素、開始位置 : 160
5. $\Phi(m_1, m_2)$ 、8 × 8= 64 個の要素、開始位置 : 224
6. $\Phi(m_1, m_3)$ 、8 × 8= 64 個の要素、開始位置 : 288
7. $\Phi(m_2, m_3)$ 、8 × 8= 64 個の要素、開始位置 : 352

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcImpulseResponse* または *pDstMatrix* が NULL。

ゲイン量子化

ゲイン量子化関数は、固定コードブック・ゲインの推定と、ポストフィルタリングされた信号のゲインの調整を実行する。

GainQuant_G723

MP-MLQ ゲインの推定と量子化を
実行する。

```
IppStatus ippsGainQuant_G723_16s (const Ipp16s *pImp, const Ipp16s
    *pSrc,       Ipp16s *pDstLoc, Ipp16s *pDstAmp, Ipp32s *pMaxErr,
    Ipp16s *pGrid, Ipp16s *pAmp, int Np, int* isBest);
```

引数

pImp	入力インパルス応答 *h* ベクトル [60] へのポインタ。
pSrc	入力ターゲット信号 *r* ベクトル [60] へのポインタ。
pDstLoc	出力パルスの位置へのポインタ。
pDstAmp	出力パルスの大きさへのポインタ。
pMaxErr	入力 / 出力量子化誤差へのポインタ。
pGrid	出力グリッドへのポインタ。 0 - パルスが偶数の位置にある場合 1 - パルスが奇数の位置にある場合
pAmp	最大の大きさの出力インデックスへのポインタ。
Np	パルスの数。 偶数サブフレームでは 6 奇数サブフレームでは 5
isBest	量子化の結果を示す。量子化誤差が入力誤差より良い (小さい) パルスが見つからない場合は、ゼロに設定される。

説明

関数 `ippsGainQuant_G723_16s` は、`ippsc.h` ファイルで宣言される。この関数は、誤差信号の平均 2 乗が最小になるような、未知のパラメータ G、$\{\alpha_k\}$、$\{m_k\}$、$k = 0,\ldots,Np-1$ を推定する。

$$err[n] = r[n] - G \cdot \sum_{k=0}^{Np-1} a_k \cdot h[n-m_k]$$

推定は、次のように行われる。

最初に、インパルス応答とターゲット・ベクトルの相互相関が計算される。

$$d[j] = \sum_{n=j}^{59} r[n] \cdot h[n-j] \quad , 0 \le n \le 59$$

最大ゲインの推定値が計算され、

$$G_{max} = \frac{\max\{|d[j]|\}_{j=0\ldots59}}{\sum_{n=0}^{59} h[n] \cdot h[n]}$$

対数的量子化機構によって量子化される。次に、この量子化されたゲインの周囲で選択されたゲインの値を使用して、誤差信号の平均 2 乗が最小になるように、偶数グリッドと奇数グリッドの両方についてパルスの符号と位置を最適化する。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pImp*、*pSrc*、*pDstLoc*、*pDstAmp*、*pMaxErr*、*pGrid*、*pAmp* または *isBest* が NULL。

GainControl_G723

ディレイ・ピッチの影響を抽出する。

```
IppStatus ippsGainControl_G723_16s_I (Ipp32s energy, Ipp16s *pSrcDst,
    Ipp16s *pGain);
```

引数

energy	入力エネルギー係数。
pSrcDst	ポストフィルタリングされた入力 / 出力信号へのポインタ。
pGain	入力 / 出力ゲインへのポインタ。

説明

関数 ippsGainControl_G723_16s_I は、ippsc.h ファイルで宣言される。この関数は、最初に、大きさの比 g_s を計算する。

$$g_s = \sqrt{\frac{energy}{\sum_{n=0}^{59} pSrcDst[n] \cdot pSrcDst[n]}}$$

この式の分母がゼロになる場合は、g_s は 1 に設定される。

次に、ポストフィルタリングされた入力信号が次のようにスケーリングされる。

$$pSrcDst[n] = pSrcDst[n] \cdot g_n \cdot (1 + \alpha) \quad , \quad n = 0,\ldots, 59,$$

ここで、g_n はそれぞれ次の式を使用して更新される。

$$g_n = (1 - \alpha) \cdot g_{n-1} + \alpha \cdot g_s \quad , n = 0,\ldots, 59, \quad g_{-1} = 0$$

α は 1/16 である。

出力ゲインは g_{59} に等しくなる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst* または *pGain* が NULL。

フィルタ関数

フィルタ関数は、エンコードの前処理段階およびデコードの後処理段階のハイパス・フィルタリングや、最終的なデコード段階のポストフィルタリングなど、各種のフィルタリングを実行する。
各種の合成フィルタ（IIR）、高調波フィルタ、プリエンファシス・フィルタ関数を組み合わせて、さらに複雑なフィルタリングを実行できる。これらのフィルタ関数は、異なるエンコード段階とデコード段階で使用できる。

HighPassFilter_G723

入力信号のハイパス・フィルタリングを
実行する。

```
IppStatus ippsHighPassFilter_G723_16s (const Ipp16s* pSrc, Ipp16s*
    pDst, int* pMem);
```

引数

pSrc	ソース・ベクトル [240] へのポインタ。
pDst	デスティネーション・ベクトル [240] へのポインタ。
pMem	フィルタ・メモリ・ベクトル [2] へのポインタ。このベクトルは、最初はゼロで埋められていなければならない。

説明

関数 ippsHighPassFilter_G723 は、ippsc.h ファイルで宣言される。この関数は、次のフィルタ伝達関数を使用して、入力信号の前処理を行う。

$$H_h(z) = \frac{1 - z^{-1}}{1 - \frac{127}{128}z^{-1}}$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst* または *pMem* が NULL。

IIR16s_G723

IIR フィルタリングを実行する。

```
IppStatus ippsIIR16s_G723_16s32s (const Ipp16s *pCoeffs, const Ipp16s
    *pSrc, Ipp32s *pDst, Ipp16s *pMem);
IppStatus ippsIIR16s_G723_16s_I (const Ipp16s *pCoeffs, Ipp16s *pSrcDst,
    Ipp16s *pMem);
```

```
IppStatus ippsIIR16s_G723_32s16s_Sfs (const Ipp16s *pCoeffs, const
    Ipp32s *pSrc, int scaleFactor, Ipp16s *pDst, Ipp16s *pMem);
```

引数

pCoeffs	フィルタ係数ベクトル [20] （$b_1,\ldots, b_{10}, a_1, \ldots,a_{10}$）の入力ベクトルへのポインタ。
pSrc	入力信号ベクトル [60] へのポインタ。
pSrcDst	入力 / 出力信号ベクトル [60] へのポインタ。
scaleFactor	スケール係数
pDst	フィルタリングされた出力ベクトル [60] へのポインタ。
pMem	フィルタ・メモリ・ベクトル [20] へのポインタ。このベクトルは、最初はゼロで埋められていなければならない。

説明

これらの関数は、`ippsc.h` ファイルで宣言される。

関数 `ippsIIR16s_G723_16s32s` と `ippsIIR16s_G723_16s_I` は、次の伝達関数を使用して、無限インパルス応答（IIR）フィルタリングを実行する。

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

関数 `ippsIIR16s_G723_32s16s_Sfs` は、次の伝達関数を使用して、IIR フィルタリングを実行する。

$$H(z) = 2^{sFs} \cdot \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

フィルタ・メモリは更新される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCoeffs*、*pSrc*、*pSrcDst*、*pDst* または *pMem* が NULL。

SynthesisFilter_G723

合成フィルタ 1/A(z) によって入力音声をフィルタリングし、音声信号を計算する。

```
IppStatus ippsSynthesisFilter_G723_16s32s (const Ipp16s* pLPC, const
    Ipp16s* pSrc, Ipp32s* pDst, Ipp16s* pMem);
IppStatus ippsSynthesisFilter_G723_16s (const Ipp16s *pLPC, const
    Ipp16s * pSrc, Ipp16s * pMem, Ipp16s *pDst);
```

引数

pLPC	Q11 の入力 LP 係数 $a_0, a_1,\ldots,a_{10}$ へのポインタ。
pSrc	ソース・ベクトルへのポインタ。
pDst	フィルタリングされた出力へのポインタ。
pMem	フィルタリングに使用されるメモリへのポインタ。short 型整数ベクトル [10] で、最初はゼロに設定される。

説明

関数 ippsSynthesisFilter_G723 は、ippsc.h ファイルで宣言される。この関数は、次の式に従ってフィルタを計算する。

$$H(z) = \hat{A}^{-1}(z) = \frac{1}{a_0 + \sum_{i=1}^{10} a_i \cdot \hat{z}^{-i}}$$

この関数は、知覚的に重み付けされた音声信号を計算する場合に、残留信号フィルタの後に適用される。

$$pDst[n] = pSrc[n] \cdot pLPC[0] + \sum_{i=1}^{10} pLPC[i] \cdot pMem[n-i+1] \quad , n = 0,\ldots,59$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pLPC*、*pDst* または *pMem* が NULL。
ippStsOverflow	警告。少なくとも 1 つの結果の値が飽和された。

TiltCompensation_G723

ティルト補正フィルタを計算する。

```
IppStatus ippsTiltCompensation_G723_32s16s (Ipp16s val, const Ipp32s*
    pSrc, Ipp16s *pDst);
```

引数

val	Q15 の 1 次部分相関係数。
pSrc	ソース・ベクトル [61] へのポインタ。
pDst	フィルタリングされた出力ベクトル [60] へのポインタ。

説明

関数 ippsTiltCompensation_G723 は、ippsc.h ファイルで宣言される。この関数は、ティルト補正フィルタを次のように計算する。

$$H_t(z) = (1 - val \cdot z^{-1})$$

$$pDst[i] = pSrc[i+1] + pSrc[i] \cdot val \text{ , } i = 0,...,59$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。

HarmonicSearch_G723

高調波ノイズ・シェーピング・フィルタの
高調波遅延と高調波ゲインを検索する。

```
IppStatus ippsHarmonicSearch_G723_16s(Ipp16s valOpenDelay, const Ipp16s
    *pSrcWgtSpch, Ipp16s *pResultHarmonicDelay, Ipp16s
    *pResultHarmonicGain);
```

引数

valOpenDelay	[18,145] の範囲のオープン・ループ・ピッチ。
pSrcWgtSpch	Q12 の重み付けされた音声ベクトル [205] へのポインタ。このポインタは 146 番目の要素を指す。
pResultHarmonicDelay	出力高調波遅延へのポインタ。
pResultHarmonicGain	Q15 の出力高調波ゲインへのポインタ。

説明

関数 `ippsHarmonicSearch_G723` は、`ippsc.h` ファイルで宣言される。この関数は、重み付けされた音声とオープン・ループ・ピッチを入力に使用して、高調波ノイズ・シェーピング・フィルタの高調波遅延と高調波ゲインを検索する。
この関数は、サブフレーム内で次のように適用される。

1. 次の式が最適化されるようなインデックス L を見つける。

$$C_{pw}(j) = \left[\sum_{i=0}^{59} s(i)s(i-j)\right]^2 \Big/ \left(\sum_{i=0}^{59} s(i-j)s(i-j)\right) \quad 、\quad j = pitch-3 \ldots pitch+3$$

2. 最適なフィルタ・ゲインを次のように計算する。

$$G_{opt} = \sum_{i=0}^{59} s(i)s(i-j) / \sum_{i=0}^{59} s(i-j)s(i-j)$$

G_{opt} は [0,1] の範囲で飽和される。

3. 重み付けされた音声サンプルのエネルギーと高調波ゲインを計算する。

intel®

$$E = \sum_{i=0}^{59} s^2(i)$$

$$\beta = \begin{cases} 0.3125 G_{opt}, & if\ -10\log 10(1 - C_L/E) \geq 2.0 \\ 0, & otherwise \end{cases}$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcWgtSpch*、*pResultHarmonicDelay* または *pResultHarmonicGain* が NULL。
ippStsRangeErr	エラー。*valOpenDelay* が [18, 145] の範囲内にない。

HarmonicNoiseSubtract_G723

高調波ノイズ・シェーピングを実行する。

```
IppStatus ippsHarmonicNoiseSubtract_G723_16s_I(Ipp16s val, int T, const
    Ipp16s *pSrc, Ipp16s *pSrcDst);
```

引数

val	Q15 の入力高調波フィルタ係数。
T	入力高調波フィルタのラグ。
pSrc	複合フィルタの入力ゼロ・インパルス応答ベクトル [60] へのポインタ。
pSrcDst	高調波ノイズで重み付けされた入力 / 出力音声ベクトル [60] へのポインタ。

説明

関数 `ippsHarmonicNoiseSubtract_G723_16s_I` は、`ippsc.h` ファイルで宣言される。この関数は、次のように、ベクトル *pSrcDst* から高調波シェーピングされたベクトル *pSrc* を引く。

$$pSrcDst[n] = pSrcDst[n] - (pSrc[n] + val \cdot pSrc[n-T]) \quad , n = 0,...,59$$

この演算は、リンギングの減算に使用される。リンギングの減算を実行するには、高調波で重み付けされた音声ベクトルからゼロ・インパルス応答を引いて、ターゲット・ベクトルを得る。

$$t[n] = w[n] - z[n]$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pSrcDst* が NULL。

DecodeAdaptiveVector_G723

励振、ピッチ、アダプティブ・ゲインから、アダプティブ・コードブック・ベクトルを復元する。

```
IppStatus ippsDecodeAdaptiveVector_G723_16s(Ipp16s valBaseDelay,
    Ipp16s valCloseLag, Ipp16s valAdptGainIndex, const Ipp16s
    *pSrcPrevExcitation, Ipp16s *pDstAdptVector, IppSpchBitRate
    bitRate);
```

引数

valBaseDelay	[18,145] の範囲のベース遅延。
valCloseLag	閉ループ・ピッチのラグ。
valAdptGainIndex	アダプティブ・ゲインのインデックス。
pSrcPrevExcitation	過去の励振ベクトル [145] へのポインタ。
bitRate	送信ビット・レート、IPP_SPCHBR_6300 または IPP_SPCHBR_5300 に等しい。
pDstAdptVector	アダプティブ・コードブック・ベクトル [60] へのポインタ。

説明

関数 ippsDecodeAdaptiveVector_G723 は、ippsc.h ファイルで宣言される。この関数は、励振、閉ループ・ピッチ、アダプティブ・ゲイン・インデックスから、アダプティブ・ベクトルをデコードする。この関数はサブフレーム内で次のように適用される。

1. ピッチを求める。

$$L = L_o + L_c - 1,$$

ここで、L_o はオープン・ループ・ピッチ、L_c は閉ループ・ピッチのラグである。

2. 次の関係を使用して、直前の励振から残留信号を求める。

res(0) = *exci*(143-*L*);

res(1) = *exci*(143-*L*+1);

第 2 サブフレームをデコードする場合、L_o と L_c の和が 146 になり得るため、要素 *exci*(-2) と *exci*(-1) が利用可能でなければならない。

$$res(i+2) = exci(145 - L + (i\%L)),\ i = 0...61$$

3. 85 エントリまたは 170 エントリのコードブックから、アダプティブ・コードブック G_b を選択する。

4. 次の公式を使用して、アダプティブ・ベクトルを得る。

$$c(i) = \sum_{j=0}^{d} res(i+j)G_b(20k+i),\ i = 0...59$$

ここで、k はアダプティブ・ゲイン・インデックスである。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcPrevExcitation* または *pDstAdptVector* が NULL。
IppStsRangeErr	エラー。*valBaseDelay* が [18, 145] の範囲内にないか、*valCloseLag* が [0, 2] の範囲内にないか、*valAdptGainIndex* が [0, 169] の範囲内にないか、*bitRate* が列挙型 IppSpchBitRate の有効な要素でない。

PitchPostFilter_G723

ピッチ・ポスト・フィルタの係数を
計算する。

```
IppStatus ippsPitchPostFilter_G723_16s(Ipp16s valBaseDelay, const
    Ipp16s *pSrcResidual, Ipp16s *pResultDelay, Ipp16s
    *pResultPitchGain, Ipp16s *pResultScalingGain, Ipp16s subFrame,
    IppSpchBitRate bitRate);
```

引数

valBaseDelay	[18,145] の範囲のベース遅延。
pSrcResidual	残留信号ベクトル [365] へのポインタ。このポインタは、146 番目の要素を指す。
pResultDelay	ピッチ・ポスト・フィルタの遅延へのポインタ。
pResultPitchGain	Q15 のピッチ・ポスト・フィルタのゲインへのポインタ。
pResultScalingGain	Q15 のピッチ・ポスト・フィルタのスケーリング・ゲインへのポインタ。
subFrame	サブフレーム番号、0 ～ 3。
bitRate	送信ビット・レート、`IPP_SPCHBR_6300` または `IPP_SPCHBR_5300` に等しい。

説明

関数 `ippsPitchPostFilter_G723` は、`ippsc.h` ファイルで宣言される。この関数は、ピッチ・ポスト・フィルタの係数を計算する。この関数はサブフレーム内で次のように適用される。

1. 順方向の相互相関から順方向のピッチ・ラグ M_f を検索する。

$$C_f = \sum_{i=0}^{59} res(i)res(i+M_f) \ , M_1 \le M_f \le M_2$$

ここで $M_1 = valBaseDelay - 3$ および $M_2 = valBaseDelay + 3$ 。

2. 逆方向の相互相関から逆方向のピッチ・ラグ M_b を検索する。

$$C_b = \sum_{i=0}^{59} res(i)res(i-M_b), M_1 \le M_b \le M_2$$

3. C_f または C_b が負の場合は、対応する重みと遅延はゼロに設定される。

4. 関連する残留エネルギーを次のように計算する。

$$D_f = \sum_{i=0}^{59} res(i+M_f)res(i+M_f)$$

$$D_b = \sum_{i=0}^{59} res(i-M_b)res(i-M_b)$$

$$T_{en} = \sum_{i=0}^{59} res(i)res(i)$$

5. 最適なゲインを求める。$g = C/D$

6. 次の公式を使用して、スケーリング・ゲインを計算する。

$$g_s = \sqrt{\sum_{i=0}^{59} res^2(i) / \sum_{i=0}^{59} res'^2(i)}$$

ここで、$res'(i)$ は、ポスト・フィルタリングされた残留信号である。

7. ポスト・フィルタの全ゲインを計算する。

$$g_p = \gamma_{ltp} \cdot g_p \cdot g$$

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSrcResidual*、*pResultDelay*、*pResultPitchGain* または *pResultScalingGain* が NULL。
ippStsRangeErr	エラー。*valBaseDelay* が [18, 145] の範囲内にないか、 サブフレームが 0 または 3 でないか、 *bitRate* が列挙型 IppSpchBitRate の有効な要素でない。

GSM-AMR に関連する関数

この項では、欧州通信規格協会（ETSI）による移動体通信用グローバルシステム（GSM）標準の適応マルチレート（AMR）である ETSI EN 301 704 GSM 06.90 version 7.5.0 Release 2001 音声コーデック（一般的に GSM-AMR 06.90 コーデックと呼ばれる）のビットごとの実装を行うインテル® IPP 関数を説明する。プリミティブは主に、GSM-AMR システムのコーデック部分を構成する、計算に時間のかかる演算を行う。

GSM 06.90 AMR 音声コーデックは、4.75、5.15、5.90、6.70、7.40、7.95、10.2、および 12.2 Kbps に圧縮したデータ・レートを使用して、電話帯域デジタル音声を効率的に表現する適応マルチレート・アルゴリズムを構成する。GSM-AMR システムは、音声コーデックのビット・レートを、チャネルの状態に合わせて出力品質が最大限になるよう制御する。

表 9-4 は、GSM-AMR プリミティブの関数グループを示す。

表 9-4　GSM-AMR コーデック・プリミティブの関数グループと概要

関数サブセット	関数名
基本関数	Interpolate_GSMAMR FFTFwd_RToPerm_GSMAMR
LP 分析	AutoCorr_GSMAMR LevinsonDurbin_GSMAMR LPCToLSP_GSMAMR LSPToLPC_GSMAMR LSFToLSP_GSMAMR LSPQuant_GSMAMR QuantLSPDecode_GSMAMR

intel®

表 9-4 GSM-AMR コーデック・プリミティブの関数グループと概要

アダプティブ・コードブック検索	OpenLoopPitchSearchNonDTX_GSMAMR OpenLoopPitchSearchDTXVAD1_GSMAMR OpenLoopPitchSearchDTXVAD2_GSMAMR ImpulseResponseTarget_GSMAMR AdaptiveCodebookSearch_GSMAMR AdaptiveCodebookDecode_GSMAMR AdaptiveCodebookGain_GSMAMR
固定コードブック検索	AlgebraicCodebookSearch_GSMAMR FixedCodebookDecode_GSMAMR
断片的な送信	Preemphasize_GSMAMR VAD1_GSMAMR VAD2_GSMAMR EncDTXSID_GSMAMR EncDTXHandler_GSMAMR EncDTXBuffer_GSMAMR, DecDTXBuffer_GSMAMR
後処理	PostFilter_GSMAMR

この項では、表 9-4 で要約された各 GSM-AMR 関数ブロックを実装する次のプリミティブの詳細について説明する。

基本関数の項では、いくつかの一般的なプリミティブについて説明する。

LP 分析と量子化プリミティブの項では、LP 分析プリミティブについて説明する。

アダプティブ・コードブック・プリミティブと固定コードブック検索の項では、アダプティブ・コードブック検索と固定コードブック検索で使用するプリミティブについて説明する。

断片的な送信（DTX）の項では、断片的な送信（DTX）で使用するプリミティブ、音声アクティビティ検出（VAD）アルゴリズム 1 と 2 を実装する関数、およびコンフォート・ノイズ生成（CNG）と DTX バッファ管理で使用するプリミティブについて説明する。

後処理の項では、後処理プリミティブについて説明する。

基本関数

Interpolate_GSMAMR

2 つのベクトルの重み付けされた和を
計算する。

```
IppStatus ippsInterpolate_GSMAMR_16s (const Ipp16s *pSrc1, const Ipp16s
    *pSrc2, Ipp16s *pDst, int len);
```

引数

pSrc1	1 番目の入力ベクトル [*len*] へのポインタ。
pSrc2	2 番目の入力ベクトル [*len*] へのポインタ。
pDst	出力ベクトル [*len*] へのポインタ。
len	入力および出力ベクトルの長さ。

説明

関数 ippsInterpolate_GSMAMR は、ippsc.h ファイルで宣言される。この関数は、2 つのベクトルの重み付けされた和を計算する。

$$pDst[i] = (pSrc1[i]>>2) + (pSrc2[i] - (pSrc2[i]>>2)); \quad 0 \le i < len$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pDst* が NULL。
ippStsSizeErr	エラー。 *len* がゼロ以下。

FFTFwd_RToPerm_GSMAMR

実数信号に対する順方向の高速フーリエ変換
（FFT）を計算する。

```
IppStatus ippsFFTFwd_RToPerm_GSMAMR_16s_I (Ipp16s *pSrcDst);
```


intel®

引数

pSrcDst	入力および出力ベクトル [128] へのポインタ。

説明

関数 `ippsFFTFwd_RToPerm_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、実数信号に対する順方向の FFT を計算する。この変換は、次の設定を使用した汎用信号処理 FFT 関数 `ippsFFTFwd_RToPerm_16s_Sfs` と同じである：`order` = 7, `flag` = `IPP_FFT_DIV_FWD_BY_N`, `scale factor` = -1。`ippsFFTFwd_RToPerm_16s_Sfs` で使用する丸めモードは、GSM AMR コーデックによるビットごとの正確な操作を保証しないため、関数の演算結果が異なる場合がある（第 7 章の「変換関数」を参照のこと）。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。*pSrcDst* ポインタが NULL。

LP 分析と量子化プリミティブ

この項は、LP 分析、量子化、エンコード、およびデコードで使用する GSM-AMR プリミティブについて説明する。また、次の処理を行うためのプリミティブについても説明する。

- 自己相関分析
- Levinson-Durbin アルゴリズム
- LPC から LSP への変換
- LSP から LPC への変換
- LSP の量子化と逆量子化
- 量子化された LSP のエンコードとデコード

AutoCorr_GSMAMR

サンプルのブロックの自己相関シーケンス
を分析する。

```
IppStatus ippsAutoCorr_GSMAMR_16s32s(const Ipp16s * pSrcSpch, Ipp32s *
    pDstAutoCorr, IppSpchBitRate mode);
```

引数

pSrcSpch	Q15.0 の入力音声ベクトル（240 サンプル）へのポインタ。8 バイト境界にアライメントする必要がある。
pDstAutoCorr	長さが 22 の自己相関係数へのポインタ。12.2 Kbps モードでは、要素 0 から 10 に自己相関ラグの 1 番目のセットが格納され、要素 11 から 21 に自己相関ラグの 2 番目のセットが格納される。その他すべてのモードでは、要素 0 から 10 に自己相関ラグのセットが 1 つだけ格納される。
mode	ビット・レート指定子。有効な値は、`IPP_SPCHBR_4750` から `IPP_SPCHBR_12200` である。

説明

関数 `ippsAutoCorr_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、240 サンプルのブロックの自己相関シーケンスを分析する（30 ms）。12.2 Kbps モードでは、現在の 20 ms フレームから 160 個のサンプルと直前のフレームから最後の 80 個のサンプルを組み合せ、2 つの自己相関シーケンスが推定される。その他すべてのビット・レートでは、現在のフレームから 160 個のサンプル、直前のフレームから最後の 40 個のサンプル、そして次のフレームから最初の 40 個のサンプルを組み合わせ、1 つの自己相関シーケンスが推定される。推定は次のように行われる。

1. テーパー窓－非対称テーパー窓が入力音声に適用される。ビット・レートに基づいて、異なる窓が選択される。12.2 Kbps フレームでは、次に示す特別なテーパー窓が 2 つの自己相関推定にそれぞれ適用される。

$$w_1(n) = \begin{cases} 0.54 - 0.46\cos(\frac{n\pi}{159}), n = 0,1,...,159 \\ 0.54 + 0.46\cos(\frac{(n-160)\pi}{79}), n = 160,161,...,239 \end{cases}$$

および

$$w_2(n) = \begin{cases} 0.54 - 0.46\cos(\frac{2n\pi}{463}), n = 0,1,...,231 \\ \cos(\frac{2(n-232)\pi}{31}), n = 232,233,...,239 \end{cases}$$

w_1 と w_2 はルック・アヘッドを持たない。12.2 Kbps 以外のビット・レートでは、現在のフレームの中心に位置する窓が適用される。

$$w_3(n) = \begin{cases} 0.54 - 0.46\cos(\frac{2n\pi}{399}), n = 0,1,...,199 \\ \cos(\frac{2(n-200)\pi}{159}), n = 200,201,...,239 \end{cases}$$

自己相関ラグの推定－自己相関ラグは、窓処理された音声サンプル $s(i)$, $i = 0, 1, \ldots, 239$ から次のように推定される。

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k) \;,\; k = 0, 1, \ldots, 10$$

2. 帯域幅の拡張－ステップ 2 で取得した自己相関シーケンスに次の二項ラグ窓を適用する。

$$bi(i) = \exp[-0.5 \times (\frac{2\pi f_0 i}{f_s})^2], i = 0,1,...,10$$

ここで、$f_0 = 60$ Hz および $f_s = 8000$ Hz となる。

戻り値

ippStsNoErr	エラーなし。

`ippStsBadArgErr`	エラー。ポインタ *pSrcSpch* または *pDstAutoCorr* が NULL。
`ippStsRangeErr`	エラー。*mode* が列挙型 IppSpchBitRate の有効な要素でない。

LevinsonDurbin_GSMAMR

Levinson-Durbin アルゴリズムを使用して
LP 係数を計算する。

```
IppStatus ippsLevinsonDurbin_GSMAMR(const Ipp32s * pSrcAutoCorr,
    Ipp16s * pSrcDstLpc);
IppStatus ippsLevinsonDurbin_GSMAMR_32s16s (const Ipp32s *
    pSrcAutoCorr, Ipp16s * pSrcDstLpc);
```

引数

pSrcAutoCorr	自己相関係数ベクトル [11] へのポインタ。
pSrcDstLpc	直前のフレームに関連した、Q3.12 の LP 係数ベクトル [11] へのポインタ。出力では、現在のフレームに関連した、Q3.12 の LP 係数 [11] を指すよう更新される。

説明

関数 `ippsLevinsonDurbin_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、Levinson-Durbin アルゴリズムを使用して、自己相関係数から 10 次 LP ラグを計算する。プリミティブによって実行される詳細ステップを次に示す。

1. 平均 2 乗における予測残留信号の最小化は、LP 係数のセット (a_i, $i = 1, 2, \ldots, 10$) を生成する Levinson-Durbin アルゴリズムを使用することで、効果的に処理できる連立一次方程式を生成する。

   つまり、$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k)$, $k = 1,2,\ldots,10$ となる。

2. Levinson-Durbin アルゴリズムは、次の再帰を実行して、分析係数を取得する。

$$E^{[0]} = r(o)$$

for $i = 1$ to 10

$$a_0^{[i-1]} = 1$$

$$k_i = -\left[\sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j)\right] / E^{[i-1]}$$

$$a_i^{[i]} = k_i$$

for $j = 1$ to $i-1$

$$a_j^{[i]} = a_j^{[i-1]} + k_i \times a_{i-j}^{[i-1]}$$

end

$$E^{[i]} = E^{[i-1]} - k_i^2 E^{[i-1]}$$

end

3. 不安定な合成フィルタの例外処理 : 自己相関分析によって、不安定な LPC 合成フィルタが引き起こされる場合（例 : $|k_i|$ が 1.0 に近いか、または大きい）、直前のフレームに関連付けられた LP 係数は、現在のフレームで予測された LP 係数を置換する必要がある。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsBadArgErr`	エラー。ポインタ *pSrcAutoCorr* または *pSrcDstLpc* が NULL。

LPCToLSP_GSMAMR

LP 係数を LSP 係数に変換する。

```
IppStatus ippsLPCToLSP_GSMAMR_16s(const Ipp16s * pSrcLpc, const Ipp16s
     * pSrcPrevLsp, Ipp16s * pDstLsp);
```

引数

pSrcLpc	Q3.12 の LP 係数ベクトル [11] へのポインタ。
pSrcPrevLsp	直前のフレームに関連した、Q0.15 の LSP 係数ベクトル [10] へのポインタ。
pDstLsp	Q0.15 の LSP 係数ベクトル [10] へのポインタ。

説明

関数 `ippsLPCToLSP_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、一連の 10 次 LP 係数を LSP 係数に変換する。この関数の機能は次のとおりである。

1. 次の再帰的関係を使用して、$F_1(z)$ と $F_2(z)$ の多項式係数を計算する。

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i)$$

$$f_2(i+1) = a_{i+1} - a_{10-i} + f_2(i),\ i = 0,1,...,4$$

ここで、$f_1(0) = f_2(0) = 1.0$ である。

2. Chebyshev 多項式を使用して $F_1(z)$ と $F_2(z)$ を求める。Chebyshev 多項式は、次の式から得られる。

$$C_1(\omega) = \cos(5\omega) + f_1(1)\times\cos(4\omega) + f_1(2)\times\cos(3\omega) + f_1(3)\times\cos(2\omega)$$
$$+ f_1(4)\times\cos(\omega) + f_1(5)/2$$

$$C_2(\omega) = \cos(5\omega) + f_2(1)\times\cos(4\omega) + f_2(2)\times\cos(3\omega) + f_2(3)\times\cos(2\omega)$$
$$+ f_2(4)\times\cos(\omega) + f_2(5)/2$$

3. 0 と p の間の等間隔の 60 個の点で F_1(z) と F_2(z) を求め、符号の変化がないかどうかチェックする。符号の変化は、根が存在することを示す。符号が変化している場合は、符号の変化の間隔を 4 回割って、根を求める。

4. LSP 係数を求めるのに必要な 10 個の根が検索中に見つからない場合は、直前の LSP が使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。ポインタ *pSrcLpc*、*pSrcPrevLsp*、または *pDstLsp* が NULL。

LSPToLPC_GSMAMR

LSP 係数を LP 係数に変換する。

```
IppStatus ippsLSPToLPC_GSMAMR_16s(const Ipp16s * pSrcLsp, Ipp16s *
    pDstLpc);
```

引数

pSrcLsp	Q0.15 の LSP 係数ベクトル [10] へのポインタ。
pDstLpc	Q3.12 の LP 係数ベクトル [11] へのポインタ。

説明

関数 ippsLSPToLPC_GSMAMR は、ippsc.h ファイルで宣言される。この関数は、一連の 10 次 LSP 係数を LP 係数に変換する。この関数の機能は次のとおりである。

1. 次の再帰的関係を使用して、$F_1(z)$ と $F_2(z)$ の多項式係数を計算する。

for i = 1 to 5

$$f_1(i) = -2q_{2i-1} \times f_1(i-1) + 2f_1(i-2)$$

for j = i-1 down to 1

$$f_1(j) = f_1(j) - 2q_{2i-1} \times f_1(j-1) + f_1(j-2)$$

end

end

ここで、初期値は $f_1(0) = 1$ と $f_1(-1) = 0$ である。係数 $f_2(i)$ は、q_{2i-1} を q_{2i} で置き換えて、同じように計算される。

2. 次に、$F_1(z)$ と $F_2(z)$ にそれぞれ $1+z^{-1}$ と $1-z^{-1}$ を掛けて、次の式を使用して $F_1'(z)$ と $F_2'(z)$ を求める。

$$f_1'(i) = f_1(i) + f_1(i-1), \quad i = 1,2,\ldots 5$$
$$f_2'(i) = f_2(i) - f_2(i-1), \quad i = 1,2,\ldots 5$$

3. 次の式を使用して、多項式 $F_1'(z)$ と $F_2'(z)$ から LP 係数を計算する。

$$a_i = \begin{cases} 0.5 \times f_1'(i) + 0.5 \times f_2'(i), i = 1,2,\ldots,5 \\ 0.5 \times f_1'(11-i) - 0.5 \times f_2'(11-i), i = 6,7,\ldots,10 \end{cases}$$

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。ポインタ *pSrcLsp* または *pDstLpc* が NULL。

LSFToLSP_GSMAMR

線スペクトル周波数を LSP 係数に変換する。

```
IppStatus ippsLSFToLSP_GSMAMR_16s (const Ipp16s *pLSF, Ipp16s *pLSP) ;
```

引数

pLSF	範囲 [0,1] の Q14 の 1/p 係数で正規化された LSF 係数ベクトル [10] へのポインタ。
pLSP	範囲 [-1;1] の Q15 の LSP ベクトル [10] へのポインタ。

説明

関数 ippsLSPToLSF_GSMAMR は、ippsc.h ファイルで宣言される。この関数は、次の式に従って、線スペクトル周波数（LSF）を LSP 係数に変換する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pLSF* または *pLSF* が NULL。

LSPQuant_GSMAMR

LSP 係数ベクトルを量子化する。

```
IppStatus ippsLSPQuant_GSMAMR_16s(const Ipp16s * pSrcLsp, Ipp16s *
    pSrcDstPrevQLsfResidual, Ipp16s * pDstQLsp, Ipp16s * pDstQLspIndex,
    IppSpchBitRate  mode);
```

引数

pSrcLsp	Q0.15 の量子化されていない LSP ベクトル [20] へのポインタ。12.2 Kbps フレームでは、1 番目の LSP セットはベクトル要素 0 から 9 に格納され、2 番目の LSP セットはベクトル要素 10 から 19 に格納される。その他すべてのビット・レートでは、ベクトル要素 0 から 9 のみ量子化で使用することができる。
pSrcDstPrevQLsfResidual	直前のフレームから量子化した LSF 残留信号へのポインタ。Q0.15 で表現され、要素の数は 10 である。出力では、現在のフレームから量子化した LSP 残留信号 [10] を指す。Q0.15 で表現され、要素の数は 10 である。
pDstQLsp	Q0.15 の量子化された LSP ベクトル [20] へのポインタ。12.2 Kbps フレームでは、要素 0 から 9 に 1 番目の量子化された LSP セットが格納され、10 から 19 に 2 番目の量子化された LSP セットが格納される。その他すべてのビット・レートでは、要素 0 から 9 に LSP セットが 1 つだけ格納される。
pDstQLspIndex	量子化された LSP インデックスのベクトル [5] へのポインタ。12.2 Kbps フレームでは、5 つのすべての要素に有効なデータが格納される。その他すべてのビット・レートでは、最初の 3 つの要素にのみ有効なインデックスが格納される (モードについては SMQ と SVQ の項で説明する)。
mode	ビット・レート指定子。有効な列挙値は、`IPP_SPCHBR_4750` から `IPP_SPCHBR_12200` である。

説明

関数 `ippsLSPQuant_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、LSP 係数ベクトルを量子化して、量子化した LSP コードブック・インデックスを得る。この関数の機能は次のとおりである。

1. LSP から LSF への変換－ LSP は線スペクトル周波数（LSF）にマップされる。12.2 Kbps フレームでは、2 つの LSP セットが LSF に変換される。その他すべてのビット・レートでは、次の関係を使用して 1 つの LSP セットが変換される。

$$f_i = \frac{f_s}{2\pi}\arccos(q_i)$$

2. LSF 予測－ 1 次移動平均（MA）予測が LSF ベクトルに適用される。次に、予測残留信号が量子化される。12.2 Kbps フレームでは、2 つの予測残量信号ベクトル $r^{(1)}(n)$ と $r^{(2)}(n)$ が次のように構成される。

$$r^{(1)}(n) = z^{(1)}(n) - 0.65\hat{r}^{(2)}(n-1)\ ,\ r^{(2)}(n) = z^{(2)}(n) - 0.65\hat{r}^{(2)}(n-1)$$

ここで $z^{(1)}(n)$ と $z^{(2)}(n)$ は、フレーム n でのゼロ平均 LSF ベクトルである。$\hat{r}^{(2)}(n-1)$ は、フレーム n-1 から量子化された残留信号ベクトルである。

その他すべてのビット・レートでは、信号予測残留信号ベクトル $r(n)$ は次の式から得られる。

$$r_j(n) = z_j(n) - \alpha_j\hat{r}_j(n-1) \quad j = 0,...,9$$

ここで、$z(n)$ はフレーム n での平均削除 LSF ベクトルである。$\hat{r}(n-1)$ は、フレーム n-1 から量子化された残留信号ベクトルである。

3. 量子化－分割行列量子化 (SMQ) は、予測残留信号ベクトルに適用される。12.2 Kbps フレームでは、行列（$r^{(1)}, r^{(2)}$）は 5 つのサブ行列（次元は 2 x 2）に分割され、それぞれ 7、8、9、8、および 6 ビットで量子化される。その他すべてのビット・レートでは、ベクトル r は 3 つのサブベクトルに分割される（次元は 3、3、および 4)。これら 3 つのサブベクトルは、7、8、および 9 ビットで量子化される。VQ 検索は、重み付けされたエラーを最小にするインデックス *k* を識別する。

$$E_{LSP} = \sum_{i=0}^{9}[r_i w_i - \hat{r}_i^k w_i]^2$$

重み係数 w_i は次の式から得られる。

$$w_i = \begin{cases} 3.347 - \frac{1.547}{450} d_i, & d_i < 450 \\ 1.8 - \frac{0.8}{1050}(d_i - 450), & otherwise \end{cases}$$

また、$f_0 = 0$ および $f_{11} = 4000$ の場合、$d_i = f_{i+1} - f_{i-1}$ となる。

4. 共鳴の最小化－量子化された LSF ベクトル要素は、LP 合成フィルタ内で強い共鳴が起こらないように再調整される。

5. LSF から LSP への変換－量子化された LSF を LSP に変換する。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。ポインタ *pSrcLsp*、*pSrcDstPrevQLsfResidual*、*pDstQLsp*、または *pDstQLspIndex* が NULL。
ippStsRangeErr	エラー。*mode* が列挙型 IppSpchBitRate の有効な要素ではない。

QuantLSPDecode_GSMAMR

量子化された LSP をデコードする。

```
IppStatus ippsQuantLSPDecode_GSMAMR_16s(const Ipp16s * pSrcQLspIndex,
    Ipp16s * pSrcDstPrevQLsfResidual, Ipp16s * pSrcDstPrevQLsf, Ipp16s
    * pSrcDstPrevQLsp, Ipp16s * pDstQLsp, Ipp16s bfi, IppSpchBitRate
    mode);
```

引数

pSrcQLspIndex	量子化された LSP のコードブック・インデックスを格納するベクトル [5] へのポインタ。12.2 Kbps フレームでは、5 つのすべての要素に有効なインデックスが格納される。その他すべてのビット・レートでは、最初の 3 つの要素にのみ有効なインデックスが格納される。

pSrcDstPrevQLsfResidual	直前のフレームから量子化した LSF 残留信号へのポインタ（Q0.15）。出力では、更新された LSP 残留信号を指す（Q0.15）。
pSrcDstPrevQLsf	直前のフレームから量子化した LSF ベクトル [10] へのポインタ（Q0.15）。出力では、更新された量子化 LSP ベクトル [10] を指す（Q0.15）。
pSrcDstPrevQLsp	直前のフレームから量子化した LSP ベクトル [10] へのポインタ（Q0.15）。出力では、更新された量子化 LSP ベクトル [10] を指す（Q0.15）。
pDstQLsp	4 つのサブフレーム LSP セットを含むベクトル [40] へのポインタ。12.2 Kbps フレームでは、2 つのセットが補間によって生成される。その他すべてのビット・レートでは、3 つのセットが補間によって生成される。すべての要素は、Q0.15 を使用する。
bfi	不良フレーム・インジケータ。「0」は良いフレームを意味する。それ以外の値は不良フレームを意味する。
mode	ビット・レート指定子。有効な列挙値は、`IPP_SPCHBR_4750` から `IPP_SPCHBR_12200` である。

説明

関数 `ippsQuantLSPDecode_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、受け取ったフレームでエラーが検出されない場合、受け取ったコードブック・インデックスから量子化された LSP をデコードする。それ以外の場合、関数は線形補間を使用して、量子化された LSP を直前の量子化された LSP で回復する。この関数の機能は次のとおりである。

1. 現在のフレームでエラーが検出されなかった場合、逆方向の LSP 量子化を使用して、コードブック・インデックスから量子化された LSP と直前の量子化された LSP 残留信号を取得する。

2. 現在のフレームでエラーが検出された場合、レートに依存した次の補間手順を使用して、量子化された LSF を取得する。

$$\begin{cases} lsf_q1(i) = lsf_q2(i) = \alpha \times past_lsf_q(i) + (1-\alpha) \times mean_lsf(i) & 12.2\,\text{kbit/s mode} \\ lsf_q(i) = \alpha \times past_lsf_q(i) + (1-\alpha) \times mean_lsf(i) & otherwise \end{cases}$$

ここで、$i = 0, 1, \ldots, 9$, $\alpha = 0.95$ である。*lsf_q1* と *lsf_q2*（12.2 Kbps の場合）は、現在のフレームの 2 つの量子化された LSF ベクトルのセットである。*past_lsf_q* は直前のフレームの *lsf_q2*、*mean_lsf* は平均 LSF ベクトルである。12.2 Kbps 以外のレートでは、量子化された LSF 係数のセットは 1 つのみである。対応する量子化された LSP は、LSF から LSP への変換によって取得される。

3. 線形補間は、直前のフレームから量子化した LSP のセットを 4 つ生成するために適用される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsBadArgErr`	エラー。ポインタ *pSrcQLspIndex*、*pSrcDstPrevQLsfResidual*、*pSrcDstPrevQLsf*、*pSrcDstPrevQLsp*、または *pDstQLsp* が NULL。
`ippStsRangeErr`	エラー。*mode* が列挙型 `IppSpchBitRate` の有効な要素でない。

アダプティブ・コードブック・プリミティブ

この項では、次の関数を実行するプリミティブを含む、アダプティブ・コードブックに関するプリミティブを説明する。

- 非 DTX、VAD1、および VAD2 を含むオープン・ループ・ピッチの検索
- インパルス応答およびターゲット信号計算
- アダプティブ・コードブック検索
- アダプティブ・コードブック・ベクトルのデコード

オープン・ループ・ピッチの検索（OLP）

OLP 検索では 3 つの関数が用意されている。これらの関数は、相互排他的に使用する。つまり、各フレームのタイプにつきアプリケーションで使用できる適切な関数は 1 つのみである。適切な OLP 検索関数は、DTX および VAD モジュールの状態によって異なる。各モードで適切な OLP 検索関数は次のとおりである。

エンコーダ・モード	適切な関数
DTX 無効	`ippsOpenLoopPitchSearchNonDTX_GSMAMR`
DTX、VAD 1 有効	`ippsOpenLoopPitchSearchDTXVAD1_GSMAMR`
DTX、VAD 2 有効	`ippsOpenLoopPitchSearchDTXVAD2_GSMAMR`

OLP 検索関数は、重み付けされた入力音声からピッチ推定を抽出する。5.15 および 4.75 Kbps フレームでは、検索は 1 フレームごとに 1 回行われる。その他すべてのモードでは、検索は 1 フレームごとに 2 回行われる。10.2 Kbps フレームでは、特殊な検索が行われる。OLP 検索の詳細は次のとおりである。

1. 送信ビット・レートが 10.2 Kbps の場合、次の OLP 検索手順が使用される。

   a. 重み付けされた音声の窓処理された自己相関を次のように計算する。

$$R(k) = w(k) \times \sum_{n=0}^{79} sw(n) \times sw(n-k), \quad 20 \le k \le 143$$

ここで、シーケンス $sw(n)$ には重み付けされた音声が含まれる。w は、次の重み付け関数である。

$$w(k) = wl(k) \times wn(k)$$

この重み付け関数の wl はロー・ピッチを示し、テーブル cw で定義される。ここで、wn は、次のような直前のフレームに関連付けられた隣接強調ラグ係数のシーケンスである。

$$wn(k) = \begin{cases} cw(|T_{old} - k| + 20) & weightflag = 1 \\ 1 & otherwise \end{cases}$$

パラメータ T_{old} は、直前の 5 つの有声音声ハーフフレームからメディアン・フィルタリングされたピッチ・ラグである。

推定されたオープン・ループ・ピッチのラグは、$R(k)$ を最大にする値 k である。これは、T_{op} によって示される。

   b. 次の関係を使用して、最適なオープン・ループ・ゲインを計算する。

$$g = \frac{\sum_{n=0}^{79} sw(n) \times sw(n - T_{op})}{\sum_{n=0}^{79} sw(n) \times sw(n)} - 0.4$$

さらに、

$$v = \begin{cases} 1 & g > 0 \\ 0.9v & otherwise \end{cases}$$

また、次の式が使用される。

$$weightflag = \begin{cases} 1 & v > 0.3 \\ 0 & otherwise \end{cases}$$

$g > 0$ の場合、直前のピッチ・ラグ・バッファと直前のピッチ・ラグのメディアン・ピッチ・ラグは更新される。

2. 10.2 Kbps 以外のレートでは、次の OLP 検索手順が使用される。

a. 次の式を使用して、3 つの異なる範囲から 3 つの最大相関値が検索される。

$$R(k) = \sum_{n=0}^{length-1} sw(n) \times sw(n-k)$$

5.15 と 4.75 Kbps フレームでは、*length* = 160 となる。その他すべてのビット・レート（10.2 Kbps を除く）では、*length* = 80 となる。

b. 3 つの最大相関値を次の式で正規化する。

$$M_i = \frac{M_i}{\sqrt{\sum_{n=0}^{length-1} sw^2(n - T_i)}} \qquad i = 1,2,3$$

c. 次の規則を使用して、最適なオープン・ループ・ラグを決定する。

$$\begin{aligned}
&T_{op} = T_1, \quad M(T_{op}) = M_1 \\
&if\,(M_2 > 0.85M(T_{op})) \\
&\quad M(T_{op}) = M_2, \quad and\ T_{op} = T_2 \\
&if\,(M_3 > 0.85M(T_{op})) \\
&\quad T_{op} = T_3
\end{aligned}$$

各 OLP 検索関数は、上記のレートに依存した検索アルゴリズムを使用する。次に、非 DTX、VAD1、および VAD2 の OLP 検索プリミティブについて説明する。

OpenLoopPitchSearchNonDTX_GSMAMR

オープン・ループ・ピッチのラグを
計算する。

```
IppStatus ippsOpenLoopPitchSearchNonDTX_GSMAMR_16s(const Ipp16s *
    pSrcWgtLpc1, const Ipp16s * pSrcWgtLpc2, const Ipp16s * pSrcSpch,
    Ipp16s * pValResultPrevMidPitchLag, Ipp16s * pValResultVvalue,
    Ipp16s * pSrcDstPrevPitchLag, Ipp16s * pSrcDstPrevWgtSpch,
    Ipp16s * pDstOpenLoopLag, Ipp16s * pDstOpenLoopGain, IppSpchBitRate
    mode);
```

引数

pSrcWgtLpc1	Q3.12 の重み付けされた LP 係数ベクトル [44] へのポインタ。この LP 係数のセットは、知覚重み付けフィルタの分子係数を構成する。
pSrcWgtLpc2	Q3.12 の重み付けされた LP 係数ベクトル [44] へのポインタ。この LP 係数のセットは、知覚重み付けフィルタの分母係数を構成する。
pSrcSpch	Q15.0 の入力音声ベクトル（170 サンプル）へのポインタ。
pValResultPrevMidPitchLag	直前の 5 つの有声音声ハーフフレームからメディアン・フィルタリングされたピッチ・ラグのベクトルへのポインタ（Q15.0)。 出力では、更新された、直前の 5 つの有声音声ハーフフレームからメディアン・フィルタリングされたピッチ・ラグのベクトルを指す（Q15.0)。この引数は、10.2 Kbps フレームでのみ有効である。
pValResultVvalue	オープン・ループ・ピッチの検索（OLP）で説明されたアダプティブ・パラメータ *v* へのポインタ（Q0.15)。出力引数としても使用される。 この引数は、10.2 Kbps フレームでのみ有効である。
pSrcDstPrevPitchLag	最新の 5 つの有声音声ハーフフレームに関連付けられたピッチ・ラグを格納するベクトル [5] へのポインタ。出力では、直前の 5 つの有声音声ハーフフレームに関連付けられたピッチ・ラグの、更新された 5 つの要素ベクトルを指す。この引数は、10.2 Kbps フレームでのみ使用される。

pSrcDstPrevWgtSpch	直前のフレームから知覚的に重み付けされた音声を格納するベクトル [143] へのポインタ（Q15.0）。出力では、知覚的に重み付けされた音声の、更新された 143 個の要素のベクトルを指す（Q15.0）。
pDstOpenLoopLag	オープン・ループのピッチ・ラグのベクトル [2] へのポインタ。5.15 と 4.75 Kbps フレームでは、1 つのラグが推定されるため、最初のベクトル要素にのみ有効なラグの値が格納される。その他すべてのビット・レートでは、両方のベクトル要素に有効なピッチ・ラグの値が格納される。
pDstOpenLoopGain	Q15.0 の最適なオープン・ループのピッチ・ゲインを格納するベクトル [2] へのポインタ。この引数は、10.2 Kbps フレームでのみ有効である。
mode	ビット・レート指定子。有効な値は、IPP_SPCHBR_4750 から IPP_SPCHBR_12200 である。

説明

関数 ippsOpenLoopPitchSearchNonDTX_GSMAMR は、ippsc.h ファイルで宣言される。この関数は、DTX と VAD が無効な場合にオープン・ループのピッチ・ラグを計算する（10.2 Kbps フレームでは、最適なピッチ・ゲインも計算する）。検索アルゴリズムはオープン・ループ・ピッチの検索（OLP）で説明されている。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。任意の入力または出力ポインタが NULL。
ippStsRangeErr	エラー。*mode* が列挙型 IppSpchBitRate の有効な要素でない。

OpenLoopPitchSearchDTXVAD1_GSMAMR

オープン・ループのピッチ・ラグ推定を
抽出する（VAD 1 は有効）。

```
IppStatus ippsOpenLoopPitchSearchDTXVAD1_GSMAMR_16s(const Ipp16s *
    pSrcWgtLpc1, const Ipp16s * pSrcWgtLpc2, const Ipp16s * pSrcSpch,
    Ipp16s * pValResultToneFlag, Ipp16s * pValResultPrevMidPitchLag,
    Ipp16s * pValResultVvalue, Ipp16s * pSrcDstPrevPitchLag,
    Ipp16s * pSrcDstPrevWgtSpch, Ipp16s * pResultMaxHpCorr, Ipp16s *
    pDstOpenLoopLag, Ipp16s * pDstOpenLoopGain, IppSpchBitRate mode);
```

引数

pSrcWgtLpc1	Q3.12 の重み付けされた LP 係数ベクトル [44] へのポインタ。これらの LP 係数は、知覚重み付けフィルタの分子を構成する。
pSrcWgtLpc2	Q3.12 の重み付けされた LP 係数ベクトル [44] へのポインタ。これらの LP 係数は、知覚重み付けフィルタの分母を構成する。
pSrcSpch	Q15.0 の入力音声ベクトル（170 サンプル）へのポインタ。
pValResultToneFlag	VAD モジュールのトーン・フラグへのポインタ。出力では、VAD モジュールの更新されたトーン・フラグを指す。
pValResultPrevMidPitchLag	直前の 5 つの有声音声ハーフフレームからメディアン・フィルタリングされたピッチ・ラグのベクトルへのポインタ（Q15.0）。出力では、更新された、直前の 5 つの有声音声ハーフフレームからメディアン・フィルタリングされたピッチ・ラグのベクトルを指す（Q15.0）。 この引数は、10.2 Kbps フレームでのみ有効である。
pValResultVvalue	Q0.15 のアダプティブ・パラメータ *v* へのポインタ。出力では、Q0.15 の更新されたアダプティブ パラメータ *v* を指す。 この引数は、10.2 Kbps フレームでのみ有効である。
pSrcDstPrevPitchLag	最新の 5 つの有声音声ハーフフレームに関連付けられたピッチ・ラグを格納するベクトル [5] へのポインタ。

intel®

	出力では、最新の 5 つの有声音声ハーフフレームに関連付けられたピッチ・ラグの、更新された 5 つの要素ベクトルを指す。この引数は、10.2 Kbps フレームでのみ有効である。
pSrcDstPrevWgtSpch	直前のフレームに関連付けられた、知覚的に重み付けされた音声を格納するベクトル [143] へのポインタ（Q15.0）。 出力では、直前のフレームに関連付けられた、知覚的に重み付けされた音声の更新された 143 個の要素のベクトルを指す（Q15.0）。
pResultMaxHpCorr	最大相関値へのポインタ。
pDstOpenLoopLag	オープン・ループのピッチ・ラグのベクトル [2] へのポインタ。5.15 と 4.75 Kbps フレームでは、1 つのラグが推定されるため、最初のベクトル要素にのみ有効なラグの値が格納される。その他すべてのビット・レートでは、両方のベクトル要素に有効なピッチ・ラグの値が格納される。
pDstOpenLoopGain	Q15.0 の最適なオープン・ループのピッチ・ゲインを格納するベクトル [2] へのポインタ。この引数は、10.2 Kbps フレームでのみ有効である。
mode	ビット・レート指定子。有効な値は、`IPP_SPCHBR_4750` から `IPP_SPCHBR_12200` である。

説明

関数 `ippsOpenLoopPitchSearchDTXVAD1_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、VAD 1 が有効な場合に、次のような変更を加えたピッチ検索アルゴリズム（オープン・ループ・ピッチの検索（OLP）で説明）を使用して、重み付けされた入力音声からオープン・ループのピッチ・ラグ推定を抽出する。

1. 10.2 Kbps フレームでは、最良のオープン・ループ・ピッチが検出された後で次の変更が行われる。

   a. トーン・フラグ（このフラグを初期化するかまたはリセットすると 0 に設定される）を次のとおりに更新する。

$$toneflag >>= 1$$

$$DelayEnergy = \sum_{n=0}^{79} sw^2(n - T_{op})$$

$$MaxCorr = \sum_{n=0}^{79} sw(n - T_{op}) \times sw(n)$$

$$if\,(0.325 \times DelayEnergy < MaxCorr) \quad toneflag = toneflag\,|\,0x4000$$

b. 現在のフレームに対する 2 番目の OLP 検索では、ハイパス・フィルタリングされた自己相関の最大値を検索する。

$$maxhpcorr = \max(R(k) \times 2 - R(k-1) - R(k+1)|\ \mathrm{k} = 142,..., 24)$$

次に、*maxhpcorr* は *NormFactor = frameEnergy - frameCorr* で正規化される。

$$frameEnergy = \sum_{n=0}^{79} sw^2(n)\ , \quad frameCorr = \sum_{n=0}^{79} sw(n) \times sw(n-1)$$

2. その他すべてのビット・レートでは、次の変更が行われる。

a. オープン・ループ・ピッチの検索を行う前に、トーン・フラグを次のように更新する。

$$toneflag = toneflag >> 1$$

ビット・レートが 5.15 または 4.75 Kbps の場合、トーン・フラグを次のように更新する。

$$toneflag = toneflag >> 1, \quad toneflag = toneflag\,|\,0x2000$$

b. 3 つのオープン・ループ・ピッチの候補を見つけた後は、トーン・フラグを次のように更新する。

$$\mathrm{if}\ (DelayEnergy \times 0.65 < MaxCorr) \quad toneflag = toneflag\,|\,\mathrm{0x4000}$$

この更新は、3 つのピッチ候補に対応する *DelayEnergy* と *MaxCorr* で 3 回繰り返される。4.75 および 5.15 Kbps フレームの *DelayEnergy* と *MaxCorr* の長さは 160 サンプルである。その他すべてのビット・レートでは、長さは 80 サンプルである。

c. 各フレームに対する 2 番目の OLP 検索では、ハイパス自己相関の最大値を検索する。これは、10.2 Kbps における相関検索と同じだが、検索範囲はレートに依存する。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。任意の入力または出力ポインタが NULL。
ippStsRangeErr	エラー。*mode* が列挙型 IppSpchBitRate の有効な要素でない。

OpenLoopPitchSearchDTXVAD2_GSMAMR

オープン・ループのピッチ・ラグ推定を抽出する（VAD 2 は有効）。

```
IppStatus ippsOpenLoopPitchSearchDTXVAD2_GSMAMR(const Ipp16s *
    pSrcWgtLpc1, const Ipp16s * pSrcWgtLpc2, const Ipp16s * pSrcSpch,
    Ipp16s * pValResultPrevMidPitchLag, Ipp16s * pValResultVvalue,
    Ipp16s * pSrcDstPrevPitchLag, Ipp16s * pSrcDstPrevWgtSpch,
    Ipp32s * pResultMaxCorr, Ipp32s pResultWgtEnergy, Ipp16s *
    pDstOpenLoopLag, Ipp16s * pDstOpenLoopGain, IppSpchBitRate mode);
IppStatus ippsOpenLoopPitchSearchDTXVAD2_GSMAMR_16s32s(const Ipp16s *
    pSrcWgtLpc1, const Ipp16s * pSrcWgtLpc2, const Ipp16s * pSrcSpch,
    Ipp16s * pValResultPrevMidPitchLag, Ipp16s * pValResultVvalue,
    Ipp16s * pSrcDstPrevPitchLag, Ipp16s * pSrcDstPrevWgtSpch, Ipp32s *
    pResultMaxCorr, Ipp32s pResultWgtEnergy, Ipp16s * pDstOpenLoopLag,
    Ipp16s * pDstOpenLoopGain, IppSpchBitRate mode);
```

引数

pSrcWgtLpc1	Q3.12 の重み付けされた LP 係数ベクトル [44] へのポインタ。これらの LP 係数は、知覚重み付けフィルタの分子を構成する。
pSrcWgtLpc2	Q3.12 の重み付けされた LP 係数ベクトル [44] へのポインタ。これらの LP 係数は、知覚重み付けフィルタの分母を構成する。
pSrcSpch	Q15.0 の入力音声ベクトル（170 サンプル）へのポインタ。
pValResultToneFlag	VAD モジュールのトーン・フラグへのポインタ。
pValResultPrevMidPitchLag	直前の 5 つの有声音声ハーフフレームからメディアン・フィルタリングされたピッチ・ラグのベクトルへのポインタ（Q15.0）。出力では、更新された、

	直前の 5 つの有声音声ハーフフレームからメディアン・フィルタリングされたピッチ・ラグのベクトルを指す（Q15.0）。この引数は、10.2 Kbps フレームでのみ有効である。
pValResultVvalue	Q0.15 のアダプティブ・パラメータ *v* へのポインタ。出力では、Q0.15 の更新されたアダプティブ パラメータ *v* を指す。この引数は、10.2 Kbps フレームでのみ有効である。
pSrcDstPrevPitchLag	最新の 5 つの有声音声ハーフフレームに関連付けられたピッチ・ラグを格納するベクトル [5] へのポインタ。出力では、最新の 5 つの有声音声ハーフフレームに関連付けられたピッチ・ラグの、更新された 5 つの要素ベクトルを指す。この引数は、10.2 Kbps フレームでのみ有効である。
pSrcDstPrevWgtSpch	直前のフレームに関連付けられた、知覚的に重み付けされた音声を格納するベクトル [143] へのポインタ（Q15.0）。出力では、直前のフレームに関連付けられた、知覚的に重み付けされた音声の更新された 143 個の要素のベクトルを指す（Q15.0）。
pResultMaxCorr	最大相関値へのポインタ。
pResultWgtEnergy	上記で説明した重み付け音声信号の遅延エネルギーへのポインタ。出力はスケーリングされる場合がある。また、Q 表現は使用されない。
pDstOpenLoopLag	オープン・ループのピッチ・ラグのベクトル [2] へのポインタ。5.15 と 4.75 Kbps フレームでは、1 つのラグが推定されるため、最初のベクトル要素にのみ有効なラグの値が格納される。その他すべてのビット・レートでは、両方のベクトル要素に有効なピッチ・ラグの値が格納される。
pDstOpenLoopGain	Q15.0 の最適なオープン・ループのピッチ・ゲインを格納するベクトル [2] へのポインタ。この引数は、10.2 Kbps フレームでのみ有効である。
mode	ビット・レート指定子。有効な値は、`IPP_SPCHBR_4750` から `IPP_SPCHBR_12200` である。

説明

関数 `ippsOpenLoopPitchSearchDTXVAD2_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、VAD 2 が有効な場合に、次のような変更を加えたピッチ検索アルゴリズム（オープン・ループ・ピッチの検索（OLP）で説明）を使用して、重み付けされた入力音声からオープン・ループのピッチ・ラグ推定を抽出する。

最良のオープン・ループ・ピッチを検索した後は、重み付けされた音声から最大相関値 *MaxCorr* と遅延エネルギー *DelayEnergy* を抽出する。4.75 および 5.15 Kbps フレームでは、160 サンプル計算される。その他すべてのビット・レートでは、関係を使用して 80 サンプルのみ計算される。

$$MaxCorr = \sum_{n=0}^{length-1} sw(n) \times sw(n - T_{op})$$

$$DelayEnergy = \sum_{n=0}^{length-1} sw^2(n - T_{op})$$

2 つのハーフフレームにおけるオープン・ループのピッチ・ラグ推定に対応する *MaxCorr* と *DelayEnergy* の値は組み合わされ、完全なフレームにおける *MaxCorr* と *DelayEnergy* の推定（OLP 検索が 1 フレームごとに 1 回のみ実行される 4.75 および 5.15 Kbps フレームを除く）を取得する。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsBadArgErr`	エラー。任意の入力または出力ポインタが `NULL`。
`ippStsRangeErr`	エラー。*mode* が列挙型 `IppSpchBitRate` の有効な要素でない。

ImpulseResponseTarget_GSMAMR

アダプティブ・コードブック検索で必要な
インパルス応答およびターゲット信号を
計算する。

```
IppStatus ippsImpulseResponseTarget_GSMAMR_16s(const Ipp16s * pSrcSpch,
    const Ipp16s * pSrcWgtLpc1, const Ipp16s * pSrcWgtLpc2, const
    Ipp16s * pSrcQLpc, const Ipp16s * pSrcSynFltState, const Ipp16s *
    pSrcWgtFltState, Ipp16s * pDstImpulseResponse, Ipp16s *
    pDstLpResidual, Ipp16s * pDstAdptTarget);
```

引数

pSrcSpch	入力音声ベクトル [50] へのポインタ。要素 0 から 9 は、直前のサブフレームである。要素 10 から 49 は現在のサブフレームである。
pSrcWgtLpc1	現在のサブフレームで $A(z/\gamma_1)$ に関連付けられた重み付け LP 係数のベクトル [11] へのポインタ（Q3.12）。
pSrcWgtLpc2	現在のサブフレームで $A(z/\gamma_2)$ に関連付けられた重み付け LP 係数のベクトル [11] へのポインタ（Q3.12）。
pSrcQLpc	現在のサブフレームの量子化された LP 係数のベクトル [11] へのポインタ（Q3.12）。
pSrcSynFltState	合成フィルタの状態を格納するベクトル [10] へのポインタ（Q15.0）。
pSrcSynFltState	重み付けフィルタの状態を格納するベクトル [10] へのポインタ（Q15.0）。
pDstImpulseResponse	インパルス応答を格納するベクトル [40] へのポインタ（Q3.12）。
pDstLpResidual	LP 残留信号を格納するベクトル [40] へのポインタ（Q15.0）。
pDstAdptTarget	アダプティブ・コードブック検索のターゲット信号を格納するベクトル [40] へのポインタ（Q15.0）。

説明

関数 ippsImpulseResponseTarget_GSMAMR は、ippsc.h ファイルで宣言される。この関数は、アダプティブ・コードブック検索で必要なインパルス応答およびターゲット信号を計算する。この関数は、次の手法を使用してサブフレームを基準に実行される。

1. 重み付け合成フィルタのインパルス応答 $h(n)$

$$H(z)W(z) = A(z/\gamma_1)/[\hat{A}(z)A(z/\gamma_2)]$$

は、フィルタ $1/\hat{A}(z)$ と $1/A(z/\gamma_2)$ をフィルタ $A(z/\gamma_1)$ のゼロでパディングされたインパルス応答に適用することで計算される。

2. ターゲット信号は、LP 残留信号 $res_{LP}(n)$ にカスケードされた合成フィルタと重み付きフィルタ $1/\hat{A}(z)$ と $A(z/\gamma_1)/A(z/\gamma_2)$ をそれぞれ適用することで得られる。また、アダプティブ・コードブック検索は残留信号 $res_{LP}(n)$ 使用して、過去の励振のヒストリを更新する。LP 残留信号は、入力音声を逆方向にフィルタリングすることで得られる。

$$res_{LP}(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i s(n-i)$$

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。ポインタ *pSrcSpch*、*pSrcWgtLpc1*、*pSrcWgtLpc2*、*pSrcQLpc*、*pSrcSynFltState*、*pSrcWgtFltState*、*pDstImpulseResponse*、*pDstLpResidual*、または *pDstAdptTarget* が NULL。

AdaptiveCodebookSearch_GSMAMR

アダプティブ・コードブック検索を実行する。

```
IppStatus ippsAdaptiveCodebookSearch_GSMAMR_16s(const Ipp16s *
    pSrcTarget, const Ipp16s *pSrcImpulseResponse, Ipp16s *
    pSrcOpenLoopLag,
```

```
Ipp16s * pValResultPrevIntPitchLag, Ipp16s * pSrcDstExcitation,
Ipp16s * pResultFracPitchLag, Ipp16s * pResultAdptIndex,
Ipp16s * pDstAdptVector, Ipp16s subFrame, IppSpchBitRate mode);
```

引数

pSrcTarget	アダプティブ・ターゲット信号ベクトル [40] へのポインタ（Q15.0）。8 バイト境界にアライメントする必要がある。
pSrcImpulseResponse	重み付き合成フィルタのインパルス応答へのポインタ。Q3.12 で表現され、要素の数は 40 である。8 バイト境界にアライメントする必要がある。
pSrcOpenLoopLag	オープン・ループのピッチ・ラグのベクトル [2] へのポインタ。5.15 と 4.75 Kbps フレームでは、1 つのラグが推定されるため、最初のベクトル要素にのみ有効なラグの値が格納される。その他すべてのビット・レートでは、両方のベクトル要素に有効なピッチ・ラグの値が格納される。
pValResultPrevIntPitchLag	直前の整数ピッチ・ラグへのポインタ。
pSrcDstExcitation	励振ベクトル [194] へのポインタ。要素 0 から 153 には、過去の励振のサンプルが格納されている（Q15.0）。要素 154 から 193 には、推定残留信号のサンプルが格納されているが、推定残留信号はサブフレームの長さが整数クローズ・ループ・ピッチ推定を超えた場合にのみ使用される。
pValResultPrevIntPitchLag	現在の整数ピッチ・ラグへのポインタ。
pSrcDstExcitation	更新された励振ベクトル [194] へのポインタ。要素 0 から 153 には、過去の励振が格納される（Q15.0）。要素 154 から 193 には、アダプティブ・コードブック・ベクトル *v* の 40 個のサンプルが格納される。
pResultFracPitchLag	アダプティブ・コードブック検索で取得した非整数ピッチ・ラグへのポインタ。
pResultAdptIndex	コードされたクローズ・ループ・ピッチのインデックスへのポインタ。
pDstAdptVector	Q15.0 のアダプティブ・コードブック・ベクトル（40 サンプル）へのポインタ。
subFrame	サブフレームのインデックス。

`mode` ビット・レート指定子。有効な値は、`IPP_SPCHBR_4750` から `IPP_SPCHBR_12200` である。

説明

関数 `ippsAdaptiveCodebookSearch_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、アダプティブ・コードブック検索を実行する。アダプティブ・コードブック検索は、クローズ・ループ・ピッチ検索とアダプティブ励振ベクトルの計算で構成される。アダプティブ励振ベクトルは、クローズ・ループ・ピッチ検索で取得した非整数ピッチ・ラグで過去の励振を補間して取得する。アダプティブ・コードブックは、各サブフレームで検索される。アダプティブ・コードブック検索手順を次に説明する。

1. クローズ・ループ・ピッチ分析は、サブフレームを基準にオープン・ループ・ピッチ推定 T_{op} の近傍で実行される。1 番目と 3 番目のサブフレーム（5.15 と 4.75 Kbps モードの場合は 1 番目のサブフレーム）では、近傍検索はレートに依存している。12.2 Kbps フレームでは、検索範囲は T_{op} ±3 であり、18…143 で境界が定められる。5.15 または 4.75 Kbps フレームでは、検索範囲は T_{op} ±5 であり、20…143 で境界が定められる。その他すべてのビット・レートでは、検索範囲は T_{op} ±3 であり、20…143 で境界が定められる。2 番目と 4 番目のサブフレームでは、近傍検索は T_1（直前のサブフレームで最も近い整数から非整数ピッチ・ラグ）を囲む範囲である。近傍境界はレートに依存する。12.2 Kbps フレームでは、検索範囲は [T_1 - 5 3/6, T_1 + 4 3/6] 、7.95 Kbps フレームでは、[T_1 - 10 2/3, T_1 + 9 2/3] 、10.2 または 7.40 Kbps フレームでは、[T_1 - 5 2/3, T_1 + 4 2/3] である。その他すべてのビット・レートでは、検索範囲は [T_1 - 5, T_1 + 4] である。

2. 最適な整数ピッチ検索は、元の音声と合成音声の間の重み付けされた平均 2 乗誤差を最小限にする。これを実現するため、次の式で得られる正規化された相互相関を最大限にする。

$$R(k) = \frac{\sum_{n=0}^{39} x(n)y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n)y_k(n)}}$$

ここで、$x(n)$ はターゲット信号を示す。$y_k(n)$ は、遅延 k で過去にフィルタリングされた励振を示す（過去の励振は、インパルス応答 $h(n)$ でたたみ込まれる）。たたみ込み $y_k(n)$ は、検索範囲内の 1 番目の遅延 t_{min} で計算される。範囲 $k = t_{min} + 1,\ldots, t_{max}$ 内のその他の遅延は、次の再帰関係を使用して更新される。

$$y_k(n) = y_{k-1}(n-1) + u(-k)h(n)$$

ここで、$u(n)$, n = -154,…,39 は励振ヒストリ・バッファである。検索を単純化するために、予測残留信号は $u(n)$, n = 0,…39 にコピーされ、すべての遅延で有効にされる。

3. 非整数ピッチ検索は、$R(k)$　を補間し、その最大値を検索することで行われる。非整数遅延補間は、±23 で切り捨てられ、±24 でゼロでパディングされたハミング窓関数 *sinc* に基づいて FRI フィルタ b_{24} を使用することで得られる。

$$R_t(k) = \sum_{i=0}^{3} R(k-i)b_{24}(t+6i) + \sum_{i=0}^{3} R(k+1+i)b_{24}(6-t+6i)$$

ここで、t は遅延補間に対応する。12.2 Kbps のビット・レートでは、解像度 1/6 を使用すると、遅延は -1/2 から 1/2 となる。それ以外の場合は、解像度 1/3 を使用して非整数は -2/3 から 2/3 となる。

4. アダプティブ・コードブック・ベクトル $v(n)$ は、過去の励振信号 $u(n)$ を整数遅延 k と非整数遅延 t で補間することで計算される。

$$v(n) = \sum_{i=0}^{9} u(n-k-i)b_{60}(t+6i) + \sum_{i=0}^{9} u(n-k+1+i)b_{60}(6-t+6i)$$

補間フィルタは、±59 で切り捨てられ、±60 でゼロでパディングされたハミング窓関数 *sinc* に基づいている。

5. 1 番目と 3 番目のサブフレーム（4.75 と 5.15 Kbps モードの場合は 1 番目のサブフレーム）では、ピッチ・ラグのビット割り当てはレートに依存している。12.2 Kbps のフレームでは、ピッチ・ラグは 9 ビットでエンコードされる。その他すべてのビット・レートでは、ピッチ・ラグは 8 ビットでエンコードされる。2 番目と 4 番目のサブフレームの 12.2 と 7.95 Kbps モードでは、ピッチは 6 ビットでエンコードされ、10.2 と 7.4 kbps モードでは 5 ビットでエンコードされる。その他すべてのモードでは、4 ビットでエンコードされる。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsBadArgErr`	エラー。ポインタ *pSrcTarget*、*pSrcImpulseResponse*、*pSrcOpenLoopLag*、*pValResultPrevIntPitchLag*、

	pSrcDstExcitation、*pResultFracPitchLag*、*pResultAdptIndex*、または *pDstAdptVector* が NULL。
ippStsRangeErr	エラー。*mode* が列挙型 IppSpchBitRate の有効な要素でない。
ippStsSizeErr	エラー。*subFrame* が [0, 3] の範囲にない。

AdaptiveCodebookDecode_GSMAMR

アダプティブ・コードブック・パラメータをデコードする。

```
IppStatus ippsAdaptiveCodebookDecode_GSMAMR_16s(Ipp16s valAdptIndex,
    Ipp16s * pValResultPrevIntPitchLag, Ipp16s * pValResultLtpLag,
    Ipp16s * pSrcDstExcitation, Ipp16s * pResultIntPitchLag,
    Ipp16s * pDstAdptVector, Ipp16s subFrame, Ipp16s bfi,
    Ipp16s inBackgroundNoise, Ipp16s voicedHangover, IppSpchBitRate
    mode);
```

引数

valAdptIndex	アダプティブ・コードブックのインデックス。
pValResultPrevIntPitchLag	直前の整数ピッチ・ラグへのポインタ。出力引数としても使用される。
pValResultLtpLag	LTP-Lag 値へのポインタ。出力引数としても使用される。
pSrcDstExcitation	励振ベクトル [194] へのポインタ。要素 0 から 153 には、過去の励振が格納されている（Q15.0）。要素 154 から 193 は、サブフレームの長さがピッチ・ラグを超えたときにバッファとして使用される。 出力では、要素 154 から 193 には、アダプティブ・コードブック・ベクトルが格納される。
pResultIntPitchLag	整数ピッチへのポインタ。
pDstAdptVector	Q15.0 のアダプティブ・コードブック・ベクトル（40 サンプル）へのポインタ。
subFrame	サブフレームのインデックス。

bfi	不良フレームのインジケータ。「0」は良いフレームを意味する。それ以外の値は不良フレームを意味する。
inBackgroundNoise	直前のフレームにバックグラウンド・ノイズが含まれており、エネルギー・レベルの小さい変更しか表示されない場合、このフラグはセットされる。
voicedHangover	フレームが音声を含むと見なされてからの時間を監視するのに使用されるカウンタ。
mode	ビット・レート指定子。有効な値は、IPP_SPCHBR_4750 から IPP_SPCHBR_12200 である。

説明

関数 ippsAdaptiveCodebookDecode_GSMAMR は、ippsc.h ファイルで宣言される。この関数は、エンコーダによって送信されたアダプティブ・コードブック・パラメータをデコードし、アダプティブ・コードブック・ベクトルを補間するために適用する。受信したフレームからエラーが検出された場合、直前に受信したパラメータを使用して、現在のフレームのパラメータを概算する。アダプティブ・コードブック・ベクトルの補間は、概算したパラメータのセットで行われる。アダプティブ・コードブック・ベクトルは、各サブフレームでデコードされる。このプリミティブの詳細を以下に説明する。

1. 現在のフレームでエラーが検出されなかった場合、アダプティブ・コードブック・インデックスから整数および非整数ピッチ・ラグが抽出される。

2. エラーが検出された場合、直前の整数ピッチまたは *LTP-Lag* から整数ピッチを使用し、非整数ピッチはゼロに設定される。*LTP-Lag* 値は、直前のフレームにある 4 番目のサブフレームの整数ピッチで置換されるか (12.2 Kbps モードの場合)、または最後に正常に受信した値に基づいて変更された値で置換される (その他のモードの場合)。

3. 項 13.4.3 で説明された同じアダプティブ・コードブックの補間操作を適用して、アダプティブ・コードブック・ベクトルを取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。ポインタ *pValResultPrevIntPitchLag*、*pValResultLtpLag*、*pSrcDstExcitation*、*pResultIntPitchLag*、または *pDstAdptVector* が NULL。

`ippStsRangeErr`	エラー。*mode* が列挙型 `IppSpchBitRate` の有効な要素でない。
`ippStsSizeErr`	エラー。*subFrame* が [0, 3] の範囲にない。

AdaptiveCodebookGain_GSMAMR

アダプティブ・コードブック・ベクトルのゲインと、フィルタリングされたコードブック・ベクトルを計算する。

```
IppStatus ippsAdaptiveCodebookGain_GSMAMR_16s (const Ipp16s *
    pSrcAdptTarget, const Ipp16s* pSrcFltAdptVector, Ipp16s*
    pResultAdptGain);
IppStatus ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s (const Ipp16s*
    pSrcAdptTarget, const Ipp16s* pSrcFltAdptVector, Ipp16s*
    pResultAdptGain, Ipp16s* pResultAdptGainCoeffs);
```

引数

pSrcAdptTarget	アダプティブ・ターゲット信号ベクトル [40] へのポインタ。
pSrcFltAdptVector	Q12. のフィルタリングされたアダプティブ・コードブック・ベクトル [40] へのポインタ。
pResultAdptGain	Q14 の出力アダプティブ・コードブック・ゲイン g_p へのポインタ。
pResultAdptGainCoeffs	Q15 の出力ゲイン係数ベクトル [4] へのポインタ。

説明

関数 `ippsAdaptiveCodebookGain_GSMAMR` と `ippsAdaptiveCodebookGainCoeffs_GSMAMR` は、`ippsc.h` ファイルで宣言される。

ippsAdaptiveCodebookGain_GSMAMR_16s。この関数は、次の式に従ってアダプティブ・コードブック・ゲイン g_p を計算する。

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)}$$

範囲は、$0 \le g_p \le 1.2$, in Q14 である。

ここで、*x* はアダプティブ・ターゲット信号ベクトルである。*y* は、フィルタリングされたアダプティブ・コードブック・ベクトルである。

ippsAdaptiveCodebookGain_G729 関数も参照のこと。

ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s。この関数は、アダプティブ・コードブック・ゲイン g_p を ippsAdaptiveCodebookGain_GSMAMR_16s 関数と同じ方法で計算し、次の公式から得られる異なる表現を返す。

$$g_p = \frac{c_{xy} 2^{\exp_{xy}}}{c_{yy} 2^{\exp_{yy}}}, 1/2 \le |c_{xy}|, |c_{yy}| < 1, Q15$$

正規化された分子と分母の仮数 c_{xy}、c_{yy} および指数 exp_{xy}、exp_{yy} は、*pResultAdptGainCoeffs* ベクトルに返される。

pResultAdptGainCoeffs[0] = $c_{yy,}$

pResultAdptGainCoeffs[1] = $exp_{yy,}$

pResultAdptGainCoeffs[2] = $c_{xy,}$

pResultAdptGainCoeffs[3] = exp_{xy}

$c_{xy} < 4$ の場合、ゼロ・ゲインが返される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcAdptTarget*、*pSrcFltAdptVector*、*pResultAdptGain*、または *pResultAdptGainCoeffs* が NULL。

固定コードブック検索

この項では、次の関数を実行するプリミティブを含む、固定コードブックに関するプリミティブについて説明する。

- 固定（代数）コードブック検索
- 固定コードブック・ベクトルのデコード

AlgebraicCodebookSearch_GSMAMR

代数コードブックを検索する。

```
IppStatus ippsAlgebraicCodebookSearch_GSMAMR_16s(Ipp16s valIntPitchLag,
    Ipp16s valBoundQAdptGain, const Ipp16s * pSrcFixedTarget,
    const Ipp16s * pSrcLtpResidual, Ipp16s * pSrcDstImpulseResponse,
    Ipp16s * pDstFixedVector, Ipp16s * pDstFltFixedVector,
    Ipp16s * pDstEncPosSign, Ipp16s subFrame, IppSpchBitRate mode);
IppStatus ippsAlgebraicCodebookSearchEX_GSMAMR_16s(Ipp16s valIntPitchLag,
    Ipp16s valBoundQAdptGain, const Ipp16s * pSrcFixedTarget,
    const Ipp16s * pSrcLtpResidual, Ipp16s * pSrcDstImpulseResponse,
    Ipp16s* pDstFixedVector, Ipp16s * pDstFltFixedVector, Ipp16s *
    pDstEncPosSign, Ipp16s subFrame, IppSpchBitRate mode, Ipp32s *
    pBuffer);
```

引数

valIntPitchLag	このサブフレームで最も近い整数ピッチ・ラグ *T* からこのサブフレームのクローズ・ループ非整数ピッチ・ラグ。クローズ・ループ・ピッチ検索ルーチンで計算される。
valBoundQAdptGain	有界量子化アダプティブ・コードブック・ゲイン。ETSI GSM 06.90 規格のフィルタ $F_E(z)$ のパラメータ β で示される。MR122 モードでは、この値は現在のサブフレームの有界量子化ピッチ・ゲインとなる。その他のモードでは、直前のサブフレームの有界量子化ピッチ・ゲインとなる。この値は、Q1.14 で表現される。
pSrcFixedTarget	固定ターゲット信号ベクトル [40] $x_2(n)$ へのポインタ。固定コードブック・ベクトルの検索に使用され、Q15.0 で表現される。また、8 バイト境界にアライメントする必要がある。

pSrcLtpResidual	長期予測の残留信号ベクトル [40] $res_{LTP}(n)$ へのポインタ。Q15.0 で表現される。
pSrcDstImpulseResponse	重み付け合成フィルタのインパルス応答ベクトル [40] へのポインタ。Q3.12 で表現される。また、8 バイト境界にアライメントする必要がある。出力では、更新されたインパルス応答ベクトル [40] を指す。更新されたベクトルは、元のインパルス応答 $h(n)$ をプレフィルタ $F_E(z)$ でフィルタリングして得られる。これは、Q3.12 で表現される。
pDstFixedVector	固定コードブック・ベクトル [40] $c(n)$ へのポインタ。Q2.13 で表現される。
pDstFltFixedVector	フィルタリングされた固定コードブック・ベクトル [40] $z(n)$ へのポインタ。このベクトルは、インパルス応答を固定コードブック・ベクトルでたたみ込むことで得られる。Q2.13 で表現される。
pDstEncPosSign	エンコードされた位置と最適なパルスの符号を格納するバッファへのポインタ。要素の数は 10 である。12.2 Kbps モードでは、10 個のショート・ワードを使用してエンコードの結果を格納する。10.2 Kbps モードでは、7 個のショート・ワードが使用される。その他のモードでは、2 個のショート・ワードが使用される。
subFrame	範囲が 0 から 3 のサブフレーム・インデックス。
mode	ビット・レート指定子。有効な値は、`IPP_SPCHBR_4750` から `IPP_SPCHBR_12200` である。
pBuffer	長さが 1K の内部作業バッファへのポインタ。

説明

関数 `ippsAlgebraicCodebookSearch_GSMAMR` と `ippsAlgebraicCodebookSearchEX_GSMAMR` は、`ippsc.h` ファイルで宣言される。これらの関数は、重み付け入力音声と重み付け合成音声の間の平均 2 乗誤差を最小限にすることで、代数コードブックを検索する。取得した固定コードブック・ベクトルは、重み付け合成フィルタでフィルタリングされる。最適なパルスの位置と符号は、GSM06.90 仕様に基づいてそれぞれエンコードされる。代数コードブック検索は、各サブフレームで実行される。

これら 2 つの関数の違いは次のとおりである。
`ippsAlgebraicCodebookSearchEX_GSMAMR` は、*`pBuffer`* が指す内部作業バッファを使用する。これは、ユーザによって割り当てられるが、`ippsAlgebraicCodebookSearch_GSMAMR` ではこの内部作業バッファをスタックに割り当てる。

代数検索の詳細を以下に説明する。

1. 固定コードブック検索を行う前に、クローズ・ループ・ピッチの整数部がサブフレームの長さより小さい場合、インパルス応答 $h(n)$ をプレフィルタ $F_E(z)$ でフィルタリングする。

   $$h(n) = h(n) - \beta h(n-T) \ , \ n = T,..., 39$$

   ここで、β は有界の量子化されたアダプティブ・ゲイン、T はクローズ・ピッチの整数部である。

2. 次の式に従って、逆方向でフィルタリングされたターゲット・ベクトル **d** を計算する。

   $$d(n) = \sum_{i=n}^{39} x_2(i) \times h(i-n) \ , \ 0 \le n \le 39$$

   ここで、$x_2(n)$ は固定コードブック検索で使用する固定ターゲット信号である。

3. 12.2 Kbps および 10.2 Kbps モードでは、パルスの大きさを事前に設定する際に使用する信号 $b(n)$ が計算される。

   $$b(n) = \frac{res_{LTP}(n)}{\sqrt{\sum_{i=0}^{39} res_{LTP}(i) res_{LTP}(i)}} + \frac{d(n)}{\sqrt{\sum_{i=0}^{39} d(i) d(i)}} \ , \ n = 0,..., 39$$

   その他のモードでは、$b(n)$ は信号 $d(n)$ と等しい。

4. 各モードの対称 Toepliz 行列 Φ を計算する。要素は次の式で計算される。

   $$\Phi(i, j) = \sum_{n=j}^{39} h(n-i) \times h(n-j) \ , i \le j \ , 0 \le i \le 39$$

5. 各モードの最適なパルスの位置を検索する。合成検索技術による非網羅的分析は、12.2 Kbps、10.2Kbps、7.95Kbps、7.4Kbps、6.70 Kbps モードで使用される。合成検索技術による網羅的分析は、その他のモードで使用される。詳細は、GSM 06.90 (条項 5.7) を参照のこと。

intel®

6. 最適なパルスの位置に基づいて固定コードブック・ベクトル $c(n)$ を構成する。次に、$h(n)$ でたたみ込み、フィルタリングされた固定コードブック・ベクトル $z(n)$ を取得する。

$$z(n) = \sum_{i=0}^{39} c(i) \times h(n-i) \quad , \quad 0 \le \quad n \le \quad 39$$

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。任意の入力または出力ポインタが NULL。
ippStsRangeErr	エラー。*mode* が列挙型 IppSpchBitRate の有効な要素でない。

FixedCodebookDecode_GSMAMR

固定コードブック・ベクトルを
デコードする。

```
IppStatus ippsFixedCodebookDecode_GSMAMR_16s(const Ipp16s *
    pSrcFixedIndex, Ipp16s * pDstFixedVector, Ipp16s subFrame,
    IppSpchBitRate mode);
```

引数

pSrcFixedIndex	固定コードブック・インデックス・ベクトルへのポインタ。12.2 Kbps モードではベクトルの長さは 10、10.2 Kbps では 7 である。それ以外のモードでは、長さは 2 である。
pDstFixedVector	固定コードブック・ベクトル [40] へのポインタ。
subFrame	サブフレームのインデックス。
mode	ビット・レート指定子。有効な値は、IPP_SPCHBR_4750 から IPP_SPCHBR_12200 である。

説明

関数 ippsFixedCodebookDecode_GSMAMR は、ippsc.h ファイルで宣言される。この関数は、受信した固定コードブック・インデックスから固定コードブック・ベクトルをデコードする。

intel®

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。ポインタ *pSrcFixedIndex* または *pDstFixedVector* が NULL。
ippStsRangeErr	エラー。*mode* が列挙型 IppSpchBitRate の有効な要素でない。
ippStsSizeErr	エラー。*subFrame* が [0, 3] の範囲にない。

断片的な送信（DTX）

この項では、次の関数を実行するプリミティブを含む、断片的な送信（DTX）に関するプリミティブを説明する。

- VAD オプション 2 前の入力信号のプリエンファシス計算
- オプション 1 の VAD 決定関数
- オプション 2 の VAD 決定関数
- SID フレームのパラメータの抽出
- DTX ハンドラ
- DTX バッファリング

これらのプリミティブについて以下に説明する。

Preemphasize_GSMAMR

VAD オプション 2 の入力信号の
プレエンファシスを計算する。

```
IppStatus ippsPreemphasize_GSMAMR_16s (Ipp16s gamma, const Ipp16s
    *pSrc, Ipp16s *pDst, int len, Ipp16s* pMem);
```

引数

gamma	Q15 のフィルタ係数。
pSrc	Q0 のソース・ベクトルへのポインタ。
pDst	Q0 のデスティネーション・ベクトルへのポインタ。
len	ソースおよびデスティネーション・ベクトルの要素の数。

pMem	フィルタ・メモリ [1] へのポインタ。

説明

関数 ippsPreemphasize_GSMAMR は、ippsc.h ファイルで宣言される。この関数は、音声アクティブティ検出（VAD）オプション 2 の周波数領域変換を行う前に入力信号のプリエンファシスを計算する。関数 ippsPreemphasize_GSMAMR は、ippsPreemphasize_G729A 関数と同じ操作を行うが、GSM-AMR トランスコード規格で必要な精度を送信するため、操作の順序が異なる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、pDst または *pMem* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

VAD1_GSMAMR

VAD オプション 1 に対応する VAD 機能を実装する。

```
IppStatus ippsVAD1_GSMAMR_16s(const Ipp16s pSrcSpch, IppGSMAMRVad1State
    * pValResultVad1State, Ipp16s * pResultVadFlag, Ipp16s maxHpCorr,
    Ipp16s toneFlag);
```

引数

pSrcSpch	Q0 の入力音声信号 [160] へのポインタ。
pValResultVad1State	入力では、VAD オプション 1 のヒストリ変数へのポインタ。出力では、更新された VAD オプション 1 のヒストリ変数を指す。構造体 IppGSMAMRVad1State は、次のように定義される。
pResultVadFlag	このフレームの VAD フラグへのポインタ。フラグが「1」に設定されている場合、送信する信号があることを示す。フラグが「0」に設定されている場合、このフレームに送信する信号がないことを示す。

`maxHpCorr`	直前のフレームの *best_corr_hp* 値。この値は、ハイパス・フィルタリングされた相関の正規化された最大値である。また、この値はオープン・ループ・ピッチの検索関数の出力である。
`toneFlag`	非常に強力な周期的成分を含む情報トーンまたは信号の有無を示すトーン・フラグ。この値は、オープン・ループ・ピッチの検索関数の出力である。

説明

関数 `ippsVAD1_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、*ETSI GSM 06.94* の VAD オプション 1 に対応する VAD 機能を実装する。この関数を使用して、送信する信号（音声、音楽、または情報トーンなど）が、各 20ms フレームごとに含まれているかを示す。構造体 `IppGSMAMRVad1State` は、VAD オプション 1 のヒストリ変数を含む。これらの変数は、エンコーダの開始前に初期化され、この関数でのみ更新することができる。実装に関する詳細は、*ETSI GSM 06.94* の VAD オプション 1 の仕様を参照のこと。

構造体 `IppGSMAMRVad1State` の定義

typedef struct{	**説明**
`Ipp16s pPrevSignalLevel[9];`	*level*[*n*] 直前のフレームの信号レベル・ベクトル。
`Ipp16s pPrevSignalSublevel[9];`	直前のフレームの中間信号サブレベル・ベクトル。
`Ipp16s pPrevAverageLevel[9];`	直前のフレームの平均信号レベル・ベクトル *ave_level*[*n*]。
`Ipp16s pBkgNoiseEstimate[9];`	直前のフレームのバックグラウンド・ノイズ推定ベクトル *back_est*[*n*]。
`Ipp16s pFifthFltState[6];`	フィルタ・バンクの 3 つの 5 番目フィルタのヒストリ・ステート。
`Ipp16s pThirdFltState[5];`	フィルタ・バンクの 5 つの 3 番目フィルタのヒストリ・ステート。
`Ipp16s burstCount;`	音声バースト・サイズをカウントするバースト・カウンタ *burst_count*。VAD ハングオーバー付加で使用される。
`Ipp16s hangCount;`	VAD ハングオーバー付加で使用されるハング・カウンタ *hang_counter*。
`Ipp16s statCount;`	バックグラウンド・ノイズ推定で使用する固定カウンタ変数 *stat_count*。
`Ipp16s vadReg;`	中間 VAD 決定を示す値。
`Ipp16s complexHigh;`	中間複素数信号決定で使用する *complex_high* 値。
`Ipp16s complexLow;`	中間複素数信号決定で使用する *complex_low* 値。
`Ipp16s complexHangTimer;`	複素数アクティビティ推定でハングオーバー・イニシエータとして使用する *complex_hang_timer*。

構造体 IppGSMAMRVad1State の定義

typedef struct{	説明
Ipp16s complexHangCount;	VAD ハングオーバー付加でハングオーバー・カウンタとして使用する *complex_hang_count*。
Ipp16s complexWarning;	*complex_warning* フラグ。
Ipp16s corrHp;	*best_corr_hp* のハイパス・フィルタリングされた値。
Ipp16s pitchFlag;	母音発音および周期的な信号の有無を示すピッチ・フラグ。
}IppGSMAMRVad1State.	

注 : **VAD オプション 1 のヒストリ変数の初期化。**
エンコーダがリセットされる際に、pPrevSignalLevel、pPrevSignalSublevel、pPrevAverageLevel ベクトルに含まれたすべての要素は 150 に、corrHp は 13106 に設定する必要がある。その他すべての変数は、ゼロに初期化する必要がある。
これらのヒストリ変数の詳細な使用方法については、*ETSI GSM 06.94* を参照のこと。

戻り値

ippStsNoErr エラーなし。

ippStsBadArgErr エラー。任意の入力または出力ポインタが NULL。

VAD2_GSMAMR

VAD オプション 2 に対応する VAD 機能を実装する。

```
IppStatus ippsVAD2_GSMAMR_16s(const Ipp16s * pSrcSpch,
    IppGSMAMRVad2State * pValResultVad2State, Ipp16s * pResultVadFlag,
    Ipp16s ltpFlag);
```

引数

pSrcSpch Q0 の入力音声フレームへのポインタ。長さは 160。

pValResultVad2State 入力では、VAD オプション 2 のヒストリ変数へのポインタ。出力では、更新された VAD オプション 2 のヒストリ変数を指す。構造体 IppGSMAMRVad2State の定義は以下に示す。

pResultVadFlag ブール・フラグ *VAD_flag* へのポインタ。フラグが「1」に設定されている場合、送信する信号があることを示す。フラグが「0」に設定されている場合、このフレームに送信する信号がないことを示す。

ltpFlag	*GSM 06.94* 式（4.24）の *LTP_flag* 値。この値は、長期予測と定数しきい値 *LTP_THLD* を比較して生成される。

説明

関数 ippsVAD2_GSMAMR は、ippsc.h ファイルで宣言される。この関数は、*ETSI GSM 06.94* の VAD オプション 2 に対応する VAD 機能を実装する。この関数を使用して、送信する信号（音声、音楽、または情報トーンなど）が、各 20ms フレームごとに含まれているかを示す。構造体 IppGSMAMRVad2State は、VAD オプション 2. のヒストリ変数を含む。

詳細は、*ETSI GSM 06.94* の VAD オプション 2 の仕様を参照のこと。

構造体 **IppGSMAMRVad2State** の定義

typedef struct{	**説明**
Ipp32s pEngyEstimate[16];	現在のハーフフレームのチャネル･エネルギー推定ベクトル E_{ch}。*ETSI GSM 06.94 (4.4)* の式に基づいて、直前のハーフフレームで計算される。
Ipp32s pNoiseEstimate[16];	現在のハーフフレームのチャネル・ノイズ推定ベクトル E_{ch}。*ETSI GSM 06.94 (4.26)* の式に基づいて、直前のハーフフレームで計算される。
Ipp16s pLongTermEngyDb[16];	チャネルの平均長期スペクトル推定ベクトル $\bar{E}_{dB}$ 。*ETSI GSM 06.94 (4.20)* の式に基づいて、直前のハーフフレームで計算される。
Ipp16s preEmphasisFactor;	プリエンファシス係数 ζ_p。*ETSI GSM 06.94 (4.1)* の式に基づいて、入力音声信号をプリエンファシスするために使用する。
Ipp16s updateCount;	バックグラウンド・ノイズ更新決定ロジックで使用する *update_cnt* 値。
Ipp16s lastUpdateCount;	バックグラウンド・ノイズ更新決定ロジックで使用する *last_update_cnt* 値。
Ipp16s hysterCount;	バックグラウンド・ノイズ更新決定ロジックで使用する *hyster_cnt* 値。
Ipp16s prevNormShift;	精度を高くし、FFT 変換でオーバーフローを回避するために、直前のハーフフレームの入力音声を正規化した際にシフトされたビット。
Ipp16s shiftState;	直前のハーフフレームでシフトが行われたかどうかを示すシフトのステート・フラグ。
Ipp16s forcedUpdateFlag;	バックグラウンド・ノイズ更新決定の強制更新ロジックによる結果を示す *fupdate_flag* 値。
Ipp16s ltpSnr;	直前のハーフフレームの長期的なピーク信号ノイズ比 SNR_p。VAD 決定の反応をキャリブレートするために使用する。
Ipp16s variabFactor;	直前のハーフフレームの可変係数 ψ。バックグラウンド・ノイズ推定の変動性を示す。この値は、*ETSI GSM 06.94 (4.13)* の式に従って更新される。
Ipp16s negSnrBias;	直前のハーフフレームの負の SNR 感度バイアス係数 μ。

構造体 **IppGSMAMRVad2State** の定義

typedef struct{	説明
Ipp16s burstCount;	10ms ハーフフレームの VAD 決定で使用されるバースト・カウンタ *b*(*m*)。
Ipp16s hangOverCount;	10ms ハーフフレームの VAD 決定で使用されるハングオーバー・カウンタ *h*(*m*)。
Ipp32s frameCount;	ハーフフレーム・カウンタ。
}IppGmrVad2State;	

注 : エンコーダーがリセットされる際に、構造体に含まれたすべての要素をゼロに初期化する必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。任意の入力または出力ポインタが NULL。

EncDTXSID_GSMAMR

SID フレームのパラメータを抽出する。

```
IppStatus ippsEncDTXSID_GSMAMR_16s(const Ipp16s * pSrcLspBuffer, const
    Ipp16s * pSrcLogEnergyBuffer, Ipp16s * pValResultLogEnergyIndex,
    Ipp16s * pValResultDtxLsfRefIndex, Ipp16s * pSrcDstQLsfIndex,
    Ipp16s * pSrcDstPredQErr, Ipp16s * pSrcDstPredQErrMR122, Ipp16s
    sidFlag);
```

引数

pSrcLspBuffer	VAD = 0 とマークされた 8 つの連続フレームの LSP 係数へのポインタ。Q0.15 で表現され、長さは 80。
pSrcLogEnergyBuffer	無声とマークされた 8 つの連続フレームのログ・エネルギー係数へのポインタ。Q5.10 で表現され、長さは 8。
pValResultLogEnergyIndex	最後のフレームのログ・エネルギー・インデックスへのポインタ。Q2.13 で表現される。 出力では、現在のフレームのログ・エネルギー・インデックスを指す。Q2.13 で表現される。

pValResultDtxLsfRefIndex	最後のフレームの LSF 量子化リファレンス・インデックスへのポインタ。出力では、現在の DTX フレームの LSF 量子化リファレンス・インデックスを指す。
pSrcDstQLsfIndex	最後のフレームの LSF 残留量子化インデックスへのポインタ。長さは 3。出力では、現在のフレームの LSF 残留量子化インデックスを指す。長さは 3。
pSrcDstPredQErr	12.2 Kbps 以外のモードでは、直前の 4 つの固定ゲイン予測エラーへのポインタ。Q5.10 で表現され、長さは 4。12.2 Kbps 以外のモードにおける出力では、更新された固定ゲイン予測エラーを指す。Q5.10 で表現され、長さは 4。
pSrcDstPredQErrMR122	12.2 Kbps モードでは、4 つ前までの固定ゲイン予測エラーへのポインタ。Q5.10 で表現され、長さは 4。12.2 Kbps のモードにおける出力では、更新された固定ゲイン予測エラーを指す。Q5.10 で表現され、長さは 4。
sidFLag	現在のフレームの SID フラグ。1 に設定すると、現在のフレームが SID フレームとなり、LSF とエネルギー・パラメータを抽出する。0 に設定すると、LSF と エネルギー・パラメータは直前のフレームからコピーされる。

説明

関数 `ippsEncDTXSID_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、現在のフレームが DTX フレームの場合にのみ呼び出される。この関数は、SID フレームで必要なパラメータ（つまり、LSF 量子化パラメータとエネルギー・インデックス・パラメータ）を抽出する。SID フラグがオフの場合、次の操作を行わずに、すべてのパラメータが最後のフレームからコピーされる。SID フラグがオンの場合、次の操作を行う。

1. ログ・エネルギー・インデックスを計算する。

$$EnLogIndex = \frac{1}{8}\sum_{n=0}^{7} EnLog(t-n) ,$$

ここで、*EnLog*(*t*) は現在のフレームでの log2 スケールのログ・エネルギーである。

2. 12.2 Kbps モードでは、固定ゲイン予測エラーを更新する。その他のモードでは、それぞれログ・エネルギー・インデックスを更新する。

$$PredErr(i) = EnLogIndex$$

$$PredErrMR122(i) = (EnLogIndex)/(20 \times \log 10(2))$$

$i = 0,..., 3$

3. 現在のフレームと過去 7 つのフレームの平均 LSP 係数を計算する。

$$Lsp_{mean}(i) = \frac{1}{8}\sum_{n=0}^{7} Lsp(i-n)$$

4. 平均 LSP 係数を量子化する。

   a. 平均 LSP を LSF に変換する。変換した LSF を並べ替え、重み付け係数と掛ける。

   b. LSF リファレンス・ベクトルを検索する。

$$Lsfref_Index = \min_{index}(\sum_{n=0}^{9}(Lsf(n) - Lsf_ref_{index}(n))^2 \mid index = 0,...,8)$$

   c. LSF 残留信号を取得する。

$$Lsf_residual(i) = Lsf(i) - Lsf_ref_{index}(i) \quad i = 0,...,9$$

   d. 分割バンド・ベクトル量子化メソッドを使用して、LSF 残留信号を量子化する。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsBadArgErr`	エラー。任意の入力または出力ポインタが `NULL`。

EncDTXHandler_GSMAMR

現在のフレームの SID フラグを確認する。

```
IppStatus ippsEncDTXHandler_GSMAMR_16s(Ipp16s * pValResultHangOverCount,
    Ipp16s *pValResultDtxElapsedCount, Ipp16s * pValResultUsedMode,
    Ipp16s * pResultSidFlag, Ipp16s vadFlag);
```

引数

pValResultHangOverCount	DTX ハングオーバー・カウントへのポインタ。初期化するかまたはリセットすると 0 に設定される。出力では、更新した DTX ハングオーバー・カウントを指す。
pValResultDtxElapsedCount	最後の非 DTX フレームから経過したフレーム・カウントへのポインタ。初期化するかまたはリセットすると、0 に設定される。出力では、更新した最後の非 DTX フレームから経過したフレーム・カウントを指す。
pValResultUsedMode	送信モードへのポインタ。入力段階では、モードは 4.75 Kbps から 12.2 Kbps までの範囲のビット・レートである。出力段階では、この値は変更されないか、または DTX フレームのモードに設定される。
pResultSidFlag	出力 SID フラグへのポインタ。「1」は SID フレームを示す。「0」は非 SID フレームを示す。
vadFlag	現在のフレームの VAD フラグ。「1」は、現在のフレームに音声が含まれていること示す。「0」は、現在のフレームに音声が含まれていないことを示す。

説明

関数 `ippsEncDTXHandler_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、現在のフレームの SID フラグを確認し、現在のフレームが DTX エンコードを使用する必要があるかどうかを決定する。

1. 最後の SID フレームから経過したフレーム・カウントを更新する。
   DTX_ElapsedCount = *DTX_ElapsedCount* + 1(Bounded to 0~0x7fff)

2. 現在のフレームの VAD フラグが 1（有声フレーム）の場合、DTX ハングオーバー・カウントは 7 に設定され、この関数は終了する。

3. 現在のフレームの VAD フラグが 0（無声フレーム）で、DTX ハングオーバー・カウントが 0 の場合、このフレームの送信モードは DTX フレーム・モードに設定される。また、最後の DTX フレームから経過したフレーム・カウントは 0 に、SID フラグは 1 に設定される。

4. 現在のフレームの VAD フラグが 0（無声フレーム）で、DTX ハングオーバー・カウントが 0 ではない場合、DTX ハングオーバー・カウントを 1 つ減らし、*DTX_HangOver_Count* + *DTX_ElapsedCount* < 30 の場合、SID フラグは 0 に、送信モードは「MRDTX」に設定される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsBadArgErr`	エラー。任意の入力または出力ポインタが NULL。

EncDTXBuffer_GSMAMR, DecDTXBuffer_GSMAMR

LSP（または LSF）係数と直前のログ・エネルギー係数をバッファする。

```
IppStatus ippsEncDTXBuffer_GSMAMR_16s(const Ipp16s * pSrcSpch,
    const Ipp16s * pSrcLsp, Ipp16s *pValResultUpdateIndex,
    Ipp16s * pSrcDstLspBuffer, Ipp16s * pSrcDstLogEnergyBuffer);
IppStatus  ippsDecDTXBuffer_GSMAMR_16s(const Ipp16s * pSrcSpch,
    const Ipp16s * pSrcLsf, Ipp16s *pValResultUpdateIndex,
    Ipp16s *pSrcDstLsfBuffer, Ipp16s * pSrcDstLogEnergyBuffer);
```

引数

pSrcSpch	Q15.0 の入力音声信号へのポインタ。長さは 160。
pSrcLsp	Q0.15 のこのフレームの LSP へのポインタ。長さは 10。
pSrcLsf	Q0.15 の現在のフレームの LSF 係数へのポインタ。長さは 10。
pValResultUpdateIndex	直前のメモリ更新のインデックスへのポインタ。出力では、現在のメモリ更新のインデックスを指す。このインデックスは、0 から 7 までの値を循環する。

intel®

pSrcDstLspBuffer	直前の 8 つのフレームの LSP 係数へのポインタ。Q0.15 で表現され、長さは 80。出力では、現在のフレームを含む最新の 8 つのフレームの LSP 係数を指す。Q0.15 で表現され、長さは 80。
pSrcDstLogEnergyBuffer	直前の 8 つのフレームのログ・エネルギー係数へのポインタ。Q5.10 で表現され、長さは 8。出力では、現在のフレームを含む最新の 8 つのフレームのログ・エネルギー係数を指す。Q5.10 で表現され、長さは 8。

説明

関数 `ippsEncDTXBuffer_GSMAMR` と `ippsDecDTXBuffer_GSMAMR` は、`ippsc.h` ファイルで宣言される。これらの関数は、LSP（または LSF）係数と直前のログ・エネルギー係数をバッファする。これらの LSP（または LSF）とエネルギー係数は、SID フレームで必要なパラメータを抽出するために使用される。メモリ更新のインデックスは、更新するバッファの箇所を示すことで、メモリのコピーを抑える。ログ・エネルギーは、次のように計算される。

$$EnLog = \log 2\left(\frac{1}{N}\sum_{n=0}^{N-1} s^2(n)\right) ,$$

ここで、N はフレームの長さ、s は入力音声信号である。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。任意の入力または出力ポインタが NULL。

後処理

PostFilter_GSMAMR

合成音声をフィルタする。

```
IppStatus ippsPostFilter_GSMAMR_16s(const Ipp16s * pSrcQLpc,
    const Ipp16s * pSrcSpch, Ipp16s * pValResultPrevResidual,
    Ipp16s * pValResultPrevScalingGain, Ipp16s * pSrcDstFormantFIRState,
    Ipp16s * pSrcDstFormantIIRState, Ipp16s * pDstFltSpch, IppSpchBitRate
    mode);
```

引数

pSrcQLpc	再構築された LP 係数へのポインタ。長さは 44 で、Q3.12 で表現される。
pSrcSpch	現在のフレームの入力音声信号の開始位置へのポインタ。Q15.0 で表現され、長さは 160。
pValResultPrevResidual	エントリ時には、直前のサブフレームのフォルマント・フィルタ の FIR フィルタにおける最後の出力へのポインタ。Q15.0 で表現される。この値は、ティルト補正フィルタの入力である。 終了時には、このサブフレーム（Q15.0）のフォルマント・フィルタの FIR フィルタにおける最後の出力を指す。この値は、ティルト補正フィルタの出力であり、0 に初期化され、この関数でのみ更新することができる。
pValResultPrevScalingGain	直前のサブフレームの最後の信号のスケーリング係数 *b* へのポインタ。Q3.12 で表現される。出力では、このサブフレームの最後の信号のスケーリング係数 *b* を指す。Q3.12 で表現される。
pSrcDstFormantFIRState	フォルマント・フィルタの FIR 部分のステートへのポインタ。Q15.0 で表現され、長さは 10。出力では、更新されたフォルマント・フィルタの FIR 部分のステートを指す。Q15.0 で表現され、長さは 10。
pSrcDstFormantIIRState	フォルマント・フィルタの IIR 部分のステートへのポインタ。Q15.0 で表現され、長さは 10。出力では、更新されたフォルマント・フィルタの IIR 部分のステートを指す。Q15.0 で表現され、長さは 10。
pDstFltSpch	フィルタリングされた音声へのポインタ。Q15.0 で表現され、長さは 160。
mode	ソース送信モード。この関数で有効な値は、`IPP_SPCHBR_4750` から `IPP_SPCHBR_12200` である。

説明

関数 `ippsPostFilter_GSMAMR` は、`ippsc.h` ファイルで宣言される。この関数は、合成音声をフィルタリングして再構成の品質を向上する。このプリミティブは、次の操作を実行する。

1. フォルマント・ポストフィルタの重み付き LP 係数を取得する。

$$H_f(z) = \frac{\hat{A}(z/\gamma_n)}{\hat{A}(z/\gamma_d)}$$

12.2 および 10.2 Kbps モードでは、

$$\gamma_n = 0.7, \quad \gamma_d = 0.75$$

その他のモードでは、

$$\gamma_n = 0.55, \quad \gamma_d = 0.7$$

次に、入力音声を $HF(z) = \hat{A}(z/\gamma_n)$ でフィルタリングする。

2. フォルマント・ポストフィルタのインパルス応答 $h(n)$ を計算し、最初の反射係数を取得する。

$$k'_1 = \frac{r_h(1)}{r_h(0)} \quad , \quad r_h(i) = \sum_{j=0}^{21-i} h(j) \times h(j+i)$$

ティルト係数を計算する。 $\mu = \gamma_t k'_1$

12.2 および 10.2 Kbps モードでは、

$$\gamma_t = \begin{cases} 0.8 & k'_1 > 0, \\ 0 & otherwise \end{cases}$$

その他のモードでは、 $\gamma_t = 0.8$

ティルト補正フィルタを使用して信号をフィルタリングする。

$$H_t(z) = 1 - \mu z^{-1}$$

ティルト補正フィルタの出力は、次の式でフィルタリングされる。

$$HI(z) = \hat{A}(z/\gamma_d)$$

3. アダプティブ・ゲインのスケーリング係数を計算する。

$$\gamma_{sc} = \sqrt{\frac{\sum_{n=0}^{39} \hat{s}(n)}{\sum_{n=0}^{39} \hat{s}_f(n)}}$$

ここで、$\hat{s}(n)$ は合成音声、$\hat{s}_f(n)$ はポストフィルタリングされた音声である。

出力音声は、次の式から得られる。

$$\hat{s}'(n) = \beta_{sc}(n)\hat{s}_f(n)$$

スケーリング係数は、サンプルごとに更新される。

$$\beta_{sc}(n) = 0.9\beta_{sc}(n-1) + 0.1\gamma_{sc}$$

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。任意の入力または出力ポインタが NULL、または入力変数 *mode* が範囲外。

GSM フル・レートに関連する関数

このセクションでは、GSM 06.10、06.11、06.12、06.31、および 06.32 に準拠した音声コーデックの開発に使用できるインテル® IPP 関数について説明する。
これらすべての関数のリストを次の表に示す。

表 9-5　インテル® IPP GSM フル・レートに関連する関数

関数の基本名	操作
RPEQuantDecode_GSMFR	APCM の逆量子化を実行する。
Deemphasize_GSMFR	デエンファシス・フィルタリングを実行する。
ShortTermAnalysisFilter_GSMFR	短期分析フィルタリングを実行する。
ShortTermSynthesisFilter_GSMFR	短期合成フィルタリングを実行する。
HighPassFilter_GSMFR	入力音声信号をハイパス・フィルタでフィルタリングする。
Schur_GSMFR	Schur 再帰を使用して反射係数を推定する。
WeightingFilter_GSMFR	重み付けフィルタを計算する。
Preemphasize_GSMFR	音声信号のプリエンファシスを計算する。

intel®

RPEQuantDecode_GSMFR

APCM の逆量子化を実行する。

```
IppStatus ippsRPEQuantDecode_GSMFR_16s (const Ipp16s *pSrc, Ipp16s ampl,
    Ipp16s amplSfs, Ipp16s *pDst);
```

引数

pSrc	RPE サンプルの入力ベクトル [13] へのポインタ。
ampl	ブロック振幅。
amplSfs	ブロック振幅のスケール係数。
pDst	再構築された出力長期残留信号ベクトル [13] へのポインタ。

説明

関数 ippsRPEQuantDecode_GSMFR は、ippsc.h ファイルで宣言される。この関数は、入力 RPE サンプルの APCM 逆量子化を実行する。再構築された出力長期残留信号ベクトルは次のとおりである。

$$pDst[i] = (2 * pSrc[i] - 7) \cdot \frac{ampl}{2^{amplSfs}}$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsRangeErr	エラー。*amplSfs* がゼロより小さい。

Deemphasize_GSMFR

デエンファシス・フィルタリングを
実行する。

```
IppStatus ippsDeemphasize_GSMFR_16s_I (Ipp16s *pSrcDst, int len,
    Ipp16s *pMem);
```

引数

pSrcDst	入力短期合成信号と出力後処理音声ベクトル [*len*] へのポインタ。
len	入力残留信号と出力音声ベクトルの長さ。
pMem	フィルタ・メモリの要素へのポインタ。

説明

関数 ippsDeemphasize_GSMFR は、ippsc.h ファイルで宣言される。この関数は、入力合成信号のデエンファシスを実行する。この信号は、次の伝達関数を使用したフィルタでフィルタリングされる。

$$H(z) = \frac{1}{1 - \alpha z^{-1}}$$

ここで $\alpha = 0.86$（Q15 の 28180）となる。

フィルタの初期メモリはゼロに設定される。

フィルタリングされた音声信号は 2 倍までスケーリングされ、3 つの最下位ビットが切り捨てられてから *pSrcDst* に格納される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst* または *pMem* が NULL。
ippStsRangeErr	エラー。*len* がゼロ以下。

ShortTermAnalysisFilter_GSMFR

短期分析フィルタリングを実行する。

```
IppStatus ippsShortTermAnalysisFilter_GSMFR_16s_I (const Ipp16s *pRC,
    Ipp16s *pSrcDstSpch, int len, Ipp16s *pMem);
```

引数

pRC	入力反射係数ベクトル [8] へのポインタ。 $a_1, a_2, \ldots, a_8$

pSrcDstSpch	入力前処理音声と出力短期残留信号ベクトル [*len*] へのポインタ。
len	入力音声と出力残留信号ベクトルの長さ。
pMem	フィルタ・メモリ・ベクトル [8] へのポインタ。$m_0, m_1, \ldots, m_7$

説明

関数 ippsShortTermAnalysisFilter_GSMFR は、ippsc.h ファイルで宣言される。この関数は、前処理音声ベクトル $s(n)$ をフィルタリングし、出力短期残留信号ベクトル $r(n)$ に結果を格納する。

$$r_0 = s(n)$$

$$r_i = r_{i-1} + a_i \cdot m_{i-1}, i = 1,\ldots,8$$

$$m_0 = s(n)$$

$$m_i = m_{i-1} + a_i \cdot r_{i-1}, i = 1,\ldots,7$$

$$r(n) = r_8$$

ここで、m_i , $i = 0,\ldots,7$ はフィルタ・メモリ、r_i , $i = 0,\ldots,8$ は再利用可能なローカル・メモリである。

フィルタ・メモリ・ベクトルの初期設定は、ゼロである。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pRC*、*pSrcDstSpch*、または *pMem* が NULL。
ippStsRangeErr	エラー。*len* がゼロ以下。

ShortTermSynthesisFilter_GSMFR

短期合成フィルタリングを実行する。

```
IppStatus ippsShortTermSynthesisFilter_GSMFR_16s (const Ipp16s *pRC,
    const Ipp16s *pSrcResidual, Ipp16s *pDstSpch, int len, Ipp16s
    *pMem);
```

引数

pRC	入力反射係数ベクトル [8] へのポインタ。$a_1, a_2, \ldots, a_8$
pSrcResidual	再構築された入力短期残留信号ベクトル [*len*] へのポインタ。
pDstSpch	出力音声ベクトル [*len*] へのポインタ。
len	入力残留信号と出力音声ベクトルの長さ。
pMem	フィルタ・メモリ・ベクトル [8] へのポインタ。$m_0, m_1, \ldots, m_7$

説明

関数 ippsShortTermSynthesisFilter_GSMFR は、ippsc.h ファイルで宣言される。この関数は、再構築された入力短期残留信号 $r(n)$ をフィルタリングし、出力音声ベクトル $s(n)$ に結果を格納する。

$$s_0 = r(n)$$

$$s_i = s_{i-1} - a_{9-i} \cdot m_{8-i}, i = 1,\ldots,8$$

$$m_{8-i} = m_{7-i} - a_{9-i} \cdot s_i, i = 1,\ldots,7$$

$$s(n) = s_8$$

$$m_0 = s(n)$$

ここで、$m_i, i = 0,\ldots,7$ はフィルタ・メモリ、$s_i, i = 0,\ldots,8$ は再利用可能なローカル・メモリである。

フィルタ・メモリ・ベクトルの初期設定は、ゼロである。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pRC*、*pSrcResidual*、*pMem*、または *pDstSpch* が NULL。
ippStsRangeErr	エラー。*len* がゼロ以下。

HighPassFilter_GSMFR

入力音声信号のハイパス・フィルタリングを実行する。

```
IppStatus ippsHighPassFilter_GSMFR_16s (const Ipp16s *pSrc, Ipp16s
     *pDst, int len, int *pMem);
```

引数

pSrc	ソース・音声ベクトル [*len*] へのポインタ。
pDst	フィルタリングされたデスティネーション・ベクトル [*len*] へのポインタ。
len	ソースおよびデスティネーション・ベクトルの長さ。
pMem	フィルタ・メモリ・ベクトル [2] へのポインタ。

説明

関数 ippsHighPassFilter_GSMFR は、ippsc.h ファイルで宣言される。この関数は、伝達関数に従って入力音声信号をフィルタリングする。

$$H(z) = 0.5 \cdot \frac{1 - z^{-1}}{1 - \alpha z^{-1}}$$

ここで、$\alpha = 0.99899$ である。

フィルタの初期メモリはゼロに設定される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。

ippStsSizeErr	エラー。*len* がゼロ以下。

Schur_GSMFR

Schur 再帰を使用して反射係数を推定する。

```
IppStatus ippsSchur_GSMFR_32s16s (const Ipp32s *pSrc, Ipp16s *pDst,
    int dstLen);
```

引数

pSrc	入力自己相関ベクトル [*dstLen*+1] へのポインタ。
pDst	出力反射係数ベクトル [*dstLen*] へのポインタ。
dstLen	推定する反射係数の数。

説明

関数 ippsSchur_GSMFR は、ippsc.h ファイルで宣言される。この関数は、GSM 06.10 条項 4.2.5 に従って Schur アルゴリズムを実装する。ippsSchur 関数も参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* が 0 以下。

WeightingFilter_GSMFR

重み付けフィルタを計算する。

```
IppStatus ippsWeightingFilter_GSMFR_16s (const Ipp16s *pSrc, Ipp16s *pDst,
    int dstLen);
```

引数

pSrc	入力長期残留信号ベクトル [-5,..,*dstLen*+4] へのポインタ。

intel®

pDst	フィルタリングされた出力ベクトル [*dstLen*] へのポインタ。
dstLen	検索するフィルタリングされた要素の数。

説明

関数 ippsWeightingFilter_GSMFR は、ippsc.h ファイルで宣言される。この関数は、定義済みのタップを使用して、対称 FIR フィルタで入力信号をフィルタリングする。

taps[i]=[-134, -374, 0, 2054, 5741, 8192, 5741, 2054, 0, -374, -134], i=0,..,10

$$dst[n] = \sum_{i=0}^{10} taps[i] \cdot src[n+i-5], n = 0,.., dstlen-1$$

フィルタリングの結果は *pDst* に格納される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*dstLen* がゼロ以下。

Preemphasize_GSMFR

音声信号のプリエンファシスを計算する。

```
IppStatus ippsPreemphasize_GSMFR_16s(const Ipp16s *pSrc, Ipp16s *pDst,
    int *pMem, int len);
```

引数

pSrc	オフセット・フリーの入力音声信号へのポインタ。
pDst	プリエンファシスされた出力信号へのポインタ。
pMem	フィルタメモリの値へのポインタ。
len	入力および出力信号の長さ。

説明

関数 ippsPreemphasize_GSMFR は、ippsc.h ファイルで宣言される。この関数は、差分信号のプリエンファシスの式に従って、入力音声のプリエンファシスを計算する。

$$H(z) = 1 - \gamma z^{-1}$$

$\gamma = -0.86$ および *pSrc*[-1] = *pMem*[0] の場合。

フィルタリングの結果は *pDst* に格納される。メモリ値 *pMem*[0] は、*pSrc*[n-1] で更新される。GSM フル・レート・コーデックでこの関数を正しく使用するために、メモリ値はゼロで初期化される。

関数 ippsPreemphasize_GSMFR は、NR 丸めを実行する（丸めモードを参照）。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* が 0 以下。

G.722.1 に関連する関数

本章では、ITU-T 勧告 G.722.1 に準拠した音声コーデックの開発に使用できるインテル® IPP 関数について説明する。
これらすべての関数のリストを次の表に示す。

表 9-6　G.722.1 に関連するインテル® IPP 関数

関数の基本名	操作
DCTFwd_G722, DCTInv_G722	信号に対する順方向または逆方向の離散コサイン変換（DCT）を計算する。
DecomposeMLTToDCT	MLT 変換入力信号を DCT 入力信号の形式で分解する。
DecomposeDCTToMLT	IDCT 出力信号を MLT 変換出力信号の形式で分解する。
HuffmanEncode_G722	量子化された大きさの包絡線インデックスに対してハフマン・エンコードを実行する。

DCTFwd_G722, DCTInv_G722

信号に対する順方向または逆方向の
離散コサイン変換（DCT）を計算する。

```
IppStatus ippsDCTFwd_G722_16s (const Ipp16s *pSrc, Ipp16s *pDst);
IppStatus ippsDCTInv_G722_16s (const Ipp16s *pSrc, Ipp16s *pDst);
```

引数

pSrc	ソース・ベクトル [320] へのポインタ。
pDst	デスティネーション・ベクトル [320] へのポインタ。

説明

関数 `ippsDCTFwd_G722` と `ippsDCTInv_G722` は、`ippsc.h` ファイルで宣言される。これらの関数は、長さが 320 の順方向と逆方向の離散コサイン変換（DCT）を次の式で計算する。

$$y(m) = \alpha \sum_{n=0}^{319} \cos\left(\frac{\pi}{320} \cdot (n+0.5) \cdot (m+0.5)\right) \cdot x(n)$$ ，順方向 DCT

ここで、$\alpha = 2/320$ である。

$$x(n) = \beta \sum_{m=0}^{319} \cos\left(\frac{\pi}{320} \cdot (m+0.5) \cdot (n+0.5)\right) \cdot y(m)$$ ，逆方向 DCT (IDCT)

ここで、$\beta = 0.81$ である。

これらの式は同じであるが、ビットごとの正確さの必要条件を満たすために異なるスケーリングが適用される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrc* または *pDst* が `NULL`。

DecomposeMLTToDCT

MLT 変換入力信号を DCT 入力信号の形式で分解する。

```
IppStatus ippsDecomposeMLTToDCT_G722_16s(const Ipp16s *pSrcSpch, Ipp16s
    *pSrcDstSpchOld, Ipp16s *pDstSpchDecomposed);
```

引数

pSrcSpch	ソース・ベクトル [320] へのポインタ。
pSrcSpchOld	直前のフレームの音声サンプルのソース / デスティネーション・ベクトル [320] へのポインタ。
pDstSpchDecomposed	デスティネーション・ベクトル [320] へのポインタ。

説明

関数 ippsDecomposeMLTToDCT_G722 は、ippsc.h ファイルで宣言される。この関数は、変調ラップ変換（MLT）の入力信号を DCT の形式に分解する。これにより、MLT 変換は 2 つの手順で行われる。最初に、入力信号を分解し、次に分解した信号の DCT を行う。

分解された音声信号は次のように計算される。

$$v(n) = w(159-n)x(159-n) + w(160+n)x(160+n),\ 0 \le n \le 159$$

$$v(n+160) = w(319-n)x(320+n) - w(n)x(639-n),\ 0 \le n \le 159$$

ここで、

$$w(n) = \sin\left(\frac{\pi}{640}(n+0.5)\right),\ 0 \le n \le 319$$

および

$$x(n) = \begin{cases} pSrcDstSpchOld[n],\ 0 \le n \le 319 \\ pSrcSpch(n-320),\ 320 \le n \le 639 \end{cases}$$

入力信号 *pSrcSpch* は、*pSrcDstSpchOld* に格納され、次のフレームで使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcSpch*、*pDstSpch*、または *pSrcSpchOld* が NULL。

DecomposeDCTToMLT

IDCT 出力信号を MLT 変換出力信号の形式で分解する。

```
IppStatus ippsDecomposeDCTToMLT_G722_16s(const Ipp16s
    *pSrcSpchDecomposed, Ipp16s *pSrcDstSpchDecomposedOld, Ipp16s
    *pDstSpch);
```

引数

pSrcSpchDecomposed	ソース・ベクトル [320] へのポインタ。
PsrcSpchDecomposedOld	直前のフレームの分解された音声サンプルのソース / デスティネーション・ベクトル [160] へのポインタ。
pDstSpch	デスティネーション・ベクトル [320] へのポインタ。

説明

関数 ippsDecomposeDCTToMLT_G722 は、ippsc.h ファイルで宣言される。この関数は、逆方向 DCT の出力信号を、逆方向の変調ラップ変換 (IMLT) の出力にフィットするよう分解する。これにより、IMLT 変換は 2 つの手順で行われる。最初に、逆方向の離散コサイン変換（IDCT）を行い、次に IDCT 出力の分解を行う。

分解された信号は次のように計算される。

$$pDstSpch(n) = w(n)u(159-n) + w(319-n)u_old(n), 0 \le n \le 159$$

$$pDstSpch(n+160) = w(160+n)u(n) - w(159-n)u_old(159-n), 0 \le n \le 159$$

ここで、

$$w(n) = \sin\left(\frac{\pi}{640}(n+0.5)\right), 0 \le n \le 319$$

u(n) = pSrcSpchDecomposed[*n*],
u_old(n)=pSrcDstDecomposedOld[*n*]

入力信号の未使用のハイ・ハーフは、*u_old*() に格納され、次のフレームで使用される。

$$u_old(n) = u(n+160), 0 \le n \le 159$$

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcSpch*、*pDstSpch*、または *pSrcSpchOld* が NULL。

HuffmanEncode_G722

量子化された大きさの包絡線インデックスに対してハフマン・エンコードを実行する。

```
IppStatus ippsHuffmanEncode_G722_16s32u(int category, int
     qntAmpEnvIndex, const Ipp16s *pSrcMLTCoeffs, Ipp32u *pDstCode, int
     *pCodeLength);
```

引数

category	Modulated Lapped Transform (MLT) 領域のカテゴリ。[0-7] の範囲。
qntAmpEnvIndex	範囲が [0-63] の量子化した大きさの包絡線インデックス。
pSrcMLTCoeffs	未処理の MLT 係数のソース・ベクトル [20] へのポインタ。
pDstCode	出力ハフマン・コードへのポインタ。
pCodeLength	出力ハフマン・コードの長さへのポインタ（ビット単位)。

説明

関数 `ippsHuffmanEncode_G722` は、`ippsc.h` ファイルで宣言される。この関数は、MLT 係数にある 20 個の領域から 1 つの量子化された大きさの、包絡線インデックスのハフマン・エンコードを実行する。

戻り値

`ippStsNoErr`	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSrcMLTCoeffs*、*pDstCode*、または *pCodeLength* が NULL。
IppStsScaleRangeErr	エラー。*category* または *qntAmpEnvIndex* が範囲外。

G.726 に関連する関数

本章では、ITU-T 勧告 G.726 付録 A に準拠した音声コーデックの開発に使用できるインテル® IPP 関数について説明する。

これらすべての関数のリストを次の表に示す。

表 9-7 G.726.1 に関連するインテル® IPP 関数

関数の基本名	操作
EncodeGetStateSize_G726	情報関数。エンコーダ・メモリで必要なバイト数を返す。
EncodeInit_G726	ADPCM エンコーダのメモリを初期化する。
Encode_G726	一様な PCM 入力の ADPCM 圧縮を実行する。
DecodeGetStateSize_G726	情報関数。デコーダ・メモリで必要なバイト数を返す。
DecodeInit_G726	G726 デコーダのメモリを初期化する。
Decode_G726	ADPCM ビットストリームの解凍を実行する。

EncodeGetStateSize_G726

情報関数。エンコーダ・メモリで必要な
バイト数を返す。

```
IppStatus ippsEncodeGetStateSize_G726_16s8u (unsigned int* pEncSize);
```

引数

pEncSize	出力メモリ・サイズへのポインタ（バイト単位）。

説明

関数 ippsEncodeGetStateSize_G726 は、ippsc.h ファイルで宣言される。この関数は、G.726 ADPCM 圧縮を実行するために必要なメモリの量に関する情報を取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pEncSize* が NULL。

EncodeInit_G726

ADPCM エンコーダのメモリを初期化する。

```
IppStatus ippsEncodeInit_G726_16s8u (IppsEncoderState_G726_16s*
    pEncMem, IppSpchBitRate rate);
```

引数

pEncMem	G.726 エンコーダを正常に初期化するために必要なサイズが割り当てられた入力メモリ・バッファへのポインタ。
rate	G.726 エンコーダのエンコード・ビット・レート。IPP_SPCHBR_16000、IPP_SPCHBR_24000、IPP_SPCHBR_32000、または IPP_SPCHBR_40000 のいずれかでなければならない。

説明

関数 ippsEncodeInit_G726 は、ippsc.h ファイルで宣言される。この関数は、ポインタ *pEncMem* で指定したメモリを初期化し、リセット状態から開始する G.726 ADPCM 圧縮を有効にする。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。*rate* がエンコード・ビット・レート IPP_SPCHBR_16000, IPP_SPCHBR_24000、IPP_SPCHBR_32000、または IPP_SPCHBR_40000 のいずれかに等しくない。
ippStsNullPtrErr	エラー。ポインタ *pEncMem* が NULL。

Encode_G726

一様な PCM 入力の ADPCM 圧縮を
実行する。

```
IppStatus ippsEncode_G726_16s8u (IppsEncoderState_G726_16s* pEncMem,
    const Ipp16s *pSrc, Ipp8u *pDst, unsigned int len);
```

引数

pEncMem	ADPCM エンコード用に初期化されたメモリ・バッファへのポインタ。
pSrc	一様な PCM 入力音声ベクトルへのポインタ。
pDst	ADPCM ビットストリームの出力ベクトルへのポインタ。
len	入力および出力ベクトルの長さ。

説明

関数 ippsEncode_G726 は、ippsc.h ファイルで宣言される。この関数は、14 ビットの一様な PCM 音声入力（ITU-T 勧告 G.726 付録 A）に ADPCM 圧縮を実行する。圧縮には、*pEncMem* で指定したメモリで初期化された G.726 エンコーダのビット・レートを使用する。出力ベクトルの各バイトには、16、24、32、40 Kbit/s ビットレートの ADPCM 圧縮に対してそれぞれ 2、3、4、5 桁のビットが含まれている。

Mu-Law または A-Law 形式の PCM 入力は、ADPCM 圧縮を行う前に 14 ビットの一様な PCM に拡張する必要がある（勧告 G.726 を参照）。例えば、最初に関数 ippsMuLawToLin_8u16s または ippsALawToLin_8u16s を実行することで、8 ビットの Mu-Law または A-Law 形式の PCM を線形の 16 ビット PCM に拡張する。

線形の 16 ビット PCM 入力は、2 ビット右にシフトして (4 で割って)、関数 ippsEncode_G726 で適切な 14 ビットの一様な PCM 入力を取得する必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。*len* がゼロ以下。
ippStsNullPtrErr	エラー。ポインタ *pEncMem*、*pSrc*、または *pDst* が NULL。

DecodeGetStateSize_G726

情報関数。デコーダ・メモリで必要な
バイト数を返す。

```
IppStatus ippsDecodeGetStateSize_G726_8u16s (unsigned int*
pDecSize);
```

引数

pDecSize	出力メモリ・サイズへのポインタ（バイト単位）。

説明

関数 ippsDecodeGetStateSize_G726 は、ippsc.h ファイルで宣言される。この関数は、G.726 ADPCM 解凍を実行するために必要なメモリの量を報告する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDecSize* が NULL。

DecodeInit_G726

G726 デコーダのメモリを初期化する。

```
IppStatus ippsDecodeInit_G726_8u16s (IppsDecoderState_G726_16s*
    pDecMem, IppSpchBitRate rate, IppPCMLaw law);
```

引数

pDecMem	G.726 デコーダを正常に初期化するために必要なサイズが割り当てられた入力メモリ・バッファへのポインタ。
rate	G.726 デコーダのデコード・ビット・レート。IPP_SPCHBR_16000、IPP_SPCHBR_24000、IPP_SPCHBR_32000、または IPP_SPCHBR_40000。
law	出力音声 PCM の法則は、IPP_PCM_MULAW, IPP_PCM_ALAW または IPP_PCM_LINEAR のいずれかでなければならない。

説明

関数 ippsDecodeInit_G726 は、ippsc.h ファイルで宣言される。この関数は、ポインタ *pDecMem* で指定したメモリを初期化し、リセット状態から開始する ADPCM 解凍を有効にする。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。*rate* がデコード・ビット・レート IPP_SPCHBR_16000, IPP_SPCHBR_24000、IPP_SPCHBR_32000、または IPP_SPCHBR_40000 のいずれかに等しくない。 または *law* が出力 PCM 値 IPP_PCM_MULAW, IPP_PCM_ALAW か IPP_PCM_LINEAR のいずれかに等しくない。
ippStsNullPtrErr	エラー。*pDecMem* ポインタが NULL。

Decode_G726

ADPCM ビットストリームの解凍を実行する。

```
ppStatus ippsDecode_G726_8u16s (IppsDecoderState_G726_16s* pDecMem,
    const Ipp8u *pSrc, Ipp16s *pDst, unsigned int len)
```

引数

pDecMem	G.726 ADPCM デコーダ用に初期化されたメモリ・バッファへのポインタ。
pSrc	入力ベクトルへのポインタ。入力ベクトルの各バイトには、16、24、32、40 Kbit/s の ADPCM ビットストリームに対してそれぞれ 2､3､4､5 桁のビットが含まれている。
pDst	線形の 16 ビット PCM 音声出力ベクトルへのポインタ。
len	入力および出力ベクトルの長さ。

説明

関数 ippsDecode_G726 は、ippsc.h ファイルで宣言される。この関数は、ADPCM ビットストリーム入力（勧告 G.726）を線形の 16 ビット PCM へ解凍する。入力ビットストリームは、初期化された G.726 デコーダのビット・レートで ADPCM 圧縮されている必要がある。

戻り値

ippStsNoErr	エラーなし。
IppStsSizeErr	エラー。*len* がゼロ以下。
ippStsNullPtrErr	エラー。ポインタ *pDecMem*、*pSrc*、または *pDst* が NULL。

G.728 に関連する関数

本章では、ITU-T* 勧告 G.728 付録 I および H に準拠した音声コーデックの開発に使用できるインテル® IPP 関数について説明する。

これらすべての関数のリストを次の表に示す。

表 9-8　G.728 に関連するインテル® IPP 関数

関数の基本名	操作
IIRGetStateSize_G728	使用される IIR ステート構造体のサイズを取得する。
IIR_Init_G728	IIR ステート構造体を初期化する。
IIR_G728	IIR フィルタを複数のサンプルに適用する。
SynthesisFilterGetStateSize_G728	合成フィルタ・ステート構造体のサイズを取得する。
SynthesisFilterInit_G728	合成フィルタ・ステート構造体を初期化する。
SyntesisFilter_G728	合成フィルタを複数のサンプルに適用する。
CombinedFilterGetStateSize_G728	複合フィルタ・ステート構造体のサイズを取得する。
CombinedFilterInit_G728	複合フィルタ・ステート構造体を初期化する。
CombinedFilter_G728	複合フィルタを複数のサンプルに適用する。
PostFilterGetStateSize_G728	ポスト・フィルタ・ステート構造体のサイズを取得する。
PostFilterInit_G728	ポスト・フィルタ・ステート構造体を初期化する。
PostFilter_G728	ポスト・フィルタを複数のサンプルに適用する。
WinHybridGetStateSize_G728	ハイブリッド窓モジュールのステート構造体のサイズを取得する。
WinHybridInit_G728	ハイブリッド窓モジュールのステート構造体を初期化する。
WinHybrid_G728	ハイブリッド窓を適用する。
LevinsonDurbin_G728	自己相関係数から LP 係数を計算する。
CodebookSearch_G728	最良のコード・ベクトルをコードブックから検索する。

表 9-8 G.728 に関連するインテル® IPP 関数

関数の基本名	操作
ImpulseResponseEnergy_G728	形状コードブック・ベクトルのたたみ込みとエネルギー計算を実行する。

IIRGetStateSize_G728

使用される IIR ステート構造体のサイズを取得する。

```
IppStatus ippsIIR16sGetStateSize_G728_16s (int *pSize);
```

引数

pSize	出力 IIR ステート・サイズの値へのポインタ。

説明

関数 ippsIIR16sGetStateSize_G728 は、ippsc.h ファイルで宣言される。この関数は、正常に IIR フィルタを使用するために割り当てる必要があるメモリの最小サイズを返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSize* ポインタが NULL。

IIR_Init_G728

IIR ステート構造体を初期化する。

```
IppStatus ippsIIR16sInit_G728_16s (IppsIIRState16s_G728_16s *pMem );
```

引数

pMem	IIR フィルタ用に割り当てられたメモリへのポインタ。

説明

関数 ippsIIR16sInit_G728 は、ippsc.h ファイルで宣言される。この関数は、指定されたメモリ・ブロックを使用して IIR ステート構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMem* が NULL。

IIR_G728

IIR フィルタを複数のサンプルに適用する。

```
IppStatus ippsIIR16s_G728_16s (const Ipp16s *pCoeffs,
    const Ipp16s * pSrcQntSpeech, Ipp16s * pDstWgtSpeech, int len,
    IppsIIRState16s_G728_16s *pMem );
```

引数

pCoeffs	フィルタ係数ベクトル [20] へのポインタ。b_0 , …, b_9 , a_0 , …, a_9 （Q14）。
pSrcQntSpeech	ソース・ベクトル [*len*] へのポインタ。
pDstWgtSpeech	デスティネーション・ベクトル [*len*] へのポインタ。
len	ソースとデスティネーション・サンプルの数。
pMem	IIR フィルタ・ステート構造体へのポインタ。

説明

関数 ippsIIRState16s_G728 は、ippsc.h ファイルで宣言される。この関数は、伝達関数に従って量子化された入力音声を IIR フィルタで 1 つずつフィルタリングして、合成した音声出力を計算する。

$$\frac{1+\sum_{i=0}^{9} b_i \cdot z^{-i}}{1+\sum_{i=0}^{9} a_i \cdot z^{-i}}$$

intel®

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCoeffs*、*pSrcQntSpeech*、*pDstWgtSpeech*、または *pMem* のいずれかが NULL。

SynthesisFilterGetStateSize_G728

合成フィルタ・ステート構造体のサイズを
取得する。

```
IppStatus ippSynthesisFilterGetStateSize_G728_16s (int *pSize);
```

引数

pSize	合成フィルタ・ステート構造体の出力サイズの値へのポインタ。

説明

関数 ippSynthesisFilterGetStateSize_G728 は、ippsc.h ファイルで宣言される。この関数は、正常に合成フィルタを使用するために割り当てる必要があるメモリの最小サイズを返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSize* ポインタが NULL。

SynthesisFilterInit_G728

合成フィルタ・ステート構造体を
初期化する。

```
IppStatus ippSynthesisFilterInit_G728_16s
    (IppsSynthesisFilterState_G728_16s *pMem);
```

引数

pMem	合成フィルタ用に割り当てられたメモリへのポインタ。

説明

関数 ippSynthesisFilterInit_G728 は、ippsc.h ファイルで宣言される。この関数は、指定されたメモリ・ブロックを使用して合成フィルタ・ステート構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMem* が NULL。

SyntesisFilter_G728

合成フィルタを複数のサンプルに適用する。

```
IppStatus ippsSyntesisFilterZeroInput_G728_16s (const Ipp16s* pCoeffs,
    Ipp16s* pSrcDstExc, Ipp16s excSfs, Ipp16s* pDstSpeech, Ipp16s*
    pSpeechSfs, IppsSynthesisFilterState_G728_16s *pMem);
```

引数

pCoeffs	フィルタ係数ベクトル [51] へのポインタ。a_0 , …, a_{50}。Q14 で表現される。
pSrcDstExc	入力 / 出力ゲイン・スケーリングされた励振ベクトル [5] へのポインタ。
excSfs	直前のゲイン・スケーリングされた励振ベクトルの入力スケール。
pDstSpeech	出力量子化音声ベクトル [5] へのポインタ。
pSpeechSfs	量子化された音声ベクトルの出力スケール。
pMem	合成フィルタ・ステート構造体へのポインタ。

intel®

説明

関数 ippsSyntesisFilterZeroInput_G728 は、ippsc.h ファイルで宣言される。この関数は、伝達関数に基づいて、デコードされた音声ベクトルを、ゼロ入力応答の和および合成フィルタのゼロ・ステート応答として計算する。

$$\frac{1}{1+\sum_{i=1}^{50} a_i \cdot z^{-i}}$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCoeffs*、*pSrcDstExc*、*pDstSpeech*、*pSpeechSfs*、または *pMem* のいずれかが NULL。

CombinedFilterGetStateSize_G728

複合フィルタ・ステート構造体のサイズを取得する。

```
IppStatus ippsCombinedFilterGetStateSize_G728_16s (int *pSize);
```

引数

pSize	複合フィルタ・ステート構造体の出力サイズの値へのポインタ。

説明

関数 ippsCombinedFilterGetStateSize_G728 は、ippsc.h ファイルで宣言される。この関数は、正常に複合フィルタを使用するために割り当てる必要があるメモリの最小サイズを返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSize* ポインタが NULL。

CombinedFilterInit_G728

複合フィルタ・ステート構造体を
初期化する。

```
IppStatus ippCombinedFilterInit_G728_16s
    (IppsCombinedFilterState_G728_16s *pMem);
```

引数

pMem	複合フィルタ用に割り当てられたメモリへのポインタ。

説明

関数 ippCombinedFilterInit_G728 は、ippsc.h ファイルで宣言される。この関数は、指定されたメモリ・ブロックを使用して複合フィルタ・ステート構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMem* が NULL。

CombinedFilter_G728

複合フィルタを複数のサンプルに適用する。

```
IppStatus ippsCombinedFilterZeroInput_G728_16s(const Ipp16s* pSyntCoeff, const
    Ipp16s* pWgtCoeff, Ipp16s* pDstWgtZIR, IppsCombinedFilterState_G728_16s*
    pMem);
IppStatus ippsCombinedFilterZeroState_G728_16s(const Ipp16s* pSyntCoeff, const
    Ipp16s* pWgtCoeff, Ipp16s* pSrcDstExc, Ipp16s excSfs, Ipp16s* pDstSpeech,
    Ipp16s* pSpeechSfs, IppsCombinedFilterState_G728_16s* pMem);
```

引数

pSyntCoeff	フィルタ係数ベクトル [50] へのポインタ。a_1 , ..., a_{50}。Q14 で表現される。

pWgtCoeff	フィルタ係数ベクトル [20] へのポインタ。B_1 , ..., B_{10} , A_1 , ..., A_{10}。Q14 で表現される。
pSrcDstExc	出力ゲイン・スケーリングされた励振ベクトル [5] へのポインタ。
excSfs	直前のゲイン・スケーリングされた励振ベクトルの入力スケール。
pDstWgtZIR	複合フィルタの出力ゼロ入力応答ベクトル [5] へのポインタ。
pDstSpeech	量子化された音声出力ベクトル [5] へのポインタ。
pSpeechSfs	量子化された音声ベクトルの出力スケール。
pMem	複合フィルタ・ステート構造体へのポインタ。

説明

関数 ippsCombinedFilterZeroInput_G728 と ippsCombinedFilterZeroState_G728 は、ippsc.h ファイルで宣言される。

ippsCombinedFilterZeroInput_G728_16s。この関数は、伝達関数に基づいて、50 次の合成フィルタと 10 次の IIR フィルタを重ね合わせて、複合フィルタのゼロ入力応答を計算する。

$$\frac{1}{1+\sum_{i=1}^{50} a_i \cdot z^{-i}} \cdot \frac{1+\sum_{i=1}^{10} B_i \cdot z^{-i}}{1+\sum_{i=1}^{10} A_i \cdot z^{-i}}$$

ippsCombinedFilterZeroState_G728_16s。この関数は、最初にゼロ・ステートの複合フィルタを使用してゲイン・スケーリングされた励振ベクトルをフィルタリングする（上記の関数を参照）。次に、合成および IIR フィルタのゼロ・ステート応答を追加して、複合フィルタのメモリを更新する。量子化された音声出力ベクトルは、メモリの更新ごとに取得される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSyntCoeff*、*pWgtCoeff*、*pSrcDstExc*、*pDstWgtZIR*、*pDstSpeech*、*pSpeechSfs*、または *pMem* が NULL。

PostFilterGetStateSize_G728

ポスト・フィルタ・ステート構造体の
サイズを取得する。

```
IppStatus ippsPostFilterGetStateSize_G728_16s (int *pSize);
```

引数

pSize	ポスト・フィルタ・ステート構造体の出力サイズの値へのポインタ。

説明

関数 ippsPostFilterGetStateSize_G728 は、ippsc.h ファイルで宣言される。この関数は、正常にポスト・フィルタを使用するために割り当てる必要があるメモリの最小サイズを返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSize* が NULL。

PostFilterInit_G728

ポスト・フィルタ・ステート構造体を
初期化する。

```
IppStatus ippPostFilterInit_G728_16s (IppsPostFilterState_G728_16s
    *pMem);
```

引数

pMem	ポスト・フィルタに割り当てられたメモリへのポインタ。

説明

関数 ippPostFilterInit_G728 は、ippsc.h ファイルで宣言される。この関数は、指定したメモリ・ブロックを使用して、ポスト・フィルタ・ステート構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pMem* が NULL。

PostFilter_G728

ポスト・フィルタを複数のサンプルに適用する。

```
IppStatus ippsPostFilter_G728_16s (Ipp16s gl, Ipp16s glb, Ipp16s kp,
    Ipp16s tiltz, const Ipp16s *pCoeffs, const Ipp16s *pSrc, Ipp16s
    *pDst, IppsPostFilterState_G728_16s *pMem);
```

引数

gl	LTP スケーリング係数。
glb	LTP 積。
kp	LTP ラグ。現在のフレームのピッチ周期。
tiltz	STP ティルト補正係数。
pCoeffs	ポスト・フィルタ係数ベクトル [20] へのポインタ。B_1 , …, B_{10} , A_1 , …, A_{10}。
pSrc	入力音声ベクトル [5] へのポインタ。要素 -*kp*,...,-1 は、LTP フィルタのメモリとして指定する必要がある。
pDst	出力ポスト・フィルタ音声ベクトル [5] へのポインタ。
pMem	ポスト・フィルタ・ステート構造体へのポインタ。

説明

関数 ippsPostFilter_G728 は、ippsc.h ファイルで宣言される。この関数は、LTP フィルタと STP フィルタで構成された伝達関数に基づいて入力サンプルを 1 回ずつフィルタリングする。

$$gl \cdot (1 - glb \cdot z^{-kp}) \cdot \frac{1 - \sum_{i=1}^{10} B_i \cdot z^{-i}}{1 - \sum_{i=1}^{10} A_i \cdot z^{-i}} \cdot (1 + tiltz \cdot z^{-1})$$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pCoeffs*、*pSrc*、*pDst*、または *pMem* が NULL。

WinHybridGetStateSize_G728

ハイブリッド窓モジュールのステート構造体のサイズを取得する。

```
IppStatus ippsWinHybridGetStateSize_G728_16s (int M, int L, int N, int
    DIM, int *pSize);
```

引数

M	LPC 窓の入力の長さ。
L	入力適応サイクルのサイズ（サンプル単位）。
N	非再帰窓サンプルの入力番号。
DIM	関数 ippsWinHybrid_G728 が、入力音声のブロック・スケーリングで使用する入力ブロック・サイズ。
pSize	ハイブリッド窓モジュールのステート構造体の出力サイズの値へのポインタ。

説明

関数 `ippsWinHybridGetStateSize_G728` は、`ippsc.h` ファイルで宣言される。この関数は、指定した窓パラメータに基づいて、正常にハイブリッド窓モジュールを使用するために割り当てる必要があるメモリの最小サイズを返す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSize* ポインタが NULL。
ippStsSizeErr	エラー。*L*、*M*、または *N* がゼロ以下。

WinHybridInit_G728

ハイブリッド窓モジュールのステート構造体を初期化する。

```
IppStatus ippWinHybridInit_G728_16s (const Ipp16s *pWinTab, int M, int L,
    int N, int DIM, Ipp16s a2L, IppsWinHybridState_G728_16s *pMem);
```

引数

pWinTab	窓係数の入力ベクトル。
M	LPC 窓の入力の長さ。長さは 10 以上でなければならない。
L	入力適応サイクルのサイズ（サンプル単位）。
N	非再帰的窓サンプルの入力番号。
a2L	ハイブリッド窓モジュールの再帰成分の計算で使用する a^{2L} 倍数。
N	非再帰的窓サンプルの入力番号。

説明

関数 `ippWinHybridInit_G728` は、`ippsc.h` ファイルで宣言される。この関数は、指定したメモリ・ブロックを使用してハイブリッド窓モジュールのステート構造体を初期化する。再帰的成分と直前の音声サンプルは、ゼロに初期化される。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pMem* が NULL。

WinHybrid_G728

ハイブリッド窓を適用する。

```
IppStatus ippsWinHybridBlock_G728_16s(Ipp16s bfi, const Ipp16s *pSrc, const
    Ipp16s *pSrcSfs, Ipp16s *pDst, IppsWinHybridState_G728_16s *pMem);
IppStatus ippsWinHybrid_G728_16s(Ipp16s bfi, const Ipp16s *pSrc, const
    Ipp16s *pSrcSfs, Ipp16s *pDst, IppsWinHybridState_G728_16s *pMem);
```

引数

bfi	入力不良フレーム・インジケータ。「1」は、不良フレームを意味する。その他の値は、良いフレームを意味する。
pSrc	入力音声ベクトル [20] へのポインタ。
pSrcSfs	入力音声ベクトルの要素のスケール係数の入力ベクトル [5] へのポインタ。*pSrcSfs* ベクトルの各要素は、入力音声ベクトルの要素のブロックに対して別個のスケール係数を指定できる。したがって、*pSrcSfs*[0] は要素 *pSrc*[0],..*pSrc*[*DIM*-1] のスケール係数、*pSrcSfs*[1] は *pSrc*[*DIM*],..*pSrc*[2**DIM*-1] のスケール係数となる。ここで、*DIM* は、`WinHybridInit_G728()` 関数で定義する必要があるブロック・サイズである。
pDst	出力反射係数ベクトル [M+1] へのポインタ。ここで、M は、`WinHybridInit_G728()` 関数で定義する必要がある LPC 窓の長さである。
pMem	ポスト・フィルタ・ステート構造体へのポインタ。

説明

関数 ippsWinHybrid_G728 は、ippsc.h ファイルで宣言される。この関数は、最初に入力音声ベクトルに窓を適用し、LPC 分析で必要な自己相関係数を次の式で計算する。

$$R_m(i) = r_m(i) + \sum_{k=m-N}^{m-1} s_m(k)s_m(k-i), m = 0,..,M$$

ここで、適応サイクルの再帰的成分は次のように計算される。

$$r_m(i) = \alpha^{2L} \cdot r_{m-L}(i) + \sum_{k=m-L-N}^{m-N-1} s_m(k)s_m(k-i)$$

再帰的成分は、直前の適応サイクルで計算されたデータを使用して計算され、モジュールのメモリに格納される。

ホワイト・ノイズ補正は、次のようにエネルギーを上げることで自己相関係数に適用される。

$$r_m(0) = \frac{257}{256} \cdot r_m(0)$$

再帰的成分と直前の音声サンプルは、モジュールのメモリに格納され、次の適応サイクルで使用される。

不良フレーム・インジケータがオンの場合、10 個の自己相関係数のみが計算され、出力される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcSfs*、*pSrc*、*pDst*、または *pMem* が NULL。

LevinsonDurbin_G728

自己相関係数から LP 係数を計算する。

```
IppStatus ippsLevinsonDurbin_G728_16s_Sfs(const Ipp16s *pSrcAutoCorr, int
     order, Ipp16s *pDstLPC, Ipp16s *pDstResidualEnergy, Ipp16s
     *pDstScaleFactor);
IppStatus ippsLevinsonDurbin_G728_16s_ISfs(const Ipp16s *pSrcAutoCorr, int
     numSrcLPC, int order, Ipp16s *pSrcDstLPC, Ipp16s *pSrcDstResidualEnergy,
     Ipp16s *pSrcDstScaleFactor);
```

引数

pSrcAutoCorr	入力自己相関係数ベクトル [*order*+1] へのポインタ。
order	計算する LP 係数の入力数。
numSrcLPC	事前に計算された LPC の入力数。
pDstLPC	出力 LPC ベクトル [*order*] へのポインタ。
pSrcDstLPC	入力 / 出力 LPC ベクトル [*order*] へのポインタ。
pDstResidualEnergy	出力残留エネルギーへのポインタ。
pSrcDstResidualEnergy	事前に計算された LPC の入力/出力残留エネルギーへのポインタ。
pDstScaleFactor	LPC ベクトルの出力スケール係数へのポインタ。
pSrcDstScaleFactor	事前に計算された LPC の入力スケール係数と出力 LPC ベクトルの出力スケール係数へのポインタ。

説明

関数 ippsLevinsonDurbin_G728_16s_Sfs と ippsLevinsonDurbin_G728_16s_ISfs は、ippsc.h ファイルで宣言される。

ippsLevinsonDurbin_G728_16s_Sfs。この関数は、次の一連の一次方程式を解くことによって、線形予測（LP）係数を計算するのに使用する。

$$\sum_{i=0}^{order-1} a_i \cdot r(|i-k|) = r(k), k = 1,2,\ldots,order$$

ここで、a_i , $i = 0,1,\ldots,order\text{-}1$ は、計算される LP 係数である。これらの LP 係数は、出力 LPC ベクトルに格納される。この関数によって使用される Levinson-Durbin アルゴリズムの説明は、ippsLevinsonDurbin_G729 関数を参照のこと。

ippsLevinsonDurbin_G728 と ippsLevinsonDurbin_G729B における演算は同じであるが、ビットごとの正確な LPC ではない。ippsLevinsonDurbin_G729B 関数は、Q12 の LPC を出力する。ippsLevinsonDurbin_G728 関数は、オーバーフローが発生すると、自動的に LPC を再度スケーリングする。

ippsLevinsonDurbin_G728_16s_Isfs。この関数は、より大きな *order* における Levinson-Durbin の再帰を続行するために使用する。再帰を続行するには、LPC、そのスケール係数、直前の再帰の残留エネルギー、および追加の自己相関係数が使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。入力または出力ポインタのいずれかが NULL。
ippStsRangeErr	エラー。*order* がゼロ以下、またはインプレース関数で *order* < *numSrcLPC* である。

CodebookSearch_G728

最良のコード・ベクトルをコードブックから検索する。

```
IppStatus ippsCodebookSearch_G728_16s(const Ipp16s* pSrcCorr, const
    Ipp16s* pSrcEnergy, int* pDstShapeIdx, int* pDstGainIdx, short*
    pDstCodebookIdx, IppSpchBitRate rate);
```

引数

pSrcCorr	ターゲット信号の時間反転たたみ込みの入力ベクトル [5] へのポインタ。
pSrcEnergy	たたみ込みされた形状コードベクトルのエネルギーの入力ベクトル [128] へのポインタ。
pDstShapeIdx	最良の 7 ビットの出力形状コードブック・インデックスへのポインタ。
pDstGainIdx	最良の 3 ビットの出力ゲイン・コードブック・インデックスへのポインタ。
pDstCodebookIdx	送信される最良の出力コードブック・インデックスへのポインタ。

rate	入力コーディング・ビット・レート。

説明

関数 ippsCodebookSearch_G728 は、ippsc.h ファイルで宣言される。この関数は、ゲイン・コードブックと形状コードブックからゲインと形状コードブック・インデックスの最良の組み合わせを検索する際に使用する、「誤差測定と最良コードブック・インデックス・セレクタ」ブロックを実行する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcCorr*、*pSrcEnergy*、*pDstShapeIdx*、*pDstGainIdx*、または *pDstCodebookIdx* が NULL。
IppStsRangeErr	エラー。*rate* が 16、12.8、または 9.6 Kbit/s コーディング・ビットでそれぞれ有効な IPP_SPCHBR_16000、IPP_SPCHBR_12800、または IPP_SPCHBR_9600 のいずれかではない。

ImpulseResponseEnergy_G728

形状コードブック・ベクトルのたたみ込み
とエネルギー計算を実行する。

```
IppStatus ippsImpulseResponseEnergy_G728_16s(const Ipp16s
    *pSrcImpResp, Ipp16s *pDstEnergy);
```

引数

pSrcImpResp	合成および重み付きフィルタの入力インパルス応答ベクトル [5] へのポインタ。
pDstEnergy	たたみ込みされた形状コードベクトルの出力エネルギー・ベクトル [128] へのポインタ。

説明

関数 ippsImpulseResponseEnergy_G728 は、ippsc.h ファイルで宣言される。この関数は、たたみ込みされた形状コードベクトルのエネルギーを計算するために使用する「形状コードベクトルのたたみ込みとエネルギー測定」ブロックを実行する。

intel®

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pSrcImpResp* または *pDstEnergy* が NULL。

オーディオ符号化関数 10

オーディオ符号化用のインテル® IPP には、各種のコーデックに利用される汎用関数と、MPEG-4 オーディオ・エンコーダおよびデコーダ（[ISO14496] を参照）と、MP3 エンコーダおよびデコーダ向けの多数の特殊関数が含まれている。これらの関数は、大規模で複雑な計算を使用するパイプライン・ブロックをサポートしている。

現在のバージョンでは、携帯機器向けに最適化された MPEG-4 AAC Main Profile デコーダと携帯機器向けに最適化された MPEG-1, 2 Layer III エンコーダ（[ISO11172] および [ISO13818] を参照）の開発が可能である。

本章では、CCITT G.711 仕様（[CCITT] を参照）に準拠した μ-law 形式および A-law 形式の圧伸を実行する圧伸関数についても説明する。

すべてのオーディオ符号化関数を 表 10-1 に示す。

表 10-1　インテル® IPP オーディオ符号化関数

関数の基本名	操作
インターリーブ形式から複数行形式への変換関数	
Interleave	非インターリーブ形式の信号をインターリーブ形式に変換する。
Deinterleave	インターリーブ形式の信号を非インターリーブ形式に変換する。
スペクトル・データ・プレ量子化関数	
Pow34	ベクトルを 3/4 乗する。
Pow43	ベクトルを 4/3 乗する。
スケール係数計算関数	
CalcSF	ビットストリームの値から実際のスケール係数を復元する。
仮数変換およびスケーリング関数	
ApplySF_I	スペクトル・バンドの境界に従って、スペクトル・バンドにスケール係数を適用する。
MakeFloat	仮数の配列と指数の配列を float 配列に変換する。
変形離散コサイン変換関数	
MDCTFwdInitAlloc, MDCTInvInitAlloc	変形離散コサイン変換の指定構造体を初期化する。
MDCTFwdFree, MDCTInvFree	離散コサイン変換の指定構造体をクローズする。
MDCTFwdGetBufSize, MDCTInvGetBufSize	MDCT 作業バッファのサイズを取得する。

表 10-1　インテル® IPP オーディオ符号化関数 （続き）

関数の基本名	操作
MDCTFwd, MDCTInv	信号の順方向または逆方向の変形離散コサイン変換（MDCT）を計算する。
ブロック・フィルタリング関数	
FIRBlockInitAlloc	FIR ブロック・フィルタ・ステートを初期化する。
FIRBlockFree	FIR ブロック・フィルタ・ステートをクローズする。
FIRBlockOne	FIR ブロック・フィルタを使用して、サンプルのベクトルをフィルタリングする。
周波数領域予測関数	
FDPInitAlloc	予測機構のステートを初期化する。
FDPFree	FDP ステートをクローズする。
ResetFDP	すべてのスペクトル線の予測機構をリセットする。
ResetFDP_SFB	指定されたスケール係数バンドの予測機構固有の情報をリセットする。
ResetFDPGroup	スペクトル線のグループの予測機構をリセットする。
FDPFwd	周波数領域予測手順を実行し、予測エラーを計算する。
FDPInv	周波数領域予測手順を使用して、予測エラーから入力信号を復元する。
ハフマン・アルゴリ Y ム関数	
GetSizeHDT	ハフマン・デコード・テーブルのサイズを計算する。
BuildHDT	ハフマン・デコード・テーブルを構築する。
DecodeVLC	ハフマン・テーブルから値をデコードする。
GetSizeHET	入力ハフマン・テーブルに必要なサイズを計算する。
BuildHET	ハフマン・テーブルを構築する。
HuffmanCountBits	量子化された値のエンコードに必要なサイズを計算する。
EncodeVLC	指定されたハフマン･テーブルを使用して､スペクトル値のブロックをエンコードする。
GetSizeHET_VLC	内部形式の入力ハフマン・テーブルに必要なサイズを計算する。
BuildHET_VLC	内部形式のハフマン・テーブルをビルドする。
CountBits	入力配列をエンコードするのに必要なサイズをビット単位で計算する。
EncodeBlock	入力配列をエンコードする。
ベクトル量子化関数	
CdbkInitAlloc	コードブック構造体を初期化する。
CdbkFree	CdbkInitAlloc で作成した IppsCdbkState_VQ_32f 構造体をクローズする。
PreSelect_VQ	コードブックの最も近いコード・ベクトルの候補を選択する。
MainSelect_VQ	歪みが最小限の最適なインデックスを検索する。
IndexSelect_VQ	指定されたコードブックに対して最適なベクトルのセットを検索する。
VectorReconstruction_VQ	インデックスからベクトルを再構築する。

表 10-1　インテル® IPP オーディオ符号化関数 （続き）

関数の基本名	操作
圧伸関数	
MuLawToLin	8 ビット μ-law 形式でエンコードされたサンプルを線形サンプルにデコードする。
LinToMuLaw	8 ビット μ-law 形式を使用して線形サンプルをエンコードし、それらをベクトルに格納する。
ALawToLin	8 ビット A-law でエンコードされたサンプルを線形サンプルにデコードする。
LinToALaw	8 ビット A-law 形式を使用して線形サンプルをエンコードし、それら配列に格納する。
MuLawToALaw	8 ビット μ-law 形式でエンコードされたサンプルを 8 ビット A-law 形式でエンコードされたサンプルに変換する。
ALawToMuLaw	8 ビット A-law 形式でエンコードされたサンプルを 8 ビット μ-law 形式でエンコードされたサンプルに変換する。
MP3 エンコーダ関数	
AnalysisPQMF_MP3	MP3 ハイブリッド分析フィルタ・バンクの第 1 ステージを実行する。
MDCTFwd_MP3	MP3 ハイブリッド分析フィルタ・バンクの第 2 ステージを実行する。
PsychoacousticModelTwo_MP3	ISO/IEC 11172-3 の心理音響モデル 2 を実装して、PCM オーディオ入力のブロックに関連するマスクされたしきい値と知覚エントロピを推定する。
JointStereoEncode_MP3	独立した左と右チャネルのスペクトル係数ベクトルを量子化するのに適した mid/side（MS）やインテンシティ（IS）モード係数ベクトルの組み合わせに変換する。
Quantize_MP3	分析フィルタ・バンクによって生成されたスペクトル係数を量子化する。
PackScalefactors_MP3	スケール係数にノイズレスの符号化を適用して、ビットストリーム・バッファに出力をパックする。
HuffmanEncode_MP3	量子化サンプルに無損失のハフマン符号化を適用して、ビットストリーム・バッファに出力をパックする。
PackFrameHeader_MP3	フレーム・ヘッダの内容をビットストリームにパックする。
PackSideInfo_MP3	サイド情報をビットストリーム・バッファにパックする。
BitReservoirInit_MP3	ビット貯蓄ステート構造体のすべての要素を初期化する。
MP3 デコーダ関数	
UnpackFrameHeader_MP3	オーディオ・フレーム・ヘッダをアンパックする。
UnpackSideInfo_MP3	入力ビットストリームからサイド情報をアンパックする。この情報は、関連するフレームのデコードに使用される。
UnpackScaleFactors_MP3	スケール係数をアンパックする。
HuffmanDecode_MP3 HuffmanDecodeSfb_MP3 HuffmanDecodeSfbMbp_MP3	ハフマン・データをデコードする。
ReQuantize_MP3 ReQuantizeSfb_MP3	デコードされたハフマン符号を再量子化する。

表 10-1 インテル® IPP オーディオ符号化関数 （続き）

関数の基本名	操作
MDCTInv_MP3	ハイブリッド合成フィルタ・バンクの第 1 ステージを実行する。
SynthPQMF_MP3	ハイブリッド合成フィルタ・バンクの第 2 ステージを実行する。
MPEG-2 プリミティブ関数	
UnpackADIFHeader_AAC	AAC ADIF フォーマット・ヘッダを取得する。
UnpackADTSFrameHeader_AAC	入力ビットストリームから ADTS フレーム・ヘッダを取得する。
DecodePrgCfgElt_AAC	入力ビットストリームからプログラム構成要素を取得する。
DecodeChanPairElt_AAC	入力ビットストリームから *channel_pair_element* を取得する。
NoiselessDecoder_LC_AAC	1 つのチャネルのすべてのデータをデコードする。
DecodeDatStrElt_AAC	入力ビットストリームからデータ・ストリーム要素を取得する。
DecodeFillElt_AAC	入力ビットストリームからフィル要素を取得する。
QuantInv_AAC	現在のチャネルのハフマン符号の逆量子化を実行する（インプレース方式）。
DecodeMsStereo_AAC	ペア・チャネルの MS ステレオを処理する（インプレース方式）。
DecodeIsStereo_AAC	ペア・チャネルのインテンシティ・ステレオを処理する。
DeinterleaveSpectrum_AAC	ショート・ブロックの係数をデインターリーブする。
DecodeTNS_AAC	TNS（Temporal Noise Shaping）をデコードする（インプレース方式）。
MDCTInv_AAC	時間－周波数領域信号をマップし、1024 個の 16 ビット符号付きリトル・エンディアン PCM サンプルを再構築する。
MPEG-4 プリミティブ関数	
DecodeMainHeader_AAC	ビットストリームからメイン・ヘッダ情報とメイン・レイヤ情報を取得する。
DecodeExtensionHeader_AAC	ビットストリームから拡張ヘッダ情報と拡張レイヤ情報を取得する。
DecodePNS_AAC	ICS 内で知覚ノイズ置換（PNS）符号化を実装する。
LongTermReconstruct_AAC	長期再構築を使用して、連続した符号化フレーム間における信号の冗長性を削減する。
MDCTFwd_AAC	PCM サンプルのスペクトル係数を生成する。
EncodeTNS_AAC	長期再構築ループで逆 TNS を実行する（インプレース方式）。
LongTermPredict_AAC	予測された時間領域信号を長期再構築（LTP）ループから取得する。
NoiseLessDecode_AAC	ノイズレス・デコードを実行する。
LtpUpdate_AAC	必要なバッファの更新を長期再構築（LTP）ループで実行する。

intel®

インターリーブ形式から複数行形式への変換関数

この項では、インターリーブ形式のマルチチャネル信号と非インターリーブ形式のマルチチャネル信号の間の変換を実行する関数について説明する。インターリーブ形式の信号では、異なるチャネルのサンプルが 1 つのベクトルにインターリーブ（交互に配置）される。非インターリーブ形式の信号では、各チャネルのサンプルが別々のベクトルに格納される。

注： 配列にはアライメントされたメモリとアライメントの合っていないメモリのいずれも使用できるが、メモリのアライメントが合っていないと、パフォーマンスが低下する。

Interleave

非インターリーブ形式の信号を
インターリーブ形式に変換する。

```
IppStatus ippsInterleave_16s(const Ipp16s** pSrc, int ch_num, int len,
    Ipp16s* pDst);
IppStatus ippsInterleave_32f(const Ipp32f** pSrc, int ch_num, int len,
    Ipp32f* pDst);
```

引数

pSrc	特定のチャネルのサンプルを格納するベクトル [*len*] へのポインタの配列。
ch_num	チャネルの数。
len	各チャネルのサンプルの数。
pDst	インターリーブ形式のデスティネーション・ベクトル [*ch_num * len*] へのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsInterleave は、次の式に従って、非インターリーブ形式の信号をインターリーブ形式に変換する。

$pDst[i] = pSrc[i \text{ div } ch_num][i \text{ mod } ch_num], 0 \le i < ch_num * len,$

div は商の整数部分、mod は剰余である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* または *ch_num* がゼロ以下。
ippStsMisalignedBuf	データのアライメントが合っていない。アライメントされたデータを提供することで、パフォーマンスは向上する。

Deinterleave

インターリーブ形式の信号を非インターリーブ形式に変換する。

```
IppStatus ippsDeinterleave_16s(const Ipp16s* pSrc, int ch_num, int len,
    Ipp16s** pDst);
IppStatus ippsDeinterleave_32f(const Ipp32f* pSrc, int ch_num, int len,
    Ipp32f** pDst);
```

引数

pSrc	インターリーブされたサンプルのベクトル [*ch_num* * *len*] へのポインタ。
ch_num	チャネルの数。
len	各チャネルのサンプルの数。
pDst	特定のチャネルのサンプルを格納するベクトル [*len*] へのポインタの配列。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsDeinterleave は、次の式に従って、インターリーブ形式の入力信号を非インターリーブ形式に変換する。

$pDst[i][j] = pSrc[i + j * ch_num], 0 \le i < ch_num, 0 \le j < len.$

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* または *ch_num* がゼロ以下。
ippStsMisalignedBuf	データのアライメントが合っていない。アライメントされたデータを提供することで、パフォーマンスは向上する。

スペクトル・データ・プレ量子化関数

MPEG-1, 2 Layer III および MPEG-4 AAC オーディオ・エンコーダは、量子化の前にスペクトル・データを 3/4 乗し、量子化される値の全範囲にわたって S/N 比の安定性を向上させる。デコーダの再量子化機構が、エンコーダの出力を 4/3 乗して値を線形化する。

Pow34

ベクトルを 3/4 乗する。

```
IppStatus ippsPow34_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int len);
IppStatus ippsPow34_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

引数

pSrc	入力データ・ベクトル [*len*] へのポインタ。
pDst	出力データ・ベクトル [*len*] へのポインタ。
len	入力ベクトルと出力ベクトル内の要素の数。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsPow34 は、次の式に従って *pSrc* の各要素の計算を実行し、

$$pDst[i] = |pSrc[i]|^{\frac{3}{4}}, 0 \le i < len$$

結果を *pDst* に格納する。

例えば、MPEG-1 Layer III では、次の式に従って、スペクトル値のベクトル全体の量子化が行われる。

$$ix(i) = nint\left(\left(\frac{|xr(i)|}{\sqrt[4]{2^{qquant+quantanf}}}\right)^{0.75} - 0.0946\right)$$ 、ここで

ix は量子化された値の配列、*i* は配列内の値の数、*nint* は非整数値を最も近い整数値に丸めるのに使用される関数、*xr* はスペクトル値の大きさのベクトル、*qquant* は量子化機構のステップ・サイズ情報、*quantanf* はスペクトルの平坦さの尺度によって異なる定数である。

この演算には、関数 `ippsPow34_32f16s` を使用できる。ただし、すべての量子化された値の最大値がテーブル範囲から外れる場合や、全体のビット和が利用可能なビット数を超える場合、新しい *qquant* 値と、同じ *xr* 配列を指定して、この演算を数回繰り返して実行する必要がある。

あるいは、関数 `ippsPow34_32f` を使用して、内側反復ループの前に 1 回だけ *xr* の 3/4 乗を計算し、一定の乗数を指定して量子化を反復するたびにその値を使用し、得られた値を `short` にも変換できる。

$$multiplier = \left(\frac{1}{\sqrt[4]{2^{qquant+quantanf}}}\right)^{0.75}$$

この演算は、関数 `ippsMulC_Low_32f16s` によってサポートされる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsMisalignedBuf	データのアライメントが合っていない。アライメントされたデータを提供することで、パフォーマンスは向上する。
ippStsSizeErr	エラー。*len* がゼロ以下。
ippStsNanArg	警告。入力データ・ベクトル内で NaN が検出された。
ippStsOutOfRangeErr	エラー。ソース配列の要素の値が 1 000 000 より大きい。

Pow43

ベクトルを 4/3 乗する。

```
IppStatus ippsPow43_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
```

引数

pSrc	入力データ・ベクトル [*len*] へのポインタ。
pDst	出力データ・ベクトル [*len*] へのポインタ。
len	入力ベクトルと出力ベクトル内の要素の数。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsPow43 は、次の式に従って、*pSrc* の各要素の計算を実行し、

$$pDst[i] = sign(pSrc[i]) * |pSrc[i]|^{\frac{4}{3}}, 0 \le i < len$$

結果を *pDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsMisalignedBuf	データのアライメントが合っていない。アライメントされたデータを提供することで、パフォーマンスは向上する。
ippStsSizeErr	エラー。*len* がゼロ以下。

注： *pSrc* 配列内の入力値は、8206 を超えてはならない。この関数は、パフォーマンスへの影響を避けるために、入力値がこの必要条件を満たしているかどうかをチェックしない。配列の入力値がこの上限を超える場合は、この関数は正常に動作しない。この場合、エラー・コードは返されない。

スケール係数計算関数

MPEG-2, 4 GA AAC デコーダでは、ビットストリームから抽出されたスケール係数に、追加の復元手順が必要である。関数 CalcSF は、ビットストリーム内で送信される値から実際のスケール係数を復元する。

CalcSF

ビットストリームの値から実際の
スケール係数を復元する。

```
IppStatus ippsCalcSF_16s32f(const Ipp16s* pSrc, int offset, Ipp32s*
    pDst,
    int len);
```

引数

pSrc	入力データ配列へのポインタ。
offset	スケール係数オフセット。
pDst	出力データ配列へのポインタ。
len	ベクトル内の要素の数。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsCalcSF は、共通のスケール係数オフセットを使用して、ビットストリーム内で送信される値 *pSrc* から実際のスケール係数を復元する。計算は次の式に従って実行される。

$$pDst[i] = 2^{\frac{1}{4}(pSrc[i] - offset)}, 0 \le i < len$$

復元されたスケール係数は、*pDst* に書き込まれる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

注： *pSrc* 配列内の入力値は、$|pSrc[i] - offset| \le 128$ の範囲内に入っている必要がある。入力値がこの範囲を外れると、この関数は正常に動作しない。この場合、エラー・メッセージは返されない。

仮数変換およびスケーリング関数

MPEG-2, 4 GA AAC デコーダでは、逆方向に量子化された一連のスペクトル係数から元のスペクトル値を復元するために、スケール適用手順が必要である。スペクトル全体が一連のスケール係数バンドに分割される。バンドによって幅が異なるため、バンドの位置はオフセット・ベクトルによって定義される。

また、AC3 デコーダでは、ビットストリーム内の一連の指数と仮数から元のスペクトル値を復元するために、仮数変換手順が必要である。

ApplySF_I

スペクトル・バンドの境界に従って、
スペクトル・バンドにスケール係数を
適用する。

```
IppStatus ippsApplySF_32f_I(Ipp32f* pSrcDst, const Ipp32f* pSF,
    const int *pBandOffset, int bands_number);
```

引数

pSrcDst	入力データ配列および出力データ配列へのポインタ。配列のサイズは、`pBandOffset[bands_number]` と同じか、それ以上でなければならない。
pSF	スケール係数を格納するデータ配列へのポインタ。配列のサイズは、*bands_number* と同じか、それ以上でなければならない。
pBandsOffset	バンド・オフセットのベクトルへのポインタ。配列のサイズは、*bands_number* + 1 と同じか、それ以上でなければならない。
bands_number	スケール係数が適用されるバンドの数。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsApplySF_I は、入力ベクトル、スケール係数のベクトル、バンド・オフセットのベクトルから、スケーリングされた値を計算する。演算はインプレースで実行される。

この関数は、入力ベクトル *pSF* から作成されるバンドに対して、*bands_number* 個の要素を持つ一連のスケール係数 *pSrc* を適用する。バンドの境界は、バンド・オフセットのベクトル *pBandOffset* によって定義される。各バンド内のすべての値に、対応するスケール係数が掛けられる。

注 : この関数は、最後のバンドの終点とスペクトル・データ・ベクトルの終点が一致する（つまり、*pSrcDst* ベクトルのサイズが *pBandsOffset*[*bands_number*] の要素に格納される）という前提で動作する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDst* または *pBandsOffset* が NULL。
ippStsSizeErr	エラー。*bands_number* がゼロ以下。

MakeFloat

仮数の配列と指数の配列を float 配列に変換する。

```
IppStatus ippsMakeFloat_16s32f (Ipp32s* inmant, Ipp32s* inexp, Ipp32s
    size, Ipp32f* outfloat);
```

引数

inmant	仮数の配列。
inexp	指数の配列。
size	配列要素の数。

outfloat 結果の float 配列の配列。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、次の式に従って、ビットストリームからデコードされた仮数の配列と指数の配列をスペクトル・サンプルの float 配列に変換する。

$$outfloat[i] = inmant[i] \times 2^{-inexp[i] - 15}$$

ここで $i = 0...size$

この変換によって、AC3 形式のビットストリームをデコードする際のアプリケーションのパフォーマンスが向上する。

戻り値

ippStsNoErr エラーなし。

変形離散コサイン変換関数

この項では、信号の変形離散コサイン変換（MDCT）を計算するインテル® IPP 関数について説明する。MDCT は、MPEG-1、MPEG-2、AC-3、AAC などの各種のオーディオ・コーデックに広く使用されている重複直交変換である。

MDCTFwdInitAlloc, MDCTInvInitAlloc

変形離散コサイン変換の指定構造体を
初期化する。

```
IppStatus ippsMDCTFwdInitAlloc_32f(IppsMDCTFwdSpec_32f** pMDCTSpec,
    int length);
IppStatus ippsMDCTInvInitAlloc_32f(IppsMDCTInvSpec_32f** pMDCTSpec,
    int length);
```

引数

pMDCTSpec 生成する MDCT 指定構造体へのポインタ。

length MDCT 中のサンプルの数。この関数はオーディオ符号化専用に設計されているため、length の値として 12、36、2^k（ここで、$k \geq 5$）のみをサポートしている。オーディオ符号化には、これらの値以外は使用されない。

説明

これらの関数は、ippac.h ヘッダ・ファイルで宣言される。
関数 ippsMDCTFwdInitAlloc と ippsMDCTInvInitAlloc は、指定された変換サイズ *length* を使用して、MDCT 指定構造体 *pMDCTSpec* を作成し、初期化する。

ippsMDCTFwdInitAlloc。関数 ippsMDCTFwdInitAlloc は順方向の MDCT 指定構造体を初期化する。

ippsMDCTInvInitAlloc。関数 ippsMDCTInvInitAlloc は逆方向の MDCT 指定構造体を初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pMDCTSpec* ポインタが NULL。
ippStsSizeErr	エラー。*length* が上記の許容される値になっていない。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

MDCTFwdFree, MDCTInvFree

離散コサイン変換の指定構造体をクローズする。

```
IppStatus ippsMDCTFwdFree_32f(IppsMDCTFwdSpec_32f* pMDCTSpec);
IppStatus ippsMDCTInvFree_32f(IppsMDCTInvSpec_32f* pMDCTSpec);
```

引数

pMDCTSpec	クローズする MDCT 指定構造体へのポインタ。

説明

これらの関数は、ippac.h ヘッダ・ファイルで宣言される。
関数 ippsMDCTFwdFree と ippsMDCTInvFree は、*ippsMDCTFwdInitAlloc* または ippsMDCTInvInitAlloc 関数によって作成された指定構造体に割り当てられたすべてのメモリを解放して、MDCT 構造体 pMDCTSpec をクローズする。

変換が完了したら、ippsMDCTFwdFree または ippsMDCTInvFree を呼び出す。

ippsMDCTFwdFree。関数 ippsMDCTFwdFree は、順方向の MDCT 指定構造体をクローズする。

ippsMDCTInvFree。関数 ippsMDCTInvFree は、逆方向の MDCT 指定構造体をクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pMDCTSpec* ポインタが NULL。
ippStsContextMatchErr	エラー。指定識別子 *pMDCTSpec* が正しくない。

MDCTFwdGetBufSize, MDCTInvGetBufSize

MDCT 作業バッファのサイズを取得する。

```
IppStatus ippsMDCTFwdGetBufSize_32f(const IppsMDCTFwdSpec_32f*
    pMDCTSpec, int* pSize);
IppStatus ippsMDCTInvGetBufSize_32f(const IppsMDCTInvSpec_32f*
    pMDCTSpec, int* pSize);
```

引数

pMDCTSpec	MDCT 指定構造体へのポインタ。
pSize	MDCT 作業バッファのサイズの値（バイト単位）のアドレス。

説明

これらの関数は、ippac.h ヘッダ・ファイルで宣言される。
関数 ippsMDCTFwdGetBufSize と ippsMDCTInvGetBufSize は、指定構造体 *pMDCTSpec* によって記述された MDCT の作業バッファのサイズを取得し、その結果を *pSize* に格納する。

ippsMDCTFwdGetBufSize。関数 ippsMDCTFwdGetBufSize は、順方向の MDCT の作業バッファのサイズを取得する。

ippsMDCTInvGetBufSize。関数 ippsMDCTInvGetBufSize は、逆方向の MDCT の作業バッファのサイズを取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pMDCTSpec* ポインタまたは *pSize* の値が NULL。
ippStsContextMatchErr	エラー。指定構造体 *pMDCTSpec* が無効。

MDCTFwd, MDCTInv

信号の順方向または逆方向の変形離散コサイン変換（MDCT）を計算する。

```
IppStatus ippsMDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst,const
    IppsMDCTFwdSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst,const
    IppsMDCTInvSpec_32f* pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTFwd_32f_I(Ipp32f* pSrcDst, const IppsMDCTFwdSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
IppStatus ippsMDCTInv_32f_I(Ipp32f* pSrcDst, const IppsMDCTInvSpec_32f*
    pMDCTSpec, Ipp8u* pBuffer);
```

引数

pSrc	入力データ配列へのポインタ。
pDst	出力データ配列へのポインタ。
pSrcDst	インプレース演算用の入力および出力データ配列へのポインタ。
pMDCTSpec	MDCT 指定構造体へのポインタ。
pBuffer	MDCT 作業バッファへのポインタ。

説明

これらの関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsMDCTFwd と ippsMDCTInv は、それぞれ順方向および逆方向の変形離散コサイン変換（MDCT）を計算する。

次の MDCT の定義で、N は長さを示し、$n_0 = (N/2+1)/2$ である。

順方向の MDCT の場合、*x*(*n*) は *pSrc*[*n*]、*y*(*k*) は *pDst*[*k*] となり、逆方向の MDCT の場合、*x*(*n*) は *pDst*[*n*]、*y*(*k*) は *pSrc*[*k*] となる。

順方向の MDCT は、次の式によって定義される。

$$y(k) = 2 \cdot \sum_{n=0}^{N-1} x(n)\cos\left(\frac{\pi}{N}(n+n_0)(2k+1)\right),\ 0 \le k < \frac{N}{2}$$ の場合。

逆方向の MDCT は、次のように定義される。

$$x(n) = \frac{2}{N} \cdot \sum_{k=0}^{\frac{N}{2}-1} y(k)\cos\left(\frac{\pi}{N}(n+n_0)(2k+1)\right),\ 0 \le n < N$$ の場合。

pBuffer 引数は、必要な作業メモリを MDCT 関数に提供し、関数内でメモリの割り当てが行われないようにする。また、キャッシュに格納された以前の操作の結果を MDCT 関数の入力配列として使用する場合、このバッファによってパフォーマンスが向上する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pMDCTSpec*、*pBuffer*、*pSrc*、*pDst*（ippsMDCTFwd_32f と ippsMDCTInv_32f の場合）または *pSrcDst*（MDCTFwd_32f_I と MDCTInv_32f_I の場合）ポインタが NULL。
ippStsContextMatchErr	エラー。指定構造体 *pMDCTSpec* が無効。
ippStsMisalignedBuf	データのアライメントが合っていない。アライメントされたデータを提供することで、パフォーマンスは向上する。

ブロック・フィルタリング関数

この項で説明するインテル® IPP 関数は、有限インパルス応答（FIR）ブロック・フィルタをサポートしている。このグループの関数を使用して、変換領域アダプティブ・フィルタを設計できる。これらのフィルタは、信号を前処理して、入力ベクトルを直交成分に分解する。これらの直交成分は、アダプティブ・サブフィルタの並列バンクに対する入力として使用される。この方法を使用して、オーディオ・コーデック（例えば、CELP および AAC）内に周波数領域の線形予測機構を実装できる。

ブロック・フィルタリング関数は、多数のベクトル（信号）を受け取る。フィルタリング関数を呼び出すたびに、各入力信号につき 1 つのフィルタリングされたサンプルが生成される。ライブラリ関数は特定の適応手法を実行しないが、フィルタリング関数を呼び出すたびにフィルタ・タップを指定できる。

FIR ブロック・フィルタ関数を使用するには、次の一般的な手順に従う。

1. FIRBlockInitAlloc を呼び出して、ブロック・フィルタのステート構造体を初期化する。
2. FIRBlockOne を呼び出して、ブロック・フィルタを使用してサンプルのベクトルをフィルタリングする。
3. FIRBlockFree を呼び出して、FIR ブロック・フィルタに割り当てられた動的メモリを解放する。

FIRBlockInitAlloc

FIR ブロック・フィルタ・ステートを初期化する。

```
IppStatus ippsFIRBlockInitAlloc_32f(IppsFIRBlockState_32f** pState,
    int order, int len);
```

引数

pState	作成される FIR ブロック・フィルタ・ステート構造体へのポインタ。
order	タップの値が入った配列の要素の数。
len	入力信号の数。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsFIRBlockInitAlloc は、FIR ブロック・フィルタ・ステートを作成し、初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

ippStsFIRLenErr	エラー。*order* または *len* がゼロ以下。

FIRBlockFree

FIR ブロック・フィルタ・ステートを
クローズする。

```
IppStatus ippsFIRBlockFree_32f(IppsFIRBlockState_32f* pState);
```

引数

pState	クローズされる FIR ブロック・フィルタ・ステート構造体へのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsFIRBlockFree は、関数 ippsFIRBlockInitAlloc によって作成されたフィルタ・ステートに割り当てられたすべてのメモリを解放して、FIR ブロック・フィルタ・ステートをクローズする。

フィルタリングが完了したら、ippsFIRBlockFree を呼び出す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsContextMatchErr	エラー。ステート構造体が無効。

FIRBlockOne

FIR ブロック・フィルタを使用して、
サンプルのベクトルをフィルタリングする。

```
IppStatus ippsFIRBlockOne_32f(Ipp32f* pSrc, Ipp32f* pDst,
    IppsFIRBlockState_32f* pState, Ipp32f *pTaps);
```

引数

pSrc	フィルタリングするサンプルの入力ベクトルへのポインタ。
pDst	フィルタリングされた出力サンプルのベクトルへのポインタ。
pState	FIR フィルタ・ステート構造体へのポインタ。
pTaps	フィルタ・タップのベクトルへのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsFIRBlockOne は、フィルタを使用して、長さ *len* のサンプルのベクトル *pSrc* をフィルタリングし、その結果を *pDst* に格納する。

フィルタ・タップは、長さ *order* のベクトル *pTaps* で指定される。*len* および *order* パラメータの値は、FIRBlockInitAlloc コールで指定される。

次の FIR フィルタの定義では、遅延 k を使用してフィルタリングされる入力ベクトル i のサンプルは h_k として示され、タップは として示されている。x^i_{n-k} 出力値 y_i は、次の式で定義される。

$$y^i_n = \sum_{k=0}^{order-1} h_k x^i_{n-k}, \quad 0 \le i < len$$

関数 ippsFIRBlockOne を呼び出す前に、関数 ippsFIRBlockInitAlloc を呼び出してフィルタ・ステートを初期化する。タップの値は、引数 *pTaps* で指定する。

例 10-1 ippsFIRBlockOne 関数によるシングルレート・フィルタリング

```
IppStatus fir(void)
{
#undef NUMITERS
#define NUMITERS 20
#undef BLOCKSIZE
#define BLOCKSIZE 20
    int n;
    int i;
    IppStatus status;
    IppsFIRBlockState_32f *fctx;
    Ipp32f x [BLOCKSIZE], y [BLOCKSIZE];
        const float taps[] = {
        0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
```

例 10-1 ippsFIRBlockOne 関数によるシングルレート・フィルタリング

```
            0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0
    };
    ippsFIRBlockInitAlloc_32f( &fctx, 11, BLOCKSIZE );
    for (n =0;n<NUMITERS;++n)
    {
        for (i = 0; i< BLOCKSIZE; i++) x[i]=(Ipp32f)sin(IPP_2PI *
            n *0.2 + i);
        status = ippsFIRBlockOne_32f( x, y, fctx, (Ipp32f*)taps );
        for (i = 0; i< BLOCKSIZE; i++)
            printf("%f", y[i]);
    }
    ippsFIRBlockFree_32f(fctx);
    return status;
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、*pState*、または *pTaps* が NULL。
ippStsContextMatchErr	エラー。ステート構造体が無効。
ippStsMisalignedBuf	配列のアライメントが合っていない。アライメントされたデータを提供することで、パフォーマンスは向上する。
ippStsFIRLenErr	エラー。*tapsLen* がゼロ以下。

周波数領域予測関数

MPEG-2, 4 AAC エンコーダは、周波数領域内の予測（FDP）を使用して、オーディオ信号の冗長性を削減し、より効果的なコード化を可能にする。各スペクトル線について、予測機構と呼ばれる 2 次アダプティブ FIR フィルタを使用して、入力信号がフィルタリングされる。次に、信号を処理する代わりに、元の信号とフィルタ出力の差（すなわち、予測エラー）が、将来の処理のために渡される。デコーダは、対称ブロック行列を使用して、予測エラーから元の信号を復元する。

定期的に予測機構を初期状態にリセットして、累積された計算誤差を小さくする必要がある。また、ISO-144963 規格に規定されている特殊な場合にも、予測機構をリセットする必要がある。スペクトル全体、複数のスケール係数バンド、または選択したスペクトル線グループについて、予測機構をリセットできる。

フィルタ係数適応アルゴリズムと FDP の使用法の詳細は、ISO-144963、条項 6.5.3.2 を参照のこと。

この項で説明する FDP 予測ツール関数を使用するには、次の手順に従う。

1. 関数 FDPInitAlloc を呼び出して、メモリを割り当て、予測機構のステートを初期化する。
2. 各フレームについて、関数 FDPFwd を呼び出して予測エラーを計算するか、関数 FDPInv を呼び出して元の信号を復元する。
3. ステート作成後、任意の時点で関数 ResetFDP、ResetFDP_SFB、または ResetFDPGroup を呼び出し、指定したスペクトル線の予測機構をリセットする。
4. 関数 FDPFree を呼び出して、ippsFDPInitAlloc によって割り当てられたメモリを解放する。

FDPInitAlloc

予測機構のステートを初期化する。

```
IppStatus ippsFDPInitAlloc_32f(IppsFDPState_32f **pFDPState, int len);
```

引数

pFDPState	生成する FDP ステート構造体へのポインタ。
len	処理されるスペクトル線の数。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsFDPInitAlloc は FDP ステートを作成し、初期化する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。関数に渡される 1 つ以上のポインタが NULL。
ippStsSizeErr	エラー。長さがゼロ以下。
ippStsMemAllocErr	エラー。メモリが割り当てられていない。

FDPFree

FDP ステートをクローズする。

```
IppStatus ippsFDPFree_32f(IppsFDPState_32f *pFDPState);
```

引数

pFDPState	クローズする FDP ステート構造体へのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsFDPFree は、関数 FIRBlockInitAlloc によって作成された FDP ステート構造体に割り当てられたすべてのメモリを解放して、FDP ステートをクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。関数に渡される 1 つ以上のポインタが NULL。
ippStsContextMatchErr	エラー。ステート構造体が無効。

ResetFDP

すべてのスペクトル線の予測機構を
リセットする。

```
IppStatus ippsResetFDP_32f(IppsFDPState_32f *pFDPState);
```

引数

pFDPState	ステート構造体へのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsResetFDP は、すべてのスペクトル線の予測機構をリセットする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。関数に渡される 1 つ以上のポインタが NULL。
ippStsContextMatchErr	エラー。ステート構造体が無効。

ResetFDP_SFB

指定されたスケール係数バンドの
予測機構固有の情報をリセットする。

```
IppStatus ippsResetFDP_SFB_32f (IppsFDPState_32f* pFDPState, const int*
    pBandsOffset, int bands_number, const Ipp8u *reset_flag);
```

引数

pFDPState	ステート構造体へのポインタ。
pBandsOffset	バンド・オフセット・ベクトルへのポインタ。
bands_number	バンドの数。
reset_flag	特定のスケール係数バンド内のスペクトル線の予測機構をリセットする必要があるかどうかを示すフラグの配列。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsResetFDP_32f は、*reset_flag*[*i*] がゼロでない各スケール係数バンド *i* 内のすべてのスペクトル線の予測機構をリセットする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。関数に渡される 1 つ以上のポインタが NULL。
ippStsSizeErr	エラー。*bands_number* がゼロ以下。
ippStsContextMatchErr	エラー。ステート構造体が無効。

intel®

ResetFDPGroup

スペクトル線のグループの予測機構を
リセットする。

```
IppStatus ippsResetFDPGroup_32f (IppsFDPState_32f* pFDPState, int
    start, int step);
```

引数

pFDPState	ステート構造体へのポインタ。
start	グループ内の最初のスペクトル線のオフセット。
step	グループ内の 2 つの隣接するスペクトル線の間の距離。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsResetFDPGroup は、スペクトルの始まりから終わりまでの *step* 番目ごとのスペクトル線の予測機構をリセットする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。関数に渡される 1 つ以上のポインタが NULL。
ippStsSizeErr	エラー。*reset_group_number* または *step* がゼロ以下。
ippStsContextMatchErr	エラー。ステート構造体が無効。

FDPFwd

周波数領域予測手順を実行し、
予測エラーを計算する。

```
IppStatus ippsFDPFwd_32f(IppsFDPState_32f* pFDPState, Ipp32f* pSrc,
    Ipp32f* pDst);
```

引数

pFDPState	ステート構造体へのポインタ。
pSrc	入力データ配列へのポインタ。
pDst	予測エラーを格納するデータ配列へのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsFDPFwd は、入力信号 *pSrc* に周波数領域予測手順を適用し、予測エラーを *pDst* ベクトルに格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。関数に渡される 1 つ以上のポインタが NULL。
ippStsContextMatchErr	エラー。ステート構造体が無効。
ippStsMisalignedBuf	配列のアライメントが合っていない。アライメントされたデータを提供することで、パフォーマンスは向上する。

FDPInv

周波数領域予測手順を使用して、予測エラーから入力信号を復元する。

```
IppStatus ippsFDPInv_32f(IppsFDPState_32f* pFDPState, Ipp32f* pSrcDst,
    const int* pBandsOffset, int bands_number, const Ipp8u*
    prediction_used);
```

引数

pFDPState	ステート構造体へのポインタ。
pSrcDst	インプレース演算用の入力および出力データ配列へのポインタ。
pBandsOffset	バンド・オフセット・ベクトルへのポインタ。
bands_number	スケール係数バンドの数。
prediction_used	特定のスケール係数バンド内で予測を使用するかどうかを示すフラグの配列。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsFDPInv は、入力スペクトル・ベクトル *pSrcDst* の特定のバンドに周波数領域予測手順を適用する。バンドの位置は、パラメータ *bands_number* および *pBandsOffset* で定義される。

各スケール係数バンド *i* について、*prediction_used[i]* がゼロでない場合は、そのバンド内の *pSrcDst* のすべての値が予測エラーとして扱われ、元の信号が復元される。*prediction_used[i]* がゼロの場合は、そのバンド内の *pSrcDst* のすべての値が信号として扱われ、変更なしに渡される。

各予測機構の係数は、*prediction_used* フラグの値に関係なく更新される。

注： この関数は、最後のバンドの終点とスペクトル・データ・ベクトルの終点が一致する（つまり、*pSrcDst* ベクトルのサイズが *pBandsOffset[bands_number]* の要素に格納される）前提で動作する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。関数に渡される 1 つ以上のポインタが NULL。
ippStsSizeErr	エラー。*bands_number* がゼロ以下。
ippStsContextMatchErr	エラー。ステート構造体が無効。
ippStsMisalignedBuf	配列のアライメントが合っていない。アライメントされたデータを提供することで、パフォーマンスは向上する。

ハフマン・アルゴリズム関数

オーディオ・データの圧縮にはいくつかの方法がある。

ハフマン符号化は、統計的モデリングを使用して、他のコードより発生頻度が高いコードを定義し、その後のエンコードおよびデコード操作のためのテーブルを作成するデータ・サイズ削減手法である。ビットストリーム内のオーディオ・データは、最も短いコードが最も頻度の高い値に対応し、長いコードが頻度の低い値に対応するように、可変長コード（VLC）テーブルを使用してエンコードされる。ハフマン符号化を使用するすべての規格は、可能なコードとその値を示すテーブルを持っている。

例 10-2　可変長コード（VLC）テーブルの形式

```
static Ipp32s Table[]=
{
max_bits,              The maximum length of code
total_subt,            The total number of all subtables
sub_sz1,               The sizes of all subtables.Their sum must
                       be equal to the maximum length of code.
sub_sz2,
…,

sub_szTotal,
N1,                    The number of 1-bit codes
code1, value1,         The 1-bit codes and values.The number of
                       pairs must be equal to N1.
code2, value2,
...
codeN1, valueN1,
N2,                    The number of 2-bit codes
code1, value1,         The 2-bit codes and values.The number of
                       pairs must be equal to N2.
code2, value2,
...
codeN2, valueN2,
....
Nm,                    The number of maximum length codes.
code1, value1,         The m-bit codes and values.The number of
                       pairs must be equal to Nm.

code2, value2,
...
codeNm, valueNm,
-1                     The significant value to indicate the end
                       of table
};
```

ハフマン・アルゴリズムは、例えば MPEG-1 レイヤ 3 と AAC における MP3 サンプルと AAC の符号化や AAC スケール係数の符号化などに、広く使用されている。

GetSizeHDT

ハフマン・デコード・テーブルのサイズを
計算する。

```
ippsGetSizeHDT_32s (const Ipp32s* pInputTable, Ipp32s* pBuffer, Ipp32s
    buffSize, Ipp32s *pSize);
```

引数

pInputTable	指定されたハフマン・テーブルへのポインタ（入力パラメータ）。
pBuffer	計算のための一時バッファへのポインタ。
buffSize	一時バッファのサイズ。
pSize	ハフマン・デコード・テーブルのサイズ（出力パラメータ）。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、関数 ippsBuildHDT の呼び出しに必要なハフマン・デコード・テーブルのサイズを計算する。

パラメータ *buffSize* は、$2^{maxlen_of_code} \cdot sizeof(int)$ と同じか、それ以上でなければばならない。ここで、*maxlen_of_code* は、指定されたハフマン・テーブル内のコードの最大長である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。関数に渡される 1 つ以上のポインタが NULL。

注： パラメータ *buffSize* が $2^{maxlen_of_code} \cdot sizeof(int)$ より小さい場合は、この関数は正常に動作しない。この場合、エラー・コードは返されない。

BuildHDT

ハフマン・デコード・テーブルを構築する。

```
ippsBuildHDT_32s (const Ipp32s* pInputTable, Ipp32s* pDecodeTable,
    Ipp32s size);
```

引数

pInputTable	指定されたハフマン・テーブルへのポインタ（入力パラメータ）。

pDecodeTable 結果のハフマン・テーブルが構築されるメモリへのポインタ。

size テーブルのサイズ。

説明

この関数は、`ippac.h` ヘッダ・ファイルで宣言される。この関数は、関数 `ippsGetVLC` の呼び出しに必要なハフマン・デコード・テーブルを構築する。この関数を呼び出す前に、デコード・テーブル用のメモリを割り当て、関数 `ippsGetHDT` を呼び出してテーブルのサイズを計算する必要がある。

図 10-1 VLC テーブルおよびサブテーブル

コード	値
0	A
1100	B
101	C
1110	D
100	E
11110	F
11111	G
1101	H

コード	000	001	010	011	100	101	110	111
値	A	A	A	A	E	C		

コード	00	01	10	11
値	B	B	H	H

コード	00	01	10	11
値	D	D	F	G

戻り値

`ippStsNoErr` エラーなし。

`ippStsNullPtrErr` エラー。関数に渡される 1 つ以上のポインタが NULL。

DecodeVLC

ハフマン・テーブルから値をデコードする。

```
ippsDecodeVLC_32s (Ipp32s** pBitStream, Ipp32u* offset, Ipp32s*
    pDecodeTable, Ipp32s* pData);
```


intel®

```
ippsDecodeVLC_Block_32s,(Ipp32s **pBitStream, Ipp32u* offset, Ipp32s
    *pDecodeTable, Ipp32u length, Ipp16s *pData);
ippsDecodeVLC_MP3ESCBlock_32s,(Ipp32s **pBitStream, Ipp32u* offset,
    Ipp32s *pDecodeTable, Ipp32u length, Ipp32u linbits, Ipp16s
    *pData);
ippsDecodeVLC_AACESCBlock_32s,(Ipp32s **pBitStream, Ipp32u* offset,
    Ipp32s *pDecodeTable, Ipp32u length, Ipp16s *pData);
```

引数

pBitStream	ビットストリームへのポインタ。
offset	*pBitStream* が指すビットとコードの始点の間のオフセットへのポインタ。
pDecodeTable	デコードに使用されるハフマン・テーブルへのポインタ。
length	デコード対象のコードの数。
linbits	エスケープ・コードのサイズ。
pData	デコードされた値が格納される変数へのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、関数 ippsBuldHDT によって作成されたテーブルを使用して、ビットストリームを解析して可変長コードをデコードし、ポインタを新しい位置にリセットする。ポインタ *pBitStream* は 32 ビット値を指す。ビット・オフセットは 1 ～ 32 の範囲で変化する。処理の終了後、ポインタは変更され、新しい値が返される。

ippsDecodeVLC_32s。関数 ippsDecodeVLC_32s は、見つかったコードに対応するテーブル値を返す。

ippsDecodeVLC_Block_32s

ippsDecodeVLC_AACESCBlock_32s
ippsDecodeVLC_MP3Block_32s

関数 ippsDecodeVLC_Block_32s、ippsDecodeVLC_AACESCBlock_32s、および ippsDecodeVLC_MP3Block_32s は、各コードに 2 つの値を割り当ててブロックをデコードし、指定された配列に格納する。2 つの値を得るには、コードに対応する値の第 2 バイトから値 *X* を抽出し、第 1 バイトから値 *Y* を抽出する。次に、この関数は、必要に応じて符号ビットとエスケープ・シーケンスを処理し、得られた値を配列に格納する。

VLC テーブルの構造については、例 10-2 を参照のこと。VLC テーブルとサブテーブルの例については、図 10-1 を参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。関数に渡される 1 つ以上のポインタが NULL。
ippStsVLCInputDataErr	エラー。エンコード / デコード関数で使用された入力が正しくない。デコード関数の場合、使用されたテーブル内で指定されていないコードがビットストリームに含まれることを示す場合もある。
ippStsVLCAACEscCodeLengthErr	エラー。ビットストリームに含まれている AAC-Esc コードの長さが 21 より大きい。

GetSizeHET

入力ハフマン・テーブルに必要なサイズを計算する。

```
IppStatus ippsGetSizeHET_16s(const Ipp16s* pInputTable,Ipp32s * pSize);
```

引数

pInputTable	指定された形式の入力ハフマン・テーブル。
pSize	内部テーブル・サイズの値（バイト単位）のアドレス。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言され。関数 ippsGetSizeHET は、内部形式の入力ハフマン・テーブルに必要なサイズを計算する。サイズはバイト単位で計算される。

例 10-3 は、指定されたテーブル形式を示している。

例 10-3　ユーザ形式のハフマン・テーブル

```
short huf_tabX[] = {
value1 , // max value in table
value2, //  length of  ESC-code
```

例 10-3　ユーザ形式のハフマン・テーブル

```
/* x, y, length of code in bit + sign bits, code */
0, 0,  2, 0x3,
0, 1,  2, 0x2,
………….
2, 2, 6, 0x0,
-1 /* end of table */
};
```

戻り値

ippStsNoErr　　エラーなし。

注： コードの length を指定するテーブルの列には、符号情報を格納するビットも含まれていなければならない。

BuildHET

ハフマン・テーブルを構築する。

```
IppStatus ippsBuildHET_16s (const Ipp16s* pInputTable, Ipp16s *
    pInternalTable);
```

引数

pInputTable	指定された形式の入力ハフマン・テーブル。
pInternalTable	内部形式の出力ハフマン・テーブル。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsBuildHET_16s は、内部形式のハフマン・テーブルを構築する。テーブルのための十分なメモリ（関数 ippsGetSizeHET によって返されるサイズ）をバイト単位で割り当て、割り当てられたメモリを *pInternalTable* パラメータとして取得する必要がある。

ハフマン・エンコード演算が完了したら、割り当てられたメモリを解放する必要がある。

戻り値

ippStsNoErr　　エラーなし。

HuffmanCountBits

量子化された値のエンコードに必要なサイズを計算する。

```
IppStatus ippsHuffmanCountBits_16s(const Ipp16s* pInputData, Ipp32s
    length, const Ipp16s* pInternalTable, Ipp16s* pCountBits);
```

引数

pInputData	量子化されたスペクトル値の入力配列へのポインタ。
length	指定されたハフマン・テーブルを使用してエンコードされる入力配列のサイズ
pInternalTable	内部形式のハフマン・テーブルへのポインタ。
pCountBits	必要なサイズの値（ビット単位）へのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。関数 ippsHuffmanCountBits は、エンコード操作の内側反復ループで量子化値をエンコードするのに必要なサイズをビット単位で計算する。

コードの length を指定するテーブルの列には、符号情報を格納するビットも含まれていなければならない（例 10-3 を参照）。

入力配列の値は、ハフマン・テーブルで指定される最大値を超えてはならない。関数 ippsThreshold_GT を使用して、入力データが最大値の必要条件を満たしているかどうか確認できる。

この関数は、入力データとしてエスケープ・テーブルを受け取った場合、全コード長にエスケープ・コードの長さを加算する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。

EncodeVLC

指定されたハフマン・テーブルを使用して、
スペクトル値のブロックをエンコードする。

```
IppStatus ippsEncodeVLC_Block_16s(Ipp16s* pInputData, Ipp32s len, const
    Ipp16s*  pInternalTable, Ipp32s **pBitStream, Ipp32u* offset);
IppStatus ippsEncodeVLC_MP3ESCBlock_16s(Ipp16s* pInputData, Ipp32s len,
    const Ipp16s*  pInternalTable, Ipp32s **pBitStream, Ipp32u*
    offset);
```

引数

pInputData	量子化されたスペクトル値の入力配列。
len	入力ハフマン・テーブルを使用してエンコードされる入力配列の値。
pInternalTable	内部形式のハフマン・テーブル。
pBitStream	バッファ内の現在のダブルワードへのポインタ。
offset	現在のダブルワード内の未読ビット数へのポインタ

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、指定されたハフマン・テーブルを使用して、スペクトル値のブロックをエンコードする。

非エスケープ・テーブルによるエンコードには、関数 ippsEncodeVLC_Block_16s が使用される。

エスケープ・テーブルによるエンコードには、関数 ippsEncodeVLC_ESCBlock_16s が使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。

`ippStsVLCInputDataErr`	エラー。エンコード / デコード関数で使用した入力が正しくない。デコード関数の場合、使用したテーブル内で指定されていないコードがビットストリームに含まれていることを示す場合もある。

GetSizeHET_VLC

内部形式の入力ハフマン・テーブルに必要なサイズを計算する。

```
IppStatus ippsGetSizeHET_VLC_32s(const Ipp32s* pInputTable, Ipp32s * pSize);
```

引数

pInputTable	指定された形式の入力ハフマン・テーブル。
pSize	内部テーブル・サイズの値（バイト単位）のアドレス。

説明

この関数は、`ippac.h` ヘッダ・ファイルで宣言される。この関数は、内部形式の入力ハフマン・テーブルに必要なサイズを計算する。サイズはバイト単位で計算される。

例 10-4　ユーザ形式のハフマン・テーブル

```
Ipp32s    huff_table[] =
{
    value0, // Number of parameters in header.
    value1, // n-tuple size, can be 1, 2, 4 (required parameter)
    value2, // Unsigned table : 1 - yes, 0 - no (required parameter)
    value3, // Number of entries in table (required parameter)

    value4, // Esc-table type (optional parameter)
    value5, // Esc-code (optional parameter)
    value6, // Esc-code length (optional parameter)
        /*   x,    y,    w,    z, length, VLC-value   */
-1,  -1,   1,    0,        8,      0x00e8,
…
}
```

例 10-5 Esc-table type

```
ippVLCNonEscAlg = 0,
ippVLCMp3EscAlg = 1,
ippVLCAACEscAlg = 2
```

Esc アルゴリズムが使用される場合、対応するハフマン・テーブル内に Esc コードを含むタプル（組）が 1 つ以上含まれている必要がある。このため、各 Esc パラメータ（Esc テーブル・タイプ、Esc コード、Esc コードの長さ）は、テーブルでそれぞれ別々にリストされる。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	データ配列へのポインタが NULL。
`ippStsVLCUsrTblHeaderErr`	エラー。ユーザ・テーブルのヘッダが無効。
`ippStsVLCUsrTblUnsupportedFmtErr`	エラー。ユーザ・テーブルのヘッダで指定されたタプルのサイズ *n* が正しくない。
`ippStsVLCUsrTblEscAlgTypeErr`	エラー。ユーザー・テーブルで指定された Esc コードのビルド・アルゴリズムがサポートされていない。
`ippStsVLCUsrTblCodeLengthErr`	エラー。ユーザ・テーブルに含まれたコードの長さがサポートされていない (例えば、コードの長さが 32 より大きい)。

BuildHET_VLC

内部形式のハフマン・テーブルをビルドする。

```
IppStatus ippsBuildHET_VLC_32s(Ipp32s* pInputTable, Ipp32s*
    pInternalTable);
```

引数

pInputTable	指定された形式の入力ハフマン・テーブル。
pInternalTable	内部形式の出力ハフマン・テーブル。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、内部形式のハフマン・テーブルを構築する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsVLCUsrTblHeaderErr	エラー。ユーザ・テーブルのヘッダが無効。
ippStsVLCUsrTblUnsupportedFmtErr	エラー。ユーザ・テーブルのヘッダで指定されたタプルのサイズ *n* が正しくない。
ippStsVLCUsrTblEscAlgTypeErr	エラー。ユーザー・テーブルで指定された Esc コードのビルド・アルゴリズムがサポートされていない。
ippStsVLCUsrTblCodeLengthErr	エラー。ユーザ・テーブルに含まれたコードの長さがサポートされていない(例えば、コードの長さが 32 より大きい)。

CountBits

入力配列をエンコードするのに必要な
サイズをビット単位で計算する。

```
IppStatus ippsCountBits_1tuple_VLC_16s(Ipp16s* pInputData, Ipp32s
    length, const Ipp32s * pInternalTable, Ipp16s* pCountBits);
IppStatus ippsCountBits_2tuple_VLC_16s(Ipp16s* pInputData, Ipp32s
    length, const Ipp32s * pInternalTable, Ipp16s* pCountBits);
IppStatus ippsCountBits_4tuple_VLC_16s(Ipp16s* pInputData, Ipp32s
    length, const Ipp32s * pInternalTable, Ipp16s* pCountBits);
```

引数

pInputData	入力配列へのポインタ。
length	エンコードされる入力配列のサイズ。
pInternalTable	内部形式のハフマン・テーブルへのポインタ。

pCountBits	必要なサイズの値（ビット単位）へのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、入力配列をエンコードするのに必要なサイズをビット単位で計算する。

ippsCountBits_2tuple_VLC_16s。2 タプル配列には、関数 ippsCountBits_2tuple_VLC_16s が使用される。

ippsCountBits_4tuple_VLC_16s。4 タプル配列には、関数 ippsCountBits_4tuple_VLC_16s が使用される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。
ippStsVLCInternalTblErr	エラー。エンコードで使用された内部テーブルが破損しているか、またはサポートされていない。例えば、4 タプルの関数で 2 タプルのテーブルを使用した場合など。

EncodeBlock

入力配列をエンコードする。

```
IppStatus ippsEncodeBlock_1tuple_VLC_16s(Ipp16s* pInputData, Ipp32s
    length, const Ipp32s * pInternalTable, Ipp32s** ppBitstream,
    Ipp32u* pOffset );
IppStatus ippsEncodeBlock_2tuple_VLC_16s(Ipp16s* pInputData, Ipp32s
    length, const Ipp32s * pInternalTable, Ipp32s** ppBitstream,
    Ipp32u* pOffset );
IppStatus ippsEncodeBlock_4tuple_VLC_16s(Ipp16s* pInputData, Ipp32s
    length, const Ipp32s * pInternalTable, Ipp32s** ppBitstream,
    Ipp32u* pOffset );
```

引数

pInputData	入力配列へのポインタ。
length	エンコードされる入力配列のサイズ。

pInternalTable	内部形式のハフマン・テーブルへのポインタ。
ppBitstream	バッファ内にある現在のダブルワードへのポインタ。
pOffset	現在のダブルワード内の未読ビット数へのポインタ

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。

ippsEncodeBlock_1tuple_VLC_16s。関数 ippsEncodeBlock_1tuple_VLC_16s は、指定されたハフマン・テーブルを使用して入力配列をエンコードする。

ippsEncodeBlock_2tuple_VLC_16s。2 タプルの非エスケープ・テーブルによるエンコードには、関数 ippsEncodeBlock_2tuple_VLC_16s が使用される。

ippsEncodeBlock_4tuple_VLC_16s。4 タプルの非エスケープ・テーブルによるエンコードには、関数 ippsEncodeBlock_4tuple_VLC_16s が使用される。

例 10-6 上記で説明したハフマン符号化関数の使用例

```
On initialization step:
    ippsGetSizeHET_VLC_32s(…);
    /// memory allocation
    ippsBuildHET_VLC_32s(…);
…
On counting bits step:

    Do
    {
        bit_number = ippsCountBits_XXX_VLC_16s(…)
        …
} While (bit_number > allowed_bit_number);

On bit stream step :

        ippsEncode_XXX_VLC_16s(…);
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	データ配列へのポインタが NULL。

ippStsVLCInternalTblErr	エラー。エンコードで使用された内部テーブルが破損しているか、またはサポートされていない。例えば、4 タプルの関数で 2 タプルのテーブルを使用した場合など。
ippStsVLCInputDataErr	エラー。エンコード/デコード関数で使用した入力が正しくない。デコード関数の場合、テーブル内で指定されていないコードがビットストリームに含まれていることを示す場合もある。

ベクトル量子化関数

この項では、ベクトル量子化で使用する関数について説明する。

CdbkInitAlloc

コードブック構造体を初期化する。

```
IppStatus ippsCdbkInitAlloc_VQ_32f(IppsCdbkState_VQ_32f** pCdbk, const
    Ipp32f* pSrc, int step, int height, Ipp_Cdbk_VQ_Hint hint);
```

引数

pCdbk	作成されるコードブック構造体へのポインタ。
pSrc	サイズが *step * height* のテーブル *Cdbk* へのポインタ。このテーブルは、長さ *step* の量子化ベクトルを *height* 個含む。
step	テーブル *pSrc* 内の次の行へのステップ。量子化ベクトルの長さ。
height	テーブルの高さ。量子化ベクトルの数。
hint	予約済みのパラメータ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、コードブックと検索操作で必要となる追加情報を格納する構造体を初期化する。この構造体は、PreSelect_VQ、MainSelect_VQ、IndexSelect_VQ、および VectorReconstruction_VQ 関数によるベクトル量子化で使用される。

この関数で割り当てられたメモリを解放するには、関数 CdbkFree を使用する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pCdbk* または *pSrc* ポインタが NULL。

CdbkFree

CdbkInitAlloc で作成した *IppsCdbkState_VQ_32f* 構造体をクローズする。

```
IppStatus ippsCdbkFree_VQ_32f(IppsCdbkState_VQ_32f* pCdbk);
```

引数

pCdbk	指定された IppsCdbkState_VQ_32f 構造体へのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、CdbkInitAlloc で作成した IppsCdbkState_VQ_32f 構造体をクローズする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pCdbk* または *pSrc* ポインタが NULL。

PreSelect_VQ

コードブックの最も近いコード・ベクトルの候補を選択する。

```
IppStatus ippsPreSelect_VQ_32f(const Ipp32f* pSrc, const Ipp32f*
    pWeight, int nDiv, const Ipp32s* pLengths, Ipp32s* pIndx, Ipps32s*
    pSign, IppsCdbkState_32f* pCdbk, int nCand, int* polbits);
```

引数

pSrc	量子化するソース・ベクトル。

pWeight	重みベクトルへのポインタ。
nDiv	*src* と *weights* ベクトルのフラグメンテーションの数。
pLengths	フラグメンテーションの長さの配列へのポインタ。
pIndx	*cCand* 個の最小候補のインデックスの出力ベクトルへのポインタ。
pSign	*nCand* 個の最小候補の符号の出力ベクトルへのポインタ。値が 1 であれば、ノルムが負の場合に歪みが最小限であることを示す。値が 0 であれば、ノルムが正の場合に歪みが最小限であることを示す。
pCdbk	指定された IppsCdbkState_VQ_32f 構造体へのポインタ。
nCand	出力候補の数。
polbits	*polbits* フラグ・ベクトルへのポインタ。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、インデックスを計算し、最も近い値を持つ *nCand* 個のベクトルをコードブックから検索する。

$$dist_p[idiv]\{icb\} = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] - pSrcDiv[ismp])^2$$

polbits[*idiv*] が MAXBIT_SHAPE より大きい場合、次の歪みも計算される。

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

但し、*idiv* = 0 〜 *nDiv* - 1。*icb* はコードブックの行数を示す。

これらの式について説明する。

- *pSrcDiv* は、次の式で計算されるストリーム内の *idiv* フラグメンテーション開始へのポインタである。

$$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s]$$

- *pWeightDiv* は、フラグメンテーションの重み配列へのポインタである。

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s]$$

- *ppTable* は、量子化するベクトルのセットを含むテーブルへのポインタである。ここで、*icb* はベクトルの番号、*ismp* は指定されたベクトルの要素数である。このテーブルは、関数 CdbkInitAlloc で初期化される *pCdbk* 構造体の一部である。

次に、この関数は歪みの尺度が最小限の候補を *nCand* 個（*dist* から $dist_n$ まで）選択する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pCdbk* または *pSrc* ポインタが NULL。

MainSelect_VQ

歪みが最小限の最適なインデックスを検索する。

```
IppStatus ippsMainSelect_VQ_32f(const Ipp32f* pSrc, const Ipp32f*
    pWeight, const Ipp32s* pLengths, int nDiv, int nCand, Ipp32s**
    pIndexCand, Ipps32s** pSignCand, Ipp32s** pIndx, Ipp32s** pSign,
    IppsCdbkState_32f** pCdbks, int nCdbks);
```

引数

pSrc	量子化するソース・ベクトル。
pWeight	長さのベクトルへのポインタ。
pLengths	フラグメンテーションの長さの配列へのポインタ。
nDiv	*src* と *weights* ベクトルのフラグメンテーションの数。
nCand	入力候補の数。
pIndexCand	*nCand* 個の最小候補のインデックスの入力ベクトルへのポインタ。
pSignCand	*nCand* 個の最小候補の符号の入力ベクトルへのポインタ。
pIndx	インデックスの出力ベクトルへのポインタ。
pSign	符号の出力ベクトルへのポインタ。

intel®

pCdbks 指定された `IppsCdbkState_VQ_32f` 構造体へのポインタ。

nCdbks コードブックの数。

説明

この関数は、`ippac.h` ヘッダ・ファイルで宣言される。この関数は、指定されたコードブックの数に基づいて、すべての可能なインデックスの組み合わせにおけるベクトルを復元し、ソース・ベクトルに対して歪みを計算する。次に、歪みが最小限である組み合わせを返す。

次の式は、量子化ベクトル *icb*[*i*] を指定されたフラグメンテーション *idiv* の歪みを計算する。ここで、*i* は [0,*nCdbks*-1] の範囲である。

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp](rec[ismp] - pSrcDiv[ismp])^2$$

ここで、

$$rec[ismp] = \frac{\sum_{i=0}^{nCdbks-1} pSignCand[idiv*nCand+icb[i]]*ppTable[pIndexCand[idiv*nCand+icb[i]]][ismp]}{nCdbks}$$

これらの式について説明する。

- *pSrcDiv* は、次の式で計算されるストリーム内の *idiv* フラグメンテーション開始へのポインタである。

  $$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s]$$

- *pWeightDiv* は、フラグメンテーションの重み配列へのポインタである。

  $$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s]$$

- *ppTable* は、量子化するベクトルのセットを含むテーブルへのポインタである。ここで、*icb* はベクトルの番号、*ismp* は指定されたベクトルの要素数である。このテーブルは、関数 CdbkInitAlloc で初期化される *pCdbk* 構造体の一部である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pCdbk* または *pSrc* ポインタが NULL。

IndexSelect_VQ

指定されたコードブックに対して最適な
ベクトルのセットを検索する。

```
IppStatus ippsIndexSelect_VQ_32f(const Ipp32f* pSrc, const Ipp32f*
    pWeight, int nDiv, const Ipp32s* pLengths,  int nCand, int**
    polbits, Ipp32s** pIndx, Ipp32s** pSign, IppsCdbkState_VQ_32f**
    pCdbks, int nCdbks);
```

引数

pSrc	量子化するソース・ベクトル。
pWeight	長さのベクトルへのポインタ。
nDiv	*src* と *weights* ベクトルのフラグメンテーションの数。
pLengths	フラグメンテーションの長さの配列へのポインタ。
nCand	入力候補の数。
polbits	指定された数のコードブックの最適なベクトルのセットを計算するために必要なノルムの数（1 または 2）を示す。
pIndx	インデックスの出力ベクトルへのポインタ。
pSign	符号の出力ベクトルへのポインタ。値が 1 であれば、ノルムが負の場合に歪みが最小限であることを示す。値が 0 であれば、ノルムが正の場合に歪みが最小限であることを示す。
pCdbks	指定された IppsCdbkState_VQ_32f 構造体へのポインタ。
nCdbks	コードブックの数。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、次の式に従って各コードブックで *nCand* 個のベクトルを計算する。

$$dist_p[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot [ppTable[icb][ismp] - pSrcDiv[ismp]]^2$$

polbits[*idiv*] が 1 である場合、次の歪みも計算される。

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

但し、idiv=0 〜 *nDiv* -1。*icb* はコードブックの行数を示す。

その後、この関数は指定されたコードブックの数に基づいて、すべての可能なインデックスの組み合わせにおけるベクトルを復元し、ソース・ベクトルに対して歪みを計算する。次に、歪みが最小限である組み合わせを返す。次の式は、量子化ベクトル *icb*[*i*] を使用して、指定されたフラグメンテーション *idiv* の歪みを計算する。ここで、*i* は [0,*nCdbks*-1] の範囲である。

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp](rec[ismp] - pSrcDiv[ismp])^2$$

$$rec[ismp] = \frac{\sum_{i=0}^{nCdbks-1} pSignCand[idiv*nCand + icb[i]]*ppTable[pIndexCand[idiv*nCand + icb[i]]][ismp]}{nCdbks}$$

但し、idiv=0 〜 *nDiv* -1。*icb* はコードブックの行数を示す。

これらの式について説明する。

- *pSrcDiv* は、次の式で計算されるストリーム内の *idiv* フラグメンテーション開始へのポインタである。

  $$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s]$$

- *pWeightDiv* は、フラグメンテーションの重み配列へのポインタである。

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s]$$

- *ppTable* は、量子化するベクトルのセットを含むテーブルへのポインタである。ここで、*icb* はベクトルの番号、*ismp* は指定されたベクトルの要素数である。このテーブルは、関数 CdbkInitAlloc で初期化される *pCdbk* 構造体の一部である。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pCdbk* または *pSrc* ポインタが NULL。

注： PreSelect および MainSelect の代わりにこの関数を呼び出すことで、処理時間とメモリの消費量を削減することができる。

VectorReconstruction_VQ

インデックスからベクトルを再構築する。

```
IppStatus ippsVectorReconstruction_VQ_32f(const Ipp32s** pIndx, const
    Ipp32s** pSign, const Ipp32s* pLength, int nDiv,
    IppsCdbkState_VQ_32f** pCdbk, int nCdbks, Ipp32f* pDst);
```

引数

pIndx	各コードブックのインデックスの入力ベクトル配列へのポインタ。
pSign	1 または -1 を含む各コードブックの符号の入力ベクトル配列へのポインタ。
pLength	出力ベクトルのパーティションの長さの配列へのポインタ。
nDiv	出力ベクトルのパーティションの数。
pCdbk	指定された IppsCdbkState_VQ_32f 構造体へのポインタの配列へのポインタ。

pDst	スペクトル値の再構築されたベクトルへのポインタ。
nCdbks	コードブックの数。

説明

この関数は、ippac.h ヘッダ・ファイルで宣言される。この関数は、いくつかのパーティションに区切られたベクトルを再構築する。パーティションは、指定された数のコードブックに含まれているベクトルのインデックスのセットを意味する。つまり、各コードブックの 1 つのパーティションにはインデックスが 1 つ含まれている。この関数は、指定された数のコードブックからベクトル値の算術平均を計算して、出力ベクトルの値を再構築する。配列 *pIndx* と *pSign* には、コードブックに含まれたベクトルの数と符号が格納される。

$$pDSt\left[\sum_{i=0}^{idiv-1} pLength[i]+j\right] = \frac{1}{nCdbk}\sum^{nCdbk-1} pSign[k][i]\cdot pCdbk[k] \rightarrow table[pIndx[k][i]][j]$$

但し、*i*=0 ～ *nDiv* -1 および *j* = 0 ～ *pLength*[i]-1。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pCdbk* または *pSrc* ポインタが NULL。

注： *pDst* 配列の長さは、次の値に等しくなければならない。

$$\sum_{k=0}^{nCdbk-1} pLengths[k]$$

圧伸関数

この項では、対数エンコーダとデコーダを使用してデータ圧縮処理（圧伸と呼ばれる）を行う関数について説明する。圧伸を使用すれば、量子化レベル [Rab78] を対数的に配分して、パーセンテージ・エラーを一定に保てる。

インテル® IPP 圧伸関数は、信号サンプルに対して次の変換操作を実行する。

- 8 ビット μ-law 形式でエンコードされたサンプルを PCM 線形サンプルに変換（またはその逆）。

- 8 ビット A-law 形式でエンコードされたサンプルを PCM 線形サンプルに変換(またはその逆)。
- 8 ビット μ-law 形式でエンコードされたサンプルを A-law 形式でエンコードされたサンプルに変換（またはその逆)。

μ-law 形式か A-law 形式でエンコードされたサンプルは、非均等に量子化する。これらの形式で使用する量子化関数は、S/N 比が、エンコードされた信号の大きさにあまり依存しないよう設計されている。これを実現するため、小さい信号レベルでは細かい解像度で量子化（圧伸）を行い、大きな信号レベルでは粗い解像度で量子化を行っている。出力値は、`[-1; +1]` の範囲で正規化される。

これらの関数は、CCITT G.711 仕様に準拠して、μ-law 圧伸や A-law 圧伸を実行する。変換の規則や詳しい説明については、[CCITT] を参照のこと。

例 10-7 は、圧伸関数の使用例を示している。

MuLawToLin

8 ビット μ-law 形式でエンコードされた
サンプルを線形サンプルにデコードする。

```
IppStatus ippsMuLawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsMuLawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

引数

pSrc	8 ビット μ-law でエンコードされたデコード対象の信号サンプルを格納するソース・ベクトルへのポインタ。
pDst	生成された線形サンプルを格納するデスティネーション・ベクトルへのポインタ。
len	ベクトル内のサンプルの数。

説明

関数 `ippsMuLawToLin` は、`ipps.h` ファイルで宣言される。この関数は、ベクトル *pSrc* 内にある 8 ビット μ-law でエンコードされたサンプルを PCM 線形サンプルにデコードし、それらをベクトル *pDst* に格納する。

μ-law による圧伸の式は、次のとおりである。

$$|c_{\mu}(x)| = \frac{\ln(1 + 255 \cdot |x|)}{\ln(256)} \cdot 128, \quad -1 \le x \le 1$$

上記の式で、x は線形信号サンプル、$C_{\mu}(x)$ は μ-law でエンコードされたサンプルを表す。

正の値と負の値は同じ方法で圧縮するため、式ではオリジナルの信号と圧縮した信号がいずれも絶対値で示される。入力の符号は、出力でも維持する。

アプリケーション・ノート

上に示した式をそのまま使用すると処理が遅くなるため、式を直接使用するのは避けなければならない。μ-law 形式のエンコードやデコードは、CCITT の仕様 G.711 に示されるルックアップ・テーブル 2a/G.711 および 2b/G.711 を使用して実行するのが一般的である。詳細は、G.711 仕様を参照のこと。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pDst* または *pSrc* が NULL。
`ippStsSizeErr`	エラー。*len* がゼロ以下。

LinToMuLaw

8 ビット μ-law 形式を使用して線形サンプルをエンコードし、それらをベクトルに格納する。

```
IppStatus ippsLinToMuLaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToMuLaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

引数

pSrc	エンコードする信号サンプル（1.0 未満に正規化）を格納するベクトルへのポインタ。
pDst	関数 `ippsLinToMuLaw` の出力を格納するベクトルへのポインタ。
len	ベクトル内のサンプルの数。

説明

関数 `ippsLinToMuLaw` は、`ipps.h` ファイルで宣言される。この関数は、ベクトル *pSrc* 内の PCM 線形サンプルを 8 ビット μ-law 形式でエンコードし、それらをベクトル *pDst* に格納する。

例 10-7 は、関数 `ippsLinToMuLaw_32f8u` の使用例を示している。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

ALawToLin

8 ビット A-law でエンコードされたサンプルを線形サンプルにデコードする。

```
IppStatus ippsALawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippsALawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

引数

pSrc	変換する信号サンプルを格納するベクトルへのポインタ。
pDst	関数 `ippsALawToLin` の出力を格納するベクトルへのポインタ。
len	ベクトル内のサンプルの数。

説明

関数 `ippsALawToLin` は、`ipps.h` ファイルで宣言される。この関数は、ベクトル *pSrc* 内にある 8 ビット A-law でエンコードされたサンプルを PCM 線形サンプルにデコードし、それらをベクトル *pDst* に格納する。

A-law による圧伸の式は、次のようになる。

$$|C_A(x)| = \begin{cases} \dfrac{87.56|x|}{1+\ln 87.56} \cdot 128, & 0 \le |x| \le \dfrac{1}{87.56} \\ \dfrac{1+\ln(87.56|x|)}{1+\ln 87.56} \cdot 128, & \dfrac{1}{87.56} < |x| \le 1 \end{cases}$$

上記の式で、*x* は線形信号サンプル、$C_A\ (x)$ は A-law でエンコードされたサンプルを表す。

正の値と負の値は同じ方法で圧縮するため、式ではオリジナルの信号と圧縮した信号がいずれも絶対値で示される。入力の符号は、出力でも維持する。

アプリケーション・ノート

前に示した式をそのまま使用すると処理が遅くなるため、式を直接使用するのは避けなければならない。A-law 形式のエンコードやデコードは、CCITT の仕様 G.711 に示されるルックアップ・テーブル 1a/G.711 および 1b/G.711 を使用して実行するのが一般的である。詳細は、G.711 仕様を参照のこと。

例 10-7 は、関数 `ippsALawToLin_8u32f` の使用例を示す。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *pDst* または *pSrc* が NULL。
`ippStsSizeErr`	エラー。*len* がゼロ以下。

LinToALaw

8 ビット A-law 形式を使用して線形サンプルをエンコードし、それら配列に格納する。

```
IppStatus ippsLinToALaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToALaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

引数

pSrc	エンコードする信号サンプルを格納するベクトルへのポインタ。

pDst	関数 ippsLinToALaw の出力を格納するベクトルへのポインタ。
len	ベクトル内のサンプルの数。

説明

関数 ippsLinToALaw は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* 内の PCM 線形サンプルを 8 ビット A-law 形式でエンコードし、それらをベクトル *pDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

MuLawToALaw

8 ビット μ-law 形式でエンコードされたサンプルを 8 ビット A-law 形式でエンコードされたサンプルに変換する。

```
IppStatus ippsMuLawToALaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

引数

pSrc	8 ビット μ-law でエンコードされた信号サンプルを格納するソース・ベクトルへのポインタ。
pDst	8 ビット A-law でエンコードされたサンプルを格納するデスティネーション・ベクトルへのポインタ。
len	ベクトル内のサンプルの数。

説明

関数 ippsMuLawToALaw は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* 内にある 8 ビットμ-law 形式でエンコードされた信号サンプルを、8ビット A-law 形式でエンコードされたサンプルに変換し、それらをベクトル *pDst* に格納する。

アプリケーション・ノート

µ-law 形式から A-law 形式への変換は、CCITT 仕様の G.711 に示されるルックアップ・テーブル 3/G.711 を使用して実行するのが一般的である。詳細は、G.711 仕様を参照のこと。

例 10-7 は、関数 ippsMuLawToALaw_8u の使用例を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

ALawToMuLaw

8 ビット A-law 形式でエンコードされたサンプルを、8 ビット µ-law 形式でエンコードされたサンプルに変換する。

```
IppStatus ippsALawToMuLaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

引数

pSrc	8 ビット A-law でエンコードされた信号サンプルを格納するソース・ベクトルへのポインタ。
pDst	8 ビット µ-law でエンコードされたサンプルを格納するデスティネーション・ベクトルへのポインタ。
len	ベクトル内のサンプルの数。

説明

関数 ippsMuLawToALaw は、ipps.h ファイルで宣言される。この関数は、ベクトル *pSrc* 内にある 8 ビット A-law 形式でエンコードされた信号サンプルを、8 ビット µ-law 形式で変換されたサンプルに変換し、それらをベクトル *pDst* に格納する。

アプリケーション・ノート

A-law 形式から μ-law 形式への変換は、CCITT 仕様の G.711 に示されるルックアップ・テーブル 4/G.711 を使用して実行するのが一般的である。詳細は、G.711 仕様を参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pDst* または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

例 10-7　圧伸関数の使用例

```
void compand( void ) {
     Ipp32f x[4] = { 0.1f, 0.2f, 0.3f, 0.4f };
     Ipp8u m[4], a[4];
     ippsLinToMuLaw_32f8u( x, m, 4 );
     ippsMuLawToALaw_8u( m, a, 4 );
     ippsALawToLin_8u32f( a, x, 4 );
     // now x must be close to original
     printf_32f("x =", x, 4, ippStsNoErr);
}
Output:
    x =  0.099609 0.207031 0.304688 0.398438
```

MP3 オーディオ符号化関数

「MP3」とも呼ばれている ISO/IEC 11172-3 MPEG-1、Layer III オーディオ符号化アルゴリズムは、ステレオおよびデュアルチャネルの音楽信号を圧縮することで幅広く利用されている。配信およびストレージ・アプリケーションに最適である MP3 アルゴリズムは、オリジナルの 1/10 のビット・レートで高品質のオーディオ再生を提供する。その結果、MP3 アルゴリズムは、新しい携帯型およびハンドヘルド・ストレージ・メディアや、インターネットを介した高品質圧縮オーディオの配信に最適な、オーディオ圧縮技術の事実上の標準となっている。MP3 エンコーダおよびデコーダは、音楽ストレージやオーディオ録音で幅広く利用されている。

マクロおよび定数

MP3 マクロおよび定数の定義を 表 10-2 に示す。

表 10-2　MP3 マクロおよび定数の定義

マクロのグローバル名	定義	注
IPP_MP3_GRANULE_LEN	576	1 つの最小単位内のサンプルの数。
IPP_MP3_V_BUF_LEN	512	V データ・バッファのサイズ（32 ビット・ワード）。
IPP_MP3_SF_BUF_LEN	40	スケール係数バッファのサイズ（8 ビット・ワード）。
IPP_MP3_SFB_TABLE_LONG_LEN	138	ロング・ブロック・サイズのスケール係数バンド・テーブル（16 ビット・ワード）。
IPP_MP3_SFB_TABLE_SHORT_LEN	84	ショート・ブロック・サイズのスケール係数バンド・テーブル（16 ビット・ワード）。

データ構造体

MP3 符号化 API には、いくつかのデータ構造体が含まれる。

IppMP3FrameHeader 構造体は、1 つのフレームに関連するすべてのヘッダ情報を格納する。

IppMP3SideInfo 構造体は、1 つのチャネルの 1 つの最小単位に関連するすべてのサイド情報を格納する。

構造体 IppMP3PsychoacousticModelTwoAnalysis には、ISO/IEC 11172-3 の心理音響モデル 2 のインテル® IPP で生成される出力が格納される。これには、現在のフレームに関連するマスクされたしきい値や知覚エントロピの推定などが含まれている。マスクされたしきい値は、マスク対シグナル・レート（MSR）で表現される。

構造体 IppMP3PsychoacousticModelTwoState には、干渉するブロック処理を容易にするため、ISO/IEC 11172-3 のインテル IPP の実装に関連するステート情報が格納される。

構造体 IppMP3BitReservoir には、量子化ビット貯蓄に関連するステート情報が格納される。

フレーム・ヘッダ

```
typedef struct {
    int id;             /* ID 1: MPEG-1, 0: MPEG-2 */
    int layer;          /* layer index 0x3: Layer I
```

```
                    //              0x2: Layer II
                    //              0x1: Layer III */
    int protectionBit; /* CRC flag 0: CRC on, 1: CRC off */
    int bitRate;       /* bit rate index */
    int samplingFreq;  /* sampling frequency index */
    int paddingBit;    /* padding flag 0: no padding, 1 padding */
    int privateBit;    /* private_bit, not used */
    int mode;          /* mono/stereo selection */
    int modeExt;       /* extension to mode */
    int copyright;     /* copyright or not, 0: no, 1: yes */
    int originalCopy;  /* original or copied, 0: copy, 1: original*/
    int emphasis;      /* flag indicating the type of de-emphasis */
    int CRCWord;       /* CRC-check word */

} IppMP3FrameHeader;
```

サイド情報

```
typedef struct {
    int  part23Len;     /* number of main_data bits */
    int  bigVals;       /* half the number of Huffman code words
                           whose maximum amplitudes may be
                           greater than 1 */
    int  globGain;      /* quantizes step size information */
    int  sfCompress;    /* number of bits used for scale factors */
    int  winSwitch;     /* window switch flag */
    int  blockType;     /* block type flag */
    int  mixedBlock;    /* flag 0: non mixed block, 1: mixed block */
    int pTableSelect[3];  /* Huffman table index for the 3
                             rectangle in <big_values> field */
    int  pSubBlkGain[3]; /* gain offset from the global gain
                            for one subblock */
    int  reg0Cnt;       /* the number of scale factor bands in
                           the first region of <big_values>
                           less one */
    int  reg1Cnt;       /* the number of scale factor bands in
                           the second region of <big_values>
                           less one */
    int  preFlag;       /* flag indicating high frequency boost */
    int  sfScale;       /* scale factor scaling */
```

```
    int  cntlTabSel;   /* Huffman table index for the <count1>
                          field of quadruples */
 } IppMP3SideInfo;
```

MP3 心理音響モデル 2 分析

```
typedef struct{
Ipp32s pMSR[36];/* MSRs for one granule/channel.
        For long blocks, elements 0-20 represent the thresholds
        associated with the 21 SFBs.For short blocks, elements
        0,3,6,..,33,
        elements 1,4,...,34, and elements 2,5,...,35,
        respectively,
        represent the thresholds associated with the 12 SFBs
        for each
        of the 3 consecutive short blocks in one
        granule/channel.That
        is, the block thresholds are interleaved such that the
        thresholds are grouped by SFB.*/
Ipp32s PE;      /* Estimated perceptual entropy, one
                   granule/channel */
} IppMP3PsychoacousticModelTwoAnalysis;
```

心理音響モデル 2 ステート

```
typedef struct {
    Ipp64s pPrevMaskedThesholdLong[2][63]; /* long block masked
        threshold
        history buffer; Contains masked threshold estimates for the
        threshold
        calculation partitions associated with the two most
        recent long
        blocks */

    Ipp64s pPrevMaskedThesholdShort[42]; /* short block masked
        threshold
        history buffer; Contains masked threshold estimates for the
        threshold
        calculation partitions associated with the most recent short
        block */

    Ipp32sc pPrevFFT[2][6];/* FFT history buffer; Contains real and
        imaginary FFT components associated with the two most recent
        long blocks */
```

```
    Ipp32s  pPrevFFTMag[2][6];/* FFT magnitude history buffer;
        contains FFT component magnitudes associated with the two most
        recent long blocks */

    int nextPerceptualEntropy; /* PE estimate for next granule; one
        granule delay provided for synchronization with analysis
        filterbank */

    int nextBlockType; /* Expected block type for next granule; either
        long (normal), short, or stop.Depending upon analysis results
        for the granule following the next, a long block could change
        to a start block, and a stop block could change to a short
        block.This buffer provides one granule of delay for
        synchronization with the analysis filterbank */

    Ipp32s pNextMSRLong[21];   /* long block MSR estimates for next
        granule.
        One granule delay provided for synchronization with analysis
        filterbank */

    Ipp32s pNextMSRShort[36];  /* short block MSR estimates for next
        granule.One granule delay provided for synchronization with
        analysis filterbank */
} IppMP3PsychoacousticModelTwoState;
```

MP3 ビット貯蓄

```
typedef struct {
    int BitsRemaining; /* bits currently remaining in the
                          reservoir */
    int MaxBits;       /* maximum possible reservoir size, in bits,
        determined as follows:  min(7680-avg_frame_len, 2^9*8),
        where: avg_frame_len is the average frame length (in bits),
        including padding bits and excluding side information bits
} IppMP3BitReservoir;
```

MP3 コーデック列挙型

インテル® IPP MP3 エンコーダおよびデコーダ API は、エンコーダ・コンポーネントとデコーダ・コンポーネント間における同期化とデータ転送を容易にする列挙データ型を定義する。表 10-3 に示すように、MP3 コーデック API は、定数を使用した周波数のセマンティク解釈を提供するいくつかの列挙型で構成されている。

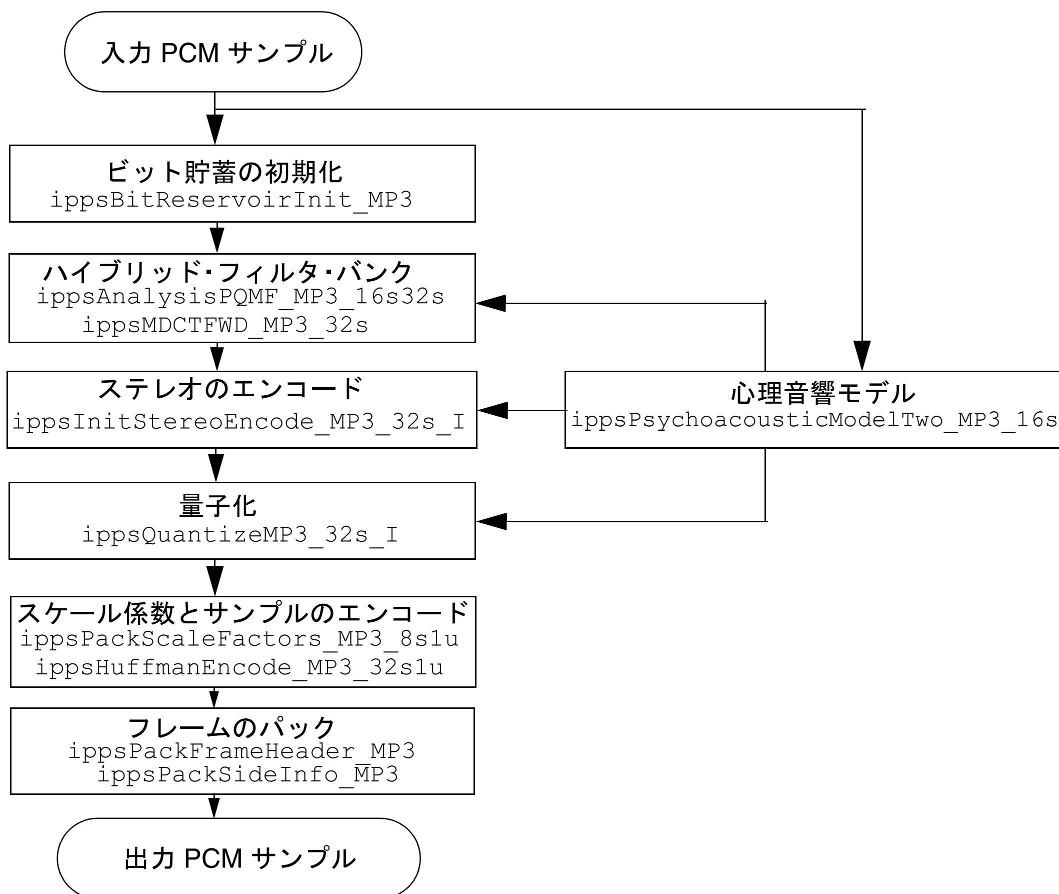
表 10-3　MP3 列挙データ型

列挙型の名前	シンボル値	定数値
IppMP3BitRate	ippMP3BitRateFree	0
	ippMP3BitRate32	1
	ippMP3BitRate40	2
	ippMP3BitRate48	3
	ippMP3BitRate56	4
	ippMP3BitRate64	5
	ippMP3BitRate80	6
	ippMP3BitRate96	7
	ippMP3BitRate112	8
	ippMP3BitRate128	9
	ippMP3BitRate160	10
	ippMP3BitRate192	11
	ippMP3BitRate224	12
	ippMP3BitRate256	13
	ippMP3BitRate320	14
IppMP3SampleRate	ippMP3SampleRate32000	2
	ippMP3SampleRate44100	0
	ippMP3SampleRate48000	1
IppMP3PcmMode	ippMP3NonInterleavedPCM	1
	ippMP3InterleavedPCM	2
IppMP3Emphasis	IppMP3EmphasisNone	0
	IppMP3Emphasis5015	1
	IppMP3EmphasisReserved	2
	IppMP3CCITTJ17	3

MP3 オーディオ・エンコーダ

MP3 エンコーダ・アプリケーション・プログラミング・インターフェイス（API）は、ビットストリーム・パック関数と MP3 コア・エンコード関数を含む、さまざまな機能を提供する（[ISO-11172] を参照）。本節は、インテル® インテグレーテッド・パフォーマンス・プリミティブ（インテル® IPP）における MP3 オーディオ・エンコーダ API のリファレンス・ガイドである。図 10-2 に示すように、この API は、関数に加えて、事前に定義されたマクロと定数で構成されている。

図 10-2　インテル® IPP MP3 エンコーダ API のフローチャート



AnalysisPQMF_MP3

MP3 ハイブリッド分析フィルタ・バンクの
第 1 ステージを実行する。

```
IppStatus ippsAnalysisPQMF_MP3_16s32s (const Ipp16s *pSrcPcm, Ipp32s
     *pDstS, int pcmMode);
```

引数

pSrcPcm	入力 PCM オーディオ・ベクトルを含むバッファの始点へのポインタ。サンプルは次の仕様を満たしていること。

- 16 ビット、符号付き、リトル・エンディアン、Q15 形式。
- 最新の 480（512-32）サンプルはベクトル
  pSrcPcm[*pcmMode**i] に含まれる。
  ここで、*i* = 0,1,..,479 である。
- 現在のグラニュールに関連するサンプルはベクトル
  pSrcPcm[*pcmMode**j] に含まれる。
  ここで、*j* = 480,481,..,1055 である。

pcmMode PCM モード・フラグ。PQMF フィルタ・バンクに入力 PCM ベクトル構成のタイプを伝える。

- *pcmMode* = 1 は、非インターリーブ形式の PCM 入力サンプルを示す。
- *pcmMode* = 2 は、インターリーブ形式の PCM 入力サンプルを示す。

pDstS 576 の要素ブロックを含む PQMF 分析出力ベクトルの始点へのポインタ。次のインデックスに従って、32 サブバンド・サンプルの連続した 18 ブロックが含まれる。
pDstXs[32**i* + *sb*]。ここで、*i* = 0、1、…、17 は時系列インデックスである。
sb = 0、1、…、31 はサブバンド・インデックスである。

説明

この関数は、ipps.h ファイルで宣言される。この関数は、MP3 ハイブリッド分析フィルタ・バンクの第 1 ステージを実行する。この関数は、512 の サンプル・プロトタイプ・ウィンドウを持つクリティカルにサンプリングされたブロック PQMF 分析バンクを、PCM 入力オーディオ・ベクトルに適用する。

各チャネルの各グラニュールごとに 18 回（つまり、各フレームの各チャネルごとに 36 回）、ippsAnalysisPQMF_MP3_16s32s 関数を呼び出す。

注： すべての係数は、Q7.24 形式で表される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcPcm* または *pDstXs* が NULL。
ippStsErr	エラー。*pcmMode* が [1,2] を超えている。

MDCTFwd_MP3

MP3 ハイブリッド分析フィルタ・バンクの
第 2 ステージを実行する。

```
IppStatus ippsMDCTFwd_MP3_32s (const Ipp32s *pSrc, Ipp32s *pDst, int
    blockType, int mixedBlock, IppMP3FrameHeader *pFrameHeader, Ipp32s
    *pOverlapBuf);
```

引数

pSrc — 576 の要素ブロックを含む PQMF 分析出力ベクトルの始点へのポインタ。次のインデックスに従って、32 サブバンド・サンプルの連続した 18 ブロックが含まれる。
pDstS[32**i* + *sb*]。ここで、*i* = 0、1、…、17 は時系列インデックスである。
sb = 0、1、…、31 はサブバンド・インデックスである。
すべての係数は、Q7.24 形式で表される。

blockType — ブロック・タイプのインジケータ。

- 0 - ノーマル・ブロック
- 1- スタート・ブロック
- 2 - ショート・ブロック
- 3 - ストップ・ブロック

mixedBlock — 複合ブロックのインジケータ。

- 0 - 非複合
- 1 - 複合

pFrameHeader — 現在のフレームに関連するヘッダを含む IppMP3FrameHeader 構造体へのポインタ。現在のところ、MPEG-1（*id* = 1）だけが使用できる。

pOverlapBuf — PQMF バンク出力の最新 576 の要素ブロックのコピーを含む MDCT オーバーラップ・バッファへのポインタ。分析フィルタ・バンクで新しいオーディオ・ストリームを処理する前に、バッファ *pOverlapBuf* のすべての要素を定数値 0 で初期化すること。

pDst — 分析フィルタ・バンクによって生成された 576 の要素のスペクトル係数出力ベクトルへのポインタ

説明

この関数は、ipps.h ファイルで宣言される。この関数は、次の操作を行って、MP3 ハイブリッド分析フィルタ・バンクの第 2 ステージを実行する。

1. **順方向 MDCT。**第 1 ステージの分析中、適切に配列された 12 ポイント、36 ポイントの順方向変形離散コサイン変換（MDCT）のセットが、32 の PQMF サブバンドでそれぞれ生成された 18 の サンプル・スペクトル係数ブロックに適用される。

2. **エリアシング削減バタフライ。**2 つのクリティカルにサンプリングされた分析フィルタ・バンクを段階的に行うことで発生するエリアシングの影響を緩和するため、ISO/IEC 11172-3 で指定されているバタフライが MDCT 出力に適用される。これらはそれぞれ、無視できない量のインターバンド・エリアシングを発生させる。

関数 ippsMDCTFwd_MP3_32s は、576 の要素の MDCT オーバーラップ・バッファ *pMDCTOverlap*[] を更新する。このバッファの内容は、ブロック処理を容易にするため、呼び出し間で保持する必要がある。この関数は、各チャネルの各グラニュールごとに 1 回（つまり、各フレームの各チャネルごとに 2 回）適用しなければならない。

注： 入力係数は、Q7.24 形式で表される。出力係数は、Q5.26 形式で表される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcXs*、*pDstXr*、*pFrameHeader*、または *pOverlapBuf* の 1 つが NULL。

PsychoacousticModelTwo_MP3

ISO/IEC 11172-3 の心理音響モデル 2 を実装して、PCM オーディオ入力のブロックに関連するマスクされたしきい値と知覚エントロピを推定する。

```
IppStatus ippsPsychoacousticModelTwo_MP3_16s (const Ipp16s *pSrcPcm,
    IppMP3PsychoacousticModelTwoAnalysis *pDstPsyInfo, int
    *pDstIsSfbBound, IppMP3SideInfo *pDstSideInfo, IppMP3FrameHeader
    *pFrameHeader, IppMP3PsychoacousticModelTwoState *pFramePsyState,
    Ipp32s *pWorkBuffer, int pcmMode);
```

引数

pSrcPcm 入力 PCM オーディオ・ベクトルを含むバッファの始点へのポインタ。サンプルは次の仕様を満たしていること。

- サンプルごとに 16 ビット、符号付き、リトル・エンディアン、Q15 形式。
- *pSrcPcm* バッファには、パラメータ *pFrameHeader* -> *mode* の値が 1（モノ）の場合、1152 サンプル（576 サンプルごとに 2 グラニュール）が含まれる。パラメータ *pFrameHeader* -> *mode* の値が 2 （ステレオ、デュアル・モノ）の場合、2304 サンプル（2 チャネルの 576 サンプルごとに 2 グラニュール）が含まれる。
- ステレオの場合、左と右チャネルに関連する PCM サンプルは *pcmMode* フラグに従って構成される。上記の PCM フォーマットやバッファ要件のすべてが満たされていない場合、未定義のモデルが出力される。

pFrameHeader 現在のフレームと関連するヘッダを含む `IppMP3FrameHeader` 構造体へのポインタ。構造体 **pFrameHeader* の `samplingFreq`、`id`、および `mode` フィールドは、心理音響モデルの動作を制御する。3 つのフィールドはすべて、この関数を呼び出す前に正しく初期化されていなければならない。他のすべてのフレーム・ヘッダ・フィールドは無視される。現在のところ、MPEG-1（*id* = 1）だけが使用できる。

pFramePsyState 直前のフレームと現在のフレームに関連する心理音響モデルのステート情報を含む、1 セットの `IppMP3PsychoacousticModelTwoState` 構造体の最初の要

素へのポインタ。セット中の要素の数は、入力オーディオに含まれているチャネルの数に等しい（つまり、各チャネルで個々の分析が行われる）。

pcmMode　PCM モード・フラグ。PCM ベクトル構成の心理音響モデルのタイプを伝える。

- *pcmMode* = 1 は、非インターリーブ形式の PCM 入力サンプルを示す。つまり、*pSrcPcm*[0..1151] は左チャネルに関連する入力サンプルを含み、*pSrcPcm*[1152..2303] は右チャネルに関連する入力サンプルを含む。
- *pcmMode* = 2 は、インターリーブ形式の PCM 入力サンプルを示す。つまり、*pSrcPcm*[2**i*] と *pSrcPcm*[2**i*+1] は左と右チャネルに関連する入力サンプルをそれぞれ含む。ここで、*i* = 0,1,…,1151 である。

列挙型 IppMP3PcmMode の型キャスト要素 ippMP3NonInterleavedPCM と ippMP3InterleavedPCM を、*pcmMode* の定数 1 と 2 の代わりに使用することもできる。

pWorkBuffer　中間結果とその他の一時データ用に、心理音響モデルによって内部的に使用されるワークスペース・バッファへのポインタ。バッファの長さは少なくとも 25,200 バイト（6300 の Ipp32s 型要素）でなければならない。

pDstPsychoInfo　1 セットの PsychoacousticModelTwoAnalysis 構造体の最初の要素へのポインタ。各セットのメンバは、1 つのグラニュール用の MSR と PE 推定を含む。セット中の要素の数は、次のように配列された出力のチャネルの数に等しい。

```
(Analysis[0] = granule 1, channel 1),
(…Analysis[1] = granule 1, channel 2),
(…Analysis[2] = granule 2, channel 1),
(…Analysis[3] = granule 2, channel 2)
```

pDstIsSfbBound　インテンシティ符号化が有効になると、*pDstIsSfbBound* は、これより上ではすべてのスペクトル係数がジョイント・ステレオ・インテンシティ符号化モジュールで処理される SFB 下限のリストを指す。インテンシティ符号化の SFB 下限はブロック固有なので、*pDstIsSfbBound* によって指される有効な要素の数は、各グラニュールに関連する個々のブロック・タイプに依存して異なる。特に、SFB 境界のリストは、次のようにインデックスが付けられる。

- *pIsSfbBound*[3**gr*]（ロング・ブロック・グラニュールの場合）

- *pIsSfbBound*[3**gr* + *w*]（ショート・ブロック・グラニュールの場合）

ここで、*gr* はグラニュール・インデックス（0 はグラニュール 1、1 はグラニュール 2 を示す）、*w* はブロック・インデックス（0 はブロック 1、1 はブロック 2、2 はブロック 3 を示す）である。

例えば、グラニュール 1 のショート・ブロック分析に続いてグラニュール 2 のロング・ブロック分析が指定された場合、SFB 境界のリストは次の順で生成される。

pIsSfbBound[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}.

グラニュールがロング・ブロック分析用に構成された場合、1 つの SFB 下限が決定される。ショート・ブロック分析用に構成された場合、3 つの SFB 下限が決定される。MS 符号化とインテンシティ符号化の両方が有効な場合、インテンシティ符号化の SFB 下限は同時に MS 符号化の SFB 上限を表す。MS 符号化のみが有効な場合、SFB 境界は最も低い非 MS SFB を表す。

pDstSideInfo すべてのグラニュールとチャネルに関連する IppMP3SideInfo 構造体の更新されたセットへのポインタ。モデルは、すべてのセット要素中のフィールド *blockType*、*winSwitch*、および *mixedBlock* を更新する。セット中の要素の数は、チャネルの数の 2 倍に等しい。セットの要素の順序は、*pDstPsychoInfo* と同じである。

pFramePsyState 現在のフレームと次のフレームに関連する更新された心理音響モデルのステート情報を含む、1 セットの IppMP3PsychoacousticModelTwoState 構造体の最初の要素へのポインタ。セット中の要素の数は、入力オーディオに含まれているチャネルの数と等しい（つまり、各チャネルで個々の分析が行われる）。

新しいオーディオ・ストリームをエンコードする前に、心理音響モデルのステート構造体 *pPsychoacousticModelState* のすべての要素を値 0 で初期化すること。

信号処理ドメインでは、次のように関数 ippsZero_16s を使用して実行できる。

```
ippsZero_16s ((Ipp16s *)
pPsychoacousticModelState,sizeof(IppMP3Psychoaco
usticModelTwoState)/sizeof(Ipp16s))
```

pFrameHeader 現在のフレームに関連するヘッダを含む、更新された IppMP3FrameHeader 構造体へのポインタ。モデルは、要素 *modeExt* を更新してジョイント・ステレオ符号化コードの決定を反映する。他のフレーム・ヘッダ・フィールドは、この関数では修正されない。

説明

この関数は、ipps.h ファイルで宣言される。この関数は、ISO/IEC 11172-3 の心理音響モデル 2 を実装して、PCM オーディオ入力のブロックに関連するマスクされたしきい値と知覚エントロピを推定する。量子化プロセスは、モデル出力を使用して、分析フィルタ・バンクによって生成されたスペクトル係数の知覚的に最適なビット割り当てを推定する。心理音響モデルは、分析フィルタ・バンクのブロック・サイズの切り替えに加えて、ステレオ MS/ インテンシティ・モードの選択と処理も制御する。チャネルごとに 1152 サンプル（各 576 サンプルの 2 グラニュール）の PCM 入力オーディオの 1 つのフレームから、心理音響モデルは次の出力を生成する。

1. **推定 SFB（スケール係数バンド）MSR（マスク対シグナル・レート）。**モデルはロング・ブロック・モード中の 21 SFB とショート・ブロック・モード中の各 3 連続ブロックの 12 SFB について推定 MSR のベクトルを生成する。

   MSR はマスクされたしきい値から得られ、入力オーディオの 1 つのグラニュール / チャネルに関連する、同時マスキングのレベルを算出する。リスナーに伝えられるオーディオ・スティミュラスの特性により、このしきい値は基本的に、グラニュールで瞬時に修正された最小可聴限界の値を算出する。理想的には、しきい値を推定する際は平均的なリスナーが量子化雑音、つまり他のスペクトルのエネルギーを知覚できない、周波数に依存した音圧レベル（dB、SPL）プロファイルを提供する。

   入力オーディオのブロックからマスクされたしきい値を推定するため、関数 ippsPsych_MP3_16s は ISO/IEC 11172-3 の Annex D.2 で推奨されている手順を実装している。

   最初に、標準的な FFT ベースのスペクトル分析の出力は、サブクリティカルな帯域幅の解像度で分析が行われるように構成された、しきい値計算パーティションにグループ化される。各しきい値計算パーティションで、モデルは、スペクトルの予測不能な時間を評価して決定される、トーン状またはノイズ状の信号の性質を重み付けした推定を使用して、各パーティションのマスキング・レベルを推定する。

   次に、聴覚システムのスペクトルの選択性をモデル化するため、スプレッド関数が適用される。

最後に、推定しきい値が最小可聴限界と比較され、最大 2 つまでしきい値計算パーティションに割り当てられる。最終的に、量子化モジュールのビット割り当て方式にその出力を一致させるため、モデルは、しきい値計算パーティション・スケールからスケール係数バンド（SFB）スケールに変換される。ロング・ブロック（576 サンプル）には 1 セットの 21 SFB しきい値が生成され、ショート・ブロック（192 サンプル）には 3 つの連続した 12 SFB しきい値のブロックが生成される。

効率的な量子化を行うため、SFB しきい値は、信号のエネルギーによって反転されて正規化され、SFB MSR（マスク対信号レート）のベクトルで返される。推定 MSR は、`PsychoacousticModelTwoAnalysis` 構造体に返される。

2. **推定知覚エントロピ。**モデルは、各グラニュールの知覚エントロピ（PE）推定を生成する。PE は、「知覚透過」で（つまり、オリジナルの符号化されていないバージョンと比較して平均的なリスナーが聞き取ることのできる音質が劣化しないように）、グラニュールの PCM サンプルを表すために必要な最小ビット数を算出する。

   推定 PE は、各 SFB で特定の SNR（信号対ノイズ・レート、SN 比）の目標を達成するために必要な最小ビット数に関する従来の仮定（1 ビット増加すると SNR が 6db 向上する）を組み合わせて、マスクされたしきい値から得られる。必要な最小 SNR と 各 SFB で必要なビット数は、SMR（信号対マスク・レート）から得られる。

   急激に PE が大きく増加すると、プリエコー歪みを発生しやすい一時的なオーディオ・イベントがしばしば発生する。このため、知覚エントロピは分析フィルタ・バンクのブロック・サイズの切り替えに使用される。PE 推定は、`PsychoacousticModelTwoAnalysis` 構造体に返される。

3. **分析フィルタ・バンクのブロック・サイズの決定。**知覚エントロピと他のインジケータを使用して、モデルは現在のグラニュールがプリエコー歪みに弱いかどうかを決定する。プリエコーが発生しそうな場合はショート・ブロックモードを、その他の場合はロング・ブロック・モードを使用する。

   適切なブロック・タイプが選択されるように、この決定は切り替えロジックの前のブロックに組み込まれる。例えば、現在のブロック・タイプがロングで次のブロック・タイプがショートの場合、モード切り替えでシームレスなブロック処理が行われるように、現在のブロック・タイプはブロック・タイプ「ロング / ノーマル」から「ロング / スタート」に変更される。同様に、現在のブロック・タイプが「ロング / ストップ」で次のブロック・タイプが

「ショート」の場合、ブロック切り替えロジックは、不要なモード切り替えが発生しないように、現在のブロック・タイプを「ロング / ストップ」から「ショート」に変更する。ブロック・タイプの決定は、IppMP3SideInfo 構造体のフレーム / グラニュール・フィールドに返される。

4. **ジョイント・ステレオ処理モードの決定。**オーディオ・ソースが 2 チャネルの場合、モデルは、チャネル間の相関関係と他のインジケータを評価して、ジョイント・ステレオ LR/MS やインテンシティ処理モードを決定する。ジョイント・ステレオ・モードの決定は、IppMP3FrameHeader 構造体の modeExt フィールドに返される。
5. **インテンシティ・ステレオ符号化の SFB 境界の決定。**インテンシティ符号化が有効になると（この前の「ジョイント・ステレオ処理モードの決定」を参照）、心理音響モデルは、これより上ではすべてのスペクトル係数が、インテンシティ・モードのステレオ処理を使用してエンコードされる適切な SFB 下限を決定する。

心理音響モデルは、ステレオまたはデュアル・モノ入力用の 2 つのグラニュールと 2 つまでのチャネルを含む、フレーム・ベース（チャネルごとに 1152 サンプル）の分析を実行する。入力ベクトルと出力ベクトルの有効な長さは、有効なチャネル・モード（モノまたはステレオ）に依存する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcPcm*、*pDstPsyInfo*、*pDstSideInfo*、*pDstIsSfbBound*、*pFrameHeader*、*pDstPsyState*、または *pWorkBuffer* のうち少なくとも 1 つが NULL。

JointStereoEncode_MP3

独立した左と右チャネルのスペクトル係数ベクトルを、量子化に適した mid/side（MS）やインテンシティ（IS）モード係数ベクトルの組み合わせに変換する。

```
IppStatus ippsJointStereoEncode_MP3_32s_I (Ipp32s *pSrcDstXrL, Ipp32s
    *pSrcDstXrR, Ipp8s *pDstScaleFactorR, IppMP3FrameHeader
    *pFrameHeader, IppMP3SideInfo *pSideInfo, int *pIsSfbBound);
```

引数

pSrcDstXrL	入力オーディオの左チャネルの分析フィルタ・バンクによって生成された 576 の要素のスペクトル係数出力ベクトルへのポインタ。すべての係数は、Q5.26 形式で表される。
pSrcDstXrR	入力オーディオの右チャネルの分析フィルタ・バンクによって生成された 576 の要素のスペクトル係数出力ベクトルへのポインタ。すべての係数は、Q5.26 形式で表される。
pFrameHeader	現在のフレームに関連するヘッダ情報を含む `IppMP3FrameHeader` 構造体へのポインタ。関数入口で、構造体フィールド `samplingFreq`、`id`、`mode`、および `modeExt` はそれぞれ、現在の入力オーディオに関連するサンプル・レート、アルゴリズム *id*（MPEG-1 または MPEG-2）、および心理音響モデルによって生成されたジョイント・ステレオ符号化コマンドを含んでいる必要がある。他のすべての **pFrameHeader* フィールドは無視される。 現在のところ、MPEG-1（`id = 1`）だけが使用できる。
pSideInfo	ジョイントしてエンコードされるチャネルのペアに関連する `IppMP3SideInfo` 構造体のペアへのポインタ。セット中の要素の数は 2 で、セットの要素の順序は次のとおりである。*pSideInfo*`[0]` はチャネル 1 を記述し、*pSideInfo*`[1]` はチャネル 2 を記述する。関数入口で、両方のチャネルの `blockType` サイド情報フィールドは、各チャネルの心理音響モデルによって選択された分析モード（ショート・ブロックまたはロング・ブロック）を反映する必要がある。*pSideInfo*`[0]` と *pSideInfo*`[1]` 構造体の他のすべてのフィールドは無視される。
pIsSfbBound	これより上では、すべての L/R チャネルのスペクトル係数がインテンシティ符号化で表現されるように組み合わされる、現在のグラニュールの両方のチャネルに対するインテンシティ符号化の SFB 下限リストへのポインタ。要素の数は、現在のグラニュールに関連するブロック・タイプに依存する。ショート・ブロックの場合、SFB 境界は次の順序で表される。*pIsSfbBound*`[0]` はブロック 1 を記述し、*pIsSfbBound*`[1]` はブロック 2 を記述し、*pIsSfbBound*`[2]` はブロック 3 を記述する。

	ロング・ブロックの場合、1 つの SFB 下限の決定のみ必要で、*pIsSfbBound*[0] で表される。MS 符号化とインテンシティ符号化の両方が有効な場合、インテンシティ符号化の SFB 下限は同時に MS 符号化の SFB 上限を表す。MS 符号化のみが有効な場合、SFB 境界は最も低い非 MS SFB を表す。
pSrcDstXrL	インテンシティ SFB 下限より上のインテンシティ符号化係数と同様に、M チャネルに関連する 576 の要素のジョイント・ステレオ・スペクトル係数出力ベクトルへのポインタ。すべての係数は、Q5.26 形式で表される。
pSrcDstXrR	S チャネルに関連する 576 の要素のジョイント・ステレオ・スペクトル係数出力ベクトルへのポインタ。すべての係数は、Q5.26 形式で表される。
pDstScaleFactorR	右 /S チャネルの 1 つのグラニュールと関連するスケール係数のベクトルへのポインタ。インテンシティ符号化が特定の SFB 下限より上の心理音響モデルによって有効にされている場合（pIsSfbBound によって指されているフレーム・ヘッダとベクトルで示される）、関数 StereoEncode_MP3_32s_I は適切なスケール係数を更新する。これは、インテンシティ符号化スケール係数バンドと関連する *pDstScaleFactorR*[] の要素である。スケール係数ベクトル中の他の SFB エントリは修正されない。*pDstScaleFactorR* によって参照されるベクトルの長さは、ブロック・サイズの関数として変化する。ベクトルは、ロング・ブロック・グラニュールの場合は 21 要素、ショート・ブロック・グラニュールの場合は 36 要素を含む。

説明

この関数は、ipps.h ファイルで宣言される。この関数は、独立した左と右チャネルのスペクトル係数ベクトルを量子化に適した mid/side（MS）、インテンシティ（IS）モード係数ベクトルの組み合わせに変換する。MS 符号化が有効な場合（*pFrameHeader* -> *modeExt* & 0x10 == 1）、左と右チャネルは次のように Mid と Side チャネルに変換される。

$$M = \frac{L+R}{\sqrt{2}} \quad S = \frac{L-R}{\sqrt{2}}$$

この関数は、デュアル・グラニュール方式で呼び出される。この関数は、グラニュール /2 チャネルごとに呼び出す必要がある。

インテンシティ符号化が有効な場合（*pFrameHeader -> modeExt* & 0x01 = 1）、左チャネルは SFB インテンシティ下限より上の SFB のインテンシティ・データを処理し、SFB 下限より上の右チャネルはクリアされる。すべての係数は 0 に設定される。

$L = L + R, R = 0$

デコーダで左と右のスペクトル係数のエネルギーに比例した回復を容易にするため、インテンシティ・エネルギー・スケール係数 *is_pos* は、右チャネルのスケール係数の代わりに送信される。SFB 境界の上にある右チャネルのスペクトル係数は除去される。エネルギー正規化定数は L/R SFB エネルギー・レートから得られた後、その量子化特性を改善するために変形される。これを式で表すと次のようになる。

$$is_pos = n\,\mathrm{int}\left(\frac{12}{\pi}\arctan\left(\sqrt{\frac{L_energy}{R_energy}}\right)\right)$$

ここで、*L_energy* と *R_energy* はそれぞれ、左と右チャネルのスペクトル係数に関連する SFB エネルギーである。デコーダで、*is_pos* スケール係数は、ジョイント符号化の前の信号エネルギーのディストリビューションと同じ方法で、左と右チャネル間の符号化エネルギーをジョイント符号化するために使用される。*is_pos* インテンシティ・スケール係数はベクトル *pDstScalefactorR* に返される。

各グラニュールで、ジョイント・ステレオ符号化はチャネルのペアごとに 1 回適用される（各チャネルの各グラニュールごとに 576 サンプル）。したがって、関数 JointStereoEncode_MP3_32s_I は、各グラニュールごとに 1 回、または各フレームごとに 2 回呼び出される必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDstXrL*、*pSrcDstXrR*、*pDstScaleFactorR*、*pFrameHeader*、*pSideInfo*、または *pIsSfbBound* のうち少なくとも 1 つが NULL。
ippStsMP3SideInfoErr	エラー。IS または MS が使用された場合に pSideInfo[0].blockType!= pSideInfo[1].blockType。

Quantize_MP3

分析フィルタ・バンクによって生成された
スペクトル係数を量子化する。

```
IppStatus ippsQuantize_MP3_32s_I (Ipp32s *pSrcDstXrIx, Ipp8s
    *pDstScalefactor, int?*pDstScfsi, int *pDstCount1Len, int
    *pDstHufSize, IppMP3FrameHeader *pFrameHeader, IppMP3SideInfo
    *pSideInfo, IppMP3PsychoacousticModelTwoAnalysis *pPsychoInfo,
    IppMP3PsychoacousticModelTwoState *pFramePsyState,
    IppMP3BitReservoir *pResv, int meanBits, int *pIsSfbBound, Ipp32s
    *pWorkBuffer);
```

引数

pSrcDstXrIx 分析フィルタ・バンクによって生成され、1 つのフレーム用のジョイント・ステレオ符号化モジュールにより処理された、1 セットの量子化されていないスペクトル係数ベクトルへのポインタ。量子化されていない係数のセットは、次のようにインデックスが付けられる。

- *pSrcDstXrIx*[*gr**1152 + *ch**576 + *i*]（ステレオまたはデュアル・モノ入力ソースの場合）
- *pSrcDstXrIx*[*gr**576 + *i*]（モノラル入力ソースの場合）

ここで、

- *i* = 0、1、…、575 はスペクトル係数インデックスである
- *gr* はグラニュール・インデックスである。0 はグラニュール 1、1 はグラニュール 2 を示す。
- *ch* はチャネル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。

適用されたジョイント符号化のタイプに応じて、各チャネルの係数は入力オーディオの L/R、M/S やインテンシティ表現に関連する。すべての係数は、Q5.26 形式で表される。

pFrameHeader 現在のフレームに関連するヘッダ情報を含む `IppMP3FrameHeader` 構造体へのポインタ。関数入口で、構造体フィールド `samplingFreq`、`id`、`mode`、および `modeExt` はそれぞれ、現在の入力オーディオに関連するサンプル・レート、アルゴリズム `id`（MPEG-1 または MPEG-2）、および心理音響モデルによって生成されたジョイント・ステレオ符号化コマンド

を含んでいる必要がある。他のすべての *pFrameHeader フィールドは無視される。現在のところ、MPEG-1（id = 1）だけが使用できる。

pSideInfo すべてのグラニュールとチャネルに関連する IppMP3SideInfo 構造体のセットへのポインタ。セットは、2*nchan、要素を含み、pSideInfo[gr*nchan + ch] のようにインデックスが付けられている必要がある。ここで、

- gr はグラニュール・インデックスである。0 はグラニュール 1、1 はグラニュール 2 を示す。
- nchan はチャネルの数である。
- ch はチャ 1 ル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。

関数入口で、すべてのセット要素中の構造体フィールド blockType、mixedBlock、および winSwitch はそれぞれ、ブロック・タイプ・インジケータ（start、short、または stop）、フィルタ・バンク複合ブロック分析モード規制子、および現在の入力オーディオに関連するウィンドウ切り替えフラグ（normal または blockType）を含んでいる必要がある。次の出力引数で記述されているように、他のすべての *pSideInfo フィールドは関数入口で無視され、関数出口で更新される。

pPsychoInfo 現在のフレームに関連する 1 セットの PsychoacousticModelTwoAnalysis 構造体の最初の要素へのポインタ。各セットのメンバは、1 つのグラニュールの 1 つのチャネルの MSR と PE 推定を含む。セットは、2*nchan、要素を含み、pPsychoaInfo[gr*nchan+ ch] のようにインデックスが付けられている必要がある。ここで、

- gr はグラニュール・インデックスである。0 はグラニュール 1、1 はグラニュール 2 を示す。
- nchan はチャネル数である。
- ch はチャネル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。

pFramePsyState 現在のフレームと次のフレームに関連する心理音響モデルのステート情報を含む 1 セットの IppMP3PsychoacousticModelTwoState 構造体の最初の要素へのポインタ。セット中の要素の数は、入力オーディオに含まれているチャネルの数に等しい（つまり、各チャネルで個々

の分析が行われる)。量子化器は、フレーム・タイプ先取り情報 *nextBlockType* を使用してビット貯蓄を管理する。他のすべての構造体要素は量子化器によって無視される。

pResv ビット貯蓄ステート情報を含む IppMP3BitReservoir 構造体へのポインタ。関数入口で、すべての構造体フィールドは有効なデータを含んでいる必要がある。

meanBits フレーム・ヘッダで指定されたターゲット・ビット・レート(キロバイト / 秒)を使用して、スペクトル係数とスケール係数の各フレームに平均的な方法で割り当てられたビットの数。この数には、フレーム・ヘッダとサイド情報に割り当てられたビットは含まれていない。量子化器は、現在のフレームのターゲット割り当てとして *meanBits* を使用する。このターゲットよりも大きな知覚ビット割り当てが要求されると、量子化器はフレームの即座の要求を満たすために、ビット貯蓄で保持されている余りビットを使用する。同様に、このターゲットよりも小さい知覚ビット割り当てが要求されると、量子化器は将来のフレームで使用するために、ビット貯蓄に余りビットを格納する。

pIsSfbBound これより上では、すべての L/R チャネルのスペクトル係数がインテンシティ符号化で表現されるように組み合わされる、SFB 下限のリストへのポインタ。pIsSfbBound によって指される有効な要素の数は、現在のフレームのグラニュールに関連するブロック・タイプに依存する。

特に、pIsSfbBound によって指される SFB 境界のリストは、*pIsSfbBound*[3**gr*](ロング・ブロック・グラニュールの場合)、および *pIsSfbBound*[3**gr* + *w*](ショート・ブロック・グラニュールの場合)のようにインデックスが付けられる。ここで、

- *gr* はグラニュール・インデックスである。0 はグラニュール 1、1 はグラニュール 2 を示す。
- *w* はブロック・インデックスである。0 はブロック 1、1 はブロック 2、2 はブロック 3 を示す。

例えば、グラニュール 1 のショート・ブロック分析に続いてグラニュール 2 のロング・ブロック分析が指定された場合、SFB 境界のリストは次の順で生成される。

pIsSfbBound[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}

グラニュールがロング・ブロック分析用に構成された場合、1 つの SFB 下限が決定される。ショート・ブロック分析用に構成され

た場合、3 つの SFB 下限が決定される。MS 符号化とインテンシティ符号化の両方が有効な場合、インテンシティ符号化の SFB 下限は同時に MS 符号化の SFB 上限を表す。MS 符号化のみが有効な場合、SFB 境界は最も低い非 MS SFB を表す。

pWorkBuffer 中間結果とその他の一時データ用に、量子化器によって内部的に使用されるワークスペース・バッファへのポインタ。バッファの長さは少なくとも 2880 バイト（各 32 ビットの 720 ワード）でなければならない。

pSrcDstXrIx 量子化されたスペクトル係数ベクトルの出力セットへのポインタ。これらはハフマン・エンコーダへの入力に適している。係数は、pSrcDstXrIx[*gr**1152 + *ch**576 + *i*]（ステレオまたはデュアル・モノ入力ソースの場合）、および pSrcDstXrIx[*gr**576 + *i*]（単一チャネル入力ソースの場合）のようにインデックスが付けられる。ここで、

- *i* = 0、1、…、575 はスペクトル係数インデックスである
- *gr* はグラニュール・インデックスである。0 はグラニュール 1、1 はグラニュール 2 を示す。
- *ch* はチャネル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。

pDstScaleFactor 量子化プロセス中に生成されたスケール係数の出力セットへのポインタ。これらのスケール係数は、量子化器のグラニュールを決定する。スケール係数ベクトルの長さは、各グラニュールに関連するブロック・モードに依存する。要素の順序は次のようになる。

1. （グラニュール 1、チャネル 1）
2. （グラニュール 1、チャネル 2）
3. （グラニュール 2、チャネル 1）
4. （グラニュール 2、チャネル 2）

この一般的な構成では、ベクトル *pDstScfsi* に含まれているフラグに関連する各グラニュール / チャネルのサイド情報を、正確なスケール係数ベクトルのインデックスと長さを決定するために使用できる。

pDstScfsi スケール係数選択情報の出力ベクトルへのポインタ。このベクトルは、スケール係数が定義済みのスケール係数選択グループ内でフレームのグラニュールで共有されるかどうかを示す、バイナリ・フラグのセットを含む。例えば、ISO/IEC 11172-3 で定義され

intel®

ているように、バンド 0、1、2、3、4、5 は第 1 グループを形成し、バンド 6、7、8、9、10 は第 2 グループを形成する。ベクトルは、*pDstScfsi*[*ch*][*scfsi_band*] のようにインデックスが付けられる。ここで、

- *ch* はチャネル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。
- *scfsi_band* は、スケール係数選択グループの番号である。グループ 0 は SFB 0-5、グループ 1 は SFB 6-10、グループ 2 は SFB 11-15、およびグループ 3 は SFB 16-20 をそれぞれ含む。

pDstCount1Len *count1* 領域長さ規定子の出力ベクトルへのポインタ。*count1* パラメータは、*bigvals* 領域より高い周波数のハフマン符号化スペクトル係数の、-1、0、+1 のいずれかの値が 4 つずつ得られる領域のサイズを示す。ベクトルは、2**nchan*、要素を含み、*pDstCount1Len*[*gr***nchan* + *ch*] のようにインデックスが付けられる。ここで、

- *gr* はグラニュール・インデックスである。0 はグラニュール 1、1 はグラニュール 2 を示す。
- *nchan* はチャネルの数である。
- *ch* はチャネル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。

pDstHufSize ハフマン符号化ビット割り当て規定子の出力ベクトルへのポインタ。各グラニュール / チャネルについて、規定子は、*bigvals* と *count1* 領域で量子化されたスペクトル係数を表すために必要なハフマン・ビットの総数を示す。

必要な場合は常に、各 *HufSize* ビット・カウントはビット貯蓄の管理に必要なビットの数を含めるために増やされる。貯蓄が最大量に達したフレームでは、量子化器は追加ビットでスペクトル・サンプルのハフマン表現をパディングして、余りビットを消費する。これらのパディング要求は、量子化器によって返される *HufSize* の結果に反映される。つまり、*HufSize*[i] は、ハフマン符号に必要なビットの数とパディング・ビットの数の合計と等しい。ベクトルは、2**nchan*、要素を含み、*pDstHufSize*[gr**nchan*+*ch*] のようにインデックスが付けられる。ここで、

- *gr* はグラニュール・インデックスである。0 はグラニュール 1、1 はグラニュール 2 を示す。
- *nchan* はチャネルの数である。

- *ch* はチャネル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。

pSideInfo 更新された IppMP3SideInfo サイド情報構造体のセットへのポインタ。すべてのセット要素で量子化器は、*part23Len*、*bigVals*、*globGain*、*sfCompress*、*pTableSelect*[0]-[2]、*pSubBlkGain*[0]-[2]、*reg0Cnt*、*reg1Cnt*、*sfScale*、*preFlag*、および *cnt1TabSel* の構造体フィールドを変更する。

セットは、2**nchan*、要素を含み、*pSideInfo*[*gr***nchan*+*ch*] のようにインデックスが付けられる。ここで、

- *gr* はグラニュール・インデックスである。0 はグラニュール 1、1 はグラニュール 2 を示す。
- *nchan* はチャネルの数である。
- *ch* はチャネル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。

pResv 更新された IppMP3BitReservoir 構造体へのポインタ。量子化器は、*BitsRemaining* フィールドを更新して必要なビットを追加または削除する。他のすべてのフィールドは量子化器によって変更されない。

説明

この関数は、ipps.h ファイルで宣言される。この関数は、分析フィルタ・バンクによって生成されたスペクトル係数を量子化する。発生する歪み（つまり、量子化ノイズ）は、心理音響モデルによって推定された、マスクされたしきい値から得られるプロファイルと一致するように形成される。

知覚の歪み基準を満たすことに加えて、量子化器は同時に全体的なビット割り当てを調節して、固定ビット・レート目標を達成する。ISO/IEC 11172-3 の推奨に従って、ビット貯蓄は平均方式で固定レートの制約条件に違反することなく、瞬間的なピーク・レート要求を満たすように維持される。平均の知覚ビット・レート要求よりも低いフレームでは、余りビットがビット貯蓄に預けられる。平均の知覚ビット・レート要求よりも高いフレームでは、補足ビットがビット貯蓄から引き出される。

量子化器は、一定の平均レート制約条件を満たすようにビット貯蓄と全体的なビット割り当てを管理する。量子化器は、データの完全なフレーム（つまり、2 つのグラニュールと 1 チャネルまたは 2 チャネル）を処理する。したがって、フレームごとに 1 回呼び出される必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcDstXrIx*、*pDstScaleFactor*、*pDstScfsi*、*pDstCount1Len*、*pDstHufSize*、*pFrameHeader*、*pSideInfo*、*pPsychInfo*、*pFramePsyState*、*pResv*、**pIsSfbBound**、または *pWorkBuffer* が NULL。
ippStsMP3SideInfo	エラー。*pSideInfo->winSwitch* と *pSideInfo->mixedBlock* の両方が定義されている。

PackScalefactors_MP3

スケール係数にノイズレスの符号化を適用して、ビットストリーム・バッファに出力をパックする。

```
IppStatus ippsPackScalefactors_MP3_8s1u (const Ipp8s *pSrcScalefactor,
    Ipp8u **ppBitStream, int *pOffset, IppMP3FrameHeader *pFrameHeader,
    IppMP3SideInfo *pSideInfo, int *pScfsi, int granule, int channel);
```

引数

pSrcScaleFactor 1 つのグラニュールの 1 つのチャネルの量子化プロセス中に生成されたスケール係数のベクトルへのポインタ。スケール係数ベクトルの長さは、ブロック・モードに依存する。ショート・ブロック・グラニュールの場合、スケール係数ベクトルは 36 の要素、または各サブブロックで 12 の要素を含む。ロング・ブロック・グラニュールの場合、スケール係数ベクトルは 21 の要素を含む。

ショート・ブロックのスケール係数ベクトルは、*pSrcScaleFactor*[*sb**12+*sfb*] のようにインデックスが付けられる。ここで、

- *sb* はサブブロック・インデックスである。0 はサブブロック 1、1 はサブブロック 2、2 はサブブロック 3 を示す。
- *sfb* はスケール係数バンド・インデックス（0-11）である。

ロング・ブロックのスケール係数ベクトルは、`pSrcScaleFactor[sfb]` のようにインデックスが付けられる。ここで、*sfb* はスケール係数バンド・インデックス（0-20）である。

個々のグラニュール / チャネルの関連するサイド情報は、適切なインデックス方式を選択するために使用できる。

ppBitStream　エンコードされたビットストリーム・バッファへのポインタ。*ppBitStream* パラメータは、関数 EncodeScaleFactors_MP3_8s1u によって生成されたハフマン符号化されたスケール係数ビットを受信する、ビットストリーム・バッファの最初のバイトへの二重ポインタである。スケール係数ハフマン・ビットは、バイト・ポインタ **ppBitStream* とビット・ポインタ *pOffset* の組み合わせによってインデックスを付けられたビットから開始するストリーム・バッファに連続して書き込まれる。

pOffset　ビットストリームのビット・ポインタ。**ppBitStream* によって参照されるバイト内の次の利用可能なビットにインデックスを付ける。*pOffset* パラメータは、**ppBitStream* によって参照されるバイト内の次の利用可能なビットにインデックスを付ける。このパラメータは、0 ～ 7 の範囲で有効である。ここで、0 は最上位ビット、7 は最下位ビットに対応する。

pFrameHeader　このフレームの `IppMP3FrameHeader` 構造体へのポインタ。関数入口で、構造体フィールド *id* および *modeExt* はそれぞれ、アルゴリズム *id*（MPEG-1 または MPEG-2）および心理音響モデルによって生成されたジョイント・ステレオ符号化コマンドを含んでいる必要がある。他のすべての **pFrameHeader* フィールドは無視される。現在のところ、MPEG-1（*id* = 1）だけが使用できる。

pSideInfo　現在のグラニュールとチャネルの `IppMP3SideInfo` 構造体へのポインタ。関数入口で、構造体フィールド *blockType*、*mixedBlock*、および *sfCompress* はそれぞれ、ブロック・タイプ・インジケータ（*start*、*short*、または *stop*）、フィルタ・バンク複合ブロック分析モード規制子、およびスケール係数ビット割り当てを含んでいる必要がある。他のすべての **pSideInfo* フィールドはスケール係数エンコーダによって無視される。

pScfsi　スケール係数が定義済みのスケール係数選択グループ内でフレームのグラニュールで共有されるかどうかを示す、バイナリ・フラグのセットを含むスケール係数選択情報テーブルへのポインタ。

例えば、ISO/IEC 11172-3 で定義されているように、バンド 0、1、2、3、4、5 は第 1 グループを形成し、バンド 6、7、8、9、10 は第 2 グループを形成する。ベクトルは、*pScfsi*[*ch*][*scfsi_band*] のようにインデックスが付けられる。ここで、

- *ch* はチャネル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。
- *scfsi_band* は、スケール係数選択グループの番号である。グループ 0 は SFB 0-5、グループ 1 は SFB 6-10、グループ 2 は SFB 11-15、およびグループ 3 は SFB 16-20 をそれぞれ含む。

granule 現在のグラニュールのインデックス。0 はグラニュール 1、1 はグラニュール 2 を示す。

channel 現在のチャネルのインデックス。0 はチャネル 1、1 はチャネル 2 を示す。

ppBitStream 更新されたビットストリームのバイト・ポインタ。このパラメータは、スケール係数ハフマン・エンコーダによって生成され、ストリーム・バッファに連続して書き込まれたビットに続く、最初の利用可能なビットストリーム・バッファのバイトを指す。スケール係数ビットは、ISO/IEC 11172-3 で指定されているビットストリーム書式に従ってフォーマットされる。

pOffset 更新されたビットストリームのビット・ポインタ。*pOffset* パラメータは、更新されたビットストリーム・バッファのバイト・ポインタ **ppBitStream* によって参照される次の利用可能なバイト内の、次の利用可能なビットにインデックスを付ける。このパラメータは、0 ～ 7 の範囲で有効である。ここで、0 は最上位ビット、7 は最下位ビットに対応する。

説明

この関数は、ipps.h ファイルで宣言される。この関数は、スケール係数にノイズレスの符号化を適用して、ビットストリーム・バッファに出力をパックする。この関数は、一度に 1 つのグラニュールの 1 つのチャネルを操作する。したがって、各グラニュールの各チャネルについて 1 回、この関数を呼び出す必要がある。

結果のビットストリームは、ISO/IEC 11172-3 で指定されている書式に完全に準拠している。

戻り値

ippStsNoErr エラーなし。

`ippStsNullPtrErr`	エラー。*pSrcScaleFactor*、*ppBitStream*、*ppBitStream*、*pOffset*、*pFrameHeader*、*pSideInfo*、または *pScfsi* が NULL。
`ippStsMP3SideInfoErr`	エラー。*pFrameHeader->id* == IPP_MP3_ID_MPEG1 および *pSideInfo->sfCompress* が [0..15] を超えている。または *pFrameHeader->id* == IPP_MP3_ID_MPEG2 および *pSideInfo->sfCompress* が [0..511] を超えている。
`ippStsMP3FrameHeaderErr`	エラー。*pFrameHeader->id* == IPP_MP3_ID_MPEG2 および *pFrameHeader->modeExt* が [0..3] を超えている。

HuffmanEncode_MP3

量子化サンプルに無損失のハフマン符号化を適用して、ビットストリーム・バッファに出力をパックする。

```
IppStatus ippsHuffmanEncode_MP3_32s1u (Ipp32s *pSrcIx, Ipp8u
    **ppBitStream, int *pOffset, IppMP3FrameHeader *pFrameHeader,
    IppMP3SideInfo *pSideInfo, int count1Len, int hufSize);
```

引数

pSrcIx グラニュールの量子化サンプルへのポインタ。バッファの長さは 576 である。適用されたジョイント符号化のタイプに応じて、係数ベクトルは量子化されたスペクトル・データの L、R、M、S やインテンシティ・チャネルに関連する。

ppBitStream ビットストリームのバイト・ポインタ。*ppBitStream* パラメータは、この関数によって生成された、ハフマン符号化されたスペクトル係数ビットを受信する、ビットストリーム・バッファの最初のバイトへの二重ポインタである。ハフマン符号化されたスペクトル係数ビットは、ストリーム・バッファに連続して書き込ま

れる。このストリーム・バッファは、バイト・ポインタ *ppBitStream とビット・ポインタ pOffset の組み合わせによってインデックスを付けられたビットから開始する。

pOffset ビットストリームのビット・ポインタ。pOffset パラメータは、*ppBitStream によって参照されるバイト内の次の利用可能なビットにインデックスを付ける。このパラメータは、0 ～ 7 の範囲で有効である。ここで、0 は最上位ビット、7 は最下位ビットに対応する。

pFrameHeader このフレームの IppMP3FrameHeader 構造体へのポインタ。ハフマン・エンコーダは、サイド情報（下記参照）と関連するフレーム・ヘッダの id フィールドを使用して、Big Value スペクトル領域のハフマン・テーブル領域境界を計算する。ハフマン・エンコーダは、他のすべてのフレーム・ヘッダ・フィールドを無視する。現在のところ、MPEG-1（*id = 1*）だけが使用できる。

pSideInfo 現在のグラニュールとチャネルの IppMP3SideInfo 構造体へのポインタ。構造体の要素 bigVals、pTableSelect[0]-[2]、reg0Cnt、および reg1Cnt は、Big Value 領域でスペクトル係数の符号化を制御するために使用される。構造体の要素 cnt1TabSel は、-1、0、+1 のいずれかの値が 4 つずつ得られる *count1* 領域で適切なハフマン・テーブルを選択するために使用される。

すべてのサイド情報の要素に関する詳細は、構造体の定義ヘッダ・ファイルを参照のこと。

count1Len count1 領域の長さ規定子。-1、0、+1 のいずれかの値が 4 つずつ得られる、Big Value 領域より上の、現在のグラニュール / チャネルのスペクトル・サンプルの数を示す。

hufSize ハフマン符号化ビット割り当て規定子。*bigvals* と *count1* 領域の両方で、現在のグラニュール / チャネルのハフマン符号化された、量子化されたスペクトル係数を表すために必要なビットの総数を示す。

必要な場合は常に、このビット・カウントはビット貯蓄の管理に必要なビットの数を含めるために増やされる。貯蓄が最大量に達したフレームでは、追加ビットでスペクトル・サンプルのハフマン表現がパディングされ、余りビットが消費される。

これらのパディング要求は、関数 Quantize_MP3_32s_I によって返される *HufSize* の結果に反映される。つまり、*HufSize[i]* は、ハフマン符号に必要なビットの数とパディング・ビットの数の合計と等しい。

ppBitStream	更新されたビットストリームのバイト・ポインタ。パラメータ **ppBitStream* は、スペクトル係数ハフマン・エンコーダによって生成され、ストリーム・バッファに連続して書き込まれたビットに続く、最初の利用可能なビットストリーム・バッファのバイトを指す。ハフマン符号ビットは、ISO/IEC 11172-3 で指定されているビットストリーム書式に従ってフォーマットされる。
pOffset	更新されたビットストリームのビット・ポインタ。*pOffset* パラメータは、更新されたビットストリーム・バッファのバイト・ポインタ **ppBitStream* によって参照される次の利用可能なバイト内の、次の利用可能なビットにインデックスを付ける。このパラメータは、0 ～ 7 の範囲で有効である。ここで、0 は最上位ビット、7 は最下位ビットに対応する。

説明

この関数は、`ipps.h` ファイルで宣言される。この関数は、量子化サンプルに無損失のハフマン符号化を適用して、ビットストリーム・バッファに出力をパックする。

この関数は、一度に 1 つのグラニュールをエンコードするため、各チャネルの各グラニュールごとに 1 回呼び出される必要がある。

結果のビットストリームは、ISO/IEC 11172-3 で指定されている書式に完全に準拠している。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。*ppBitStream*、*pOffset*、*pSrcIx*、*pSideInfo*、*ppBitStream*、または *pFrameHeader* が NULL。
`ippStsBadArgErr`	エラー。*pOffset* が `[0,7]` を超えている。
`ippStsMP3SideInfoErr`	エラー。*pSideInfo*`->bigVals*2>IPP_MP3_GRANULE_LEN (pSideInfo->reg0Cnt + pSideInfo->reg1Cnt + 2) >= 23`、*pSideInfo*`->`*cnt1TabSel* が `[0,1]` を超えているか、または *pSideInfo*`->`*pTableSelect*`[i]` が `[0..31]` を超えている。

intel®

ippStsMP3FrameHeader	エラー。*FrameHeader*->id != 1、*pFrameHeader*->layer != 1、または *pFrameHeader*->samplingFreq が [0..2] を超えている。

PackFrameHeader_MP3

フレーム・ヘッダの内容をビットストリームにパックする。

```
IppStatus ippsPackFrameHeader_MP3 (IppMP3FrameHeader *pSrcFrameHeader,
    Ipp8u **ppBitStream);
```

引数

pSrcFrameHeader	IppMP3FrameHeader 構造体へのポインタ。この構造体は、現在のフレームに関連するヘッダ情報をすべて含む。すべての構造体フィールドは有効なデータを含んでいる必要がある。
ppBitStream	エンコードされたビットストリーム・バッファへのポインタ。*ppBitStream* パラメータは、この関数によって生成された、パックされたフレーム・ヘッダ・ビットを受信する、ビットストリーム・バッファの最初のバイトへの二重ポインタである。フレーム・ヘッダ・ビットは、バイト・ポインタ **ppBitStream* の組み合わせによってインデックスを付けられたビットから開始するストリーム・バッファに連続して書き込まれる。
ppBitStream	更新されたビットストリームのバイト・ポインタ。パラメータ **ppBitStream* は、パックされたフレーム・ヘッダ・ビットに続く、最初の利用可能なビットストリーム・バッファのバイトを指す。フレーム・ヘッダ・ビットは、ISO/IEC 11172-3 で指定されているビットストリーム書式に従ってフォーマットされる。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、フレーム・ヘッダの内容をビットストリームにパックする。

結果のビットストリームは、ISO 11172-3 で指定されている書式に完全に準拠している。

この関数は、各フレームごとに 1 回呼び出される必要がある。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。*pSrcFrameHeader*、*ppBitStream*、または *ppBitStream* が NULL。

PackSideInfo_MP3

サイド情報をビットストリーム・バッファにパックする。

```
IppStatus ippsPackSideInfo_MP3 (IppMP3SideInfo *pSrcSideInfo, Ipp8u
    **ppBitStream, int mainDataBegin, int privateBits, int *pSrcScfsi,
    IppMP3FrameHeader *pFrameHeader);
```

引数

pSrcSideInfo — IppMP3SideInfo 構造体へのポインタ。この構造体はチャネル数の 2 倍の要素を含む。要素の順序は次のようになる。

- (グラニュール 1、チャネル 1)
- (グラニュール 1、チャネル 2)
- (グラニュール 2、チャネル 1)
- (グラニュール 2、チャネル 2)

すべてのセット要素の全フィールドは、関数入口で有効なデータを含んでいる必要がある。

mainDataBegin — 負のビットストリーム・オフセット (バイト単位)。このパラメータ値は、通常は、現在のフレームの量子化が開始する前にビット貯蓄に残されているバイトの数である。*mainDataBegin* を計算する場合、ヘッダとサイド情報のバイト数を除外する必要がある。

サイド情報フォーマッタは、*mainDataBegin* の 9 ビットの値を出力ビットストリームの main_data_begin フィールドにパックする。

privateBits — チャネルの数に応じて、関数はパラメータ *privateBits* から適切な数の最下位ビットを抽出し、出力ビットストリームの private_bits フィールドにパックする。ISO/IEC 11172-3 の

ビットストリーム書式は、パラメータ main_data_begin に続くレイヤ III のビットストリーム・オーディオ・データ・セクションに含まれるアプリケーション特有の（プライベート）ビット数を、チャネルに応じて保存する。ISO/IEC 11172-3:1993 を参照のこと。デュアル・チャネル・ストリームとシングル・チャネル・ストリームでは、それぞれ、3 ビットと 5 ビットが保存される。

pSrcScfsi スケール係数選択情報テーブルへのポインタ。このベクトルは、スケール係数が定義済みのスケール係数選択グループ内でフレームのグラニュールで共有されるかどうかを示す、バイナリ・フラグのセットを含む。例えば、ISO/IEC 11172-3 [2] で定義されているように、バンド 0、1、2、3、4、5 は第 1 グループを形成し、バンド 6、7、8、9、10 は第 2 グループを形成する。

ベクトルは、*pDstScfsi*[*ch*][*scfsi_band*] のようにインデックスが付けられる。ここで、

- *ch* はチャネル・インデックスである。0 はチャネル 1、1 はチャネル 2 を示す。
- *scfsi_band* は、スケール係数選択グループの番号である。グループ 0 は SFB 0-5、グループ 1 は SFB 6-10、グループ 2 は SFB 11-15、およびグループ 3 は SFB 16-20 をそれぞれ含む。

pFrameHeader IppMP3FrameHeader 構造体へのポインタ。現在のところ、MPEG-1（*id* = 1）だけが使用できる。関数入口で、構造体フィールド id、mode、および layer はそれぞれ、アルゴリズム id（MPEG-1 または MPEG-2）、モノまたはステレオ・モード、および MPEG レイヤ規定子を含んでいる必要がある。

他のすべての **pFrameHeader* フィールドは無視される。

ppBitStream エンコードされたビットストリーム・バッファへのポインタ。このパラメータは、この関数によって生成された、パックされたサイド情報を受信する、ビットストリーム・バッファの最初のバイトへの二重ポインタである。サイド情報ビットは、**ppBitStream* によって参照されるバイトでアライメントされた場所から開始するストリーム・バッファに連続して書き込まれる。

ppBitStream 更新されたビットストリームのバイト・ポインタ。パラメータ **ppBitStream* は、パックされたサイド情報ビットに続く、最初の利用可能なビットストリーム・バッファのバイトを指す。サイド情報ビットは、ISO/IEC 11172-3: 1993 で指定されているビットストリーム書式に従ってフォーマットされる。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、サイド情報をビットストリーム・バッファにパックする。

結果のビットストリームは、ISO 11172-3 で指定されている書式に完全に準拠している。

この関数は、各フレームごとに 1 回呼び出される必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcSideInfo*、*ppBitStream*、*ppBitStream*、*pSrcScfsi*、または *pFrameHeader* が NULL。
ippStsMP3FrameHeaderErr	エラー。*pFrameHeader->id* が [0,1] を超えているか、*pFrameHeader->layer* != 1 、または *pFrameHeader->mode* が [0..3] を超えている。

BitReservoirInit_MP3

ビット貯蓄ステート構造体のすべての要素を初期化する。

```
IppStatus ippsBitReservoirInit_MP3 (IppMP3BitReservoir *pDstBitResv,
    IppMP3FrameHeader *pFrameHeader);
```

引数

pFrameHeader 現在のフレームに関連するヘッダ情報を含む IppMP3FrameHeader 構造体へのポインタ。関数入口で、フレーム・ヘッダ・フィールド bitRate と id はそれぞれ、有効なビット・レート・インデックスとアルゴリズム id（MPEG-1 または MPEG-2）を含んでいる必要がある。

他のすべてのフレーム・ヘッダ・パラメータは無視される。現在のところ、MPEG-1（*id* = 1）だけが使用できる。

pDstBitResv	初期化された IppMP3BitReservoir ステート構造体へのポインタ。構造体の要素 BitsRemaining は、0 に初期化される。構造体の要素 MaxBits は、指定されたフレームの始点で貯蓄できるビットの最大数を反映するように初期化される。MaxBits の適切な値は、選択されたアルゴリズム（MPEG-1 または MPEG-2）とレート・インデックス・パラメータ *pFrameHeader*.bitRate によって示されたストリーム・ビット・レートによって直接決定される。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、フレーム・ヘッダで指定された符号化アルゴリズム（MPEG-1 または MPEG-2）とフレーム・ビット・割り当てごとの平均に基づいて、ビット貯蓄ステート構造体のすべての要素を初期化する。

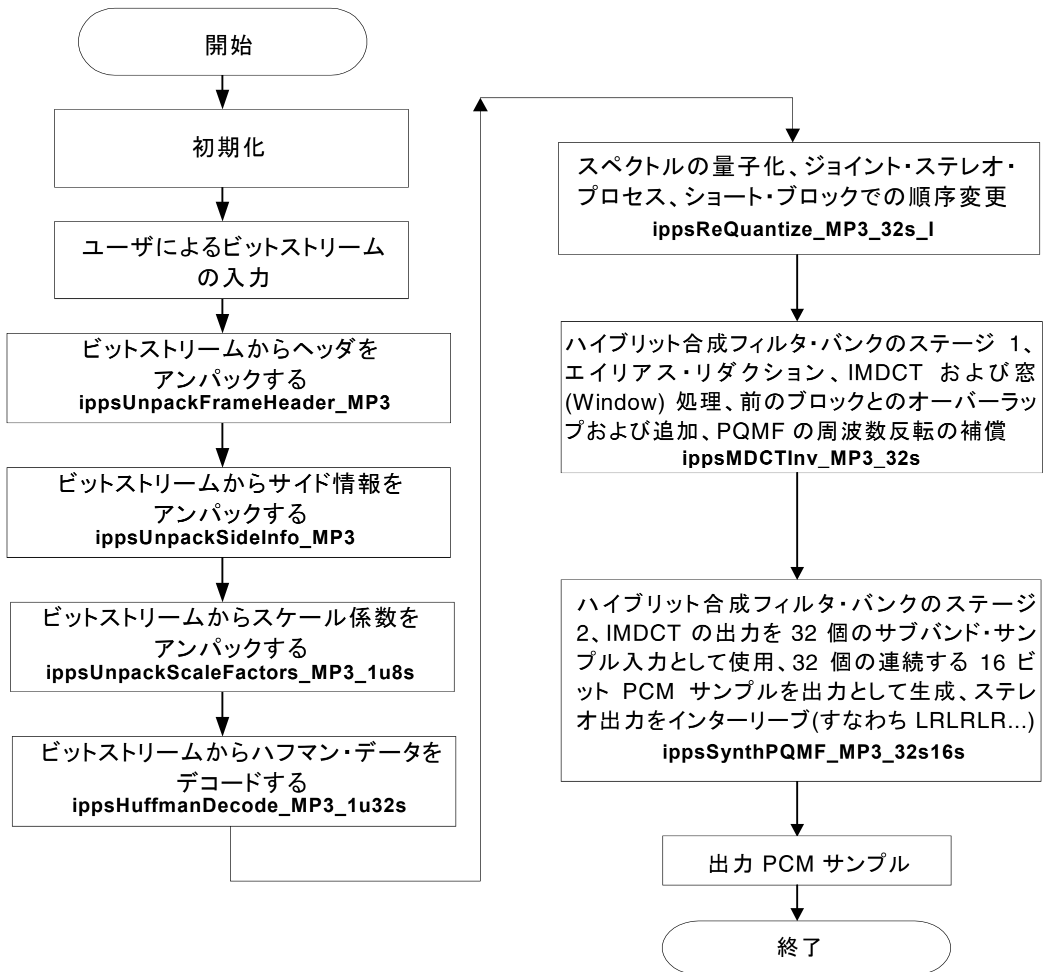
戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pDstBitResv* または *pFrameHeader* が NULL。
ippStsMP3FrameHeaderErr	エラー。*pFrameHeader->id* != _IPP_MP3_ID_MPEG1。

MP3 オーディオ・デコーダ

この項では、ISO/IEC 13818-3 MPEG-2 オーディオ使用（[ISO] を参照）のレイヤ III の部分に準拠したオーディオ・デコーダを作成するインテル® IPP 関数について説明する。また、インテル IPP における MP3 アプリケーション・プログラミング・インターフェイス（API）のリファレンス・ガイドも紹介する。図 10-3 に示すように、MP3 API は、7 つの関数、2 つのデータ構造体、事前に定義された定数とマクロで構成されている。

図 10-3　MP3 デコーダ API



UnpackFrameHeader_MP3

オーディオ・フレーム・ヘッダを
アンパックする。

```
IppStatus ippsUnpackFrameHeader_MP3(Ipp8u** ppBitStream,
    IppMP3FrameHeader* pFrameHeader);
```

引数

ppBitStream	MP3 フレーム・ヘッダの第 1 バイトへのポインタへのポインタ (**ppBitstream* は関数内で更新される)。
pFrameHeader	MP3 フレーム・ヘッダ構造体へのポインタ。

説明

関数 `ippsUnpackFrameHeader_MP3` は、`ippac.h` ファイルで宣言される。この関数は、オーディオ・フレーム・ヘッダをアンパックする。巡回冗長検査 (CRC) がイネーブルになっている場合、この関数は CRC ワードもアンパックする。

`ippsUnpackFrameHeader_MP3` を呼び出す前に、ユーザはビットストリーム同期化ワードを探し、**ppBitStream* が 32 ビット・フレーム・ヘッダの第 1 バイトを指していることを確認する必要がある。

CRC がイネーブルになっている場合、MP3 規格の規定に従って、16 ビット CRC ワードは 32 ビット・フレーム・ヘッダに隣接すると見なされる。呼び出し元に戻る前に、この関数は、フレーム・ヘッダまたは CRC ワードの次のバイトを参照するように、ポインタ **ppBitStream* を更新する。

16 ビット CRC ワードの第 1 バイトは、*pFrameHeader*`->CRCWord(15:8)` に格納され、第 2 バイトは、*pFrameHeader*`->CRCWord(7:0)` に格納される。

この関数は、破壊されたフレーム・ヘッダを検出しない。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。*ppBitStreamp*、*FrameHeader*、または *ppBitStream* が NULL。

UnpackSideInfo_MP3

入力ビットストリームからサイド情報を
アンパックする。この情報は、関連する
フレームのデコードに使用される。

```
IppStatus ippsUnpackSideInfo_MP3(Ipp8u** ppBitStream,
    IppMP3SideInfo* pDstSideInfo, int* pDstMainDataBegin,
    int* pDstPrivateBits, int* pDstScfsi,  IppMP3FrameHeader*
    pFrameHeader);
```

引数

ppBitStream	ビットストリーム・バッファ内の現在のフレームに関連するサイド情報の第 1 バイトへのポインタへのポインタ。この関数は、このパラメータを更新する。
pFrameHeader	アンパックされた MP3 フレーム・ヘッダを格納する構造体へのポインタ。このヘッダ構造体は、入力ビットストリームのフォーマットに関する情報を提供する。MPEG-1 および MPEG-2 のシングルチャネルおよびデュアルチャネル・モードをサポートしている
pDstSideInfo	MP3 サイド情報構造体へのポインタ。この構造体は、現在のフレームのすべての最小単位およびすべてのチャネルに適用されるサイド情報を格納する。1 つ以上の構造体が、次の順序で、*pDstSideInfo* によって指定されるバッファ内に連続して配置される。 `{granule 0 (channel 0, channel 1), granule 1 (channel 0, channel 1)}`
pDstMainDataBegin	`main_data_begin` フィールドへのポインタ。
pDstPrivateBits	`private bits` フィールドへのポインタ。
pDstScfsi	現在のフレームに関連するスケール係数選択情報へのポインタ。このデータは、次の順序で、*pDstScfsi* によって指定されるバッファ内に連続して配置される。 `{channel 0 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3), channel 1 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3)}`

intel®

説明

関数 ippsUnpackSideInfo_MP3 は、ippac.h ファイルで宣言される。この関数は、入力ビットストリームからサイド情報をアンパックする。関数 ippsUnpackSideInfo_MP3 呼び出す前に、ポインタ **ppBitStream* は、現在のフレームに関連するサイド情報を格納するビットストリームの第 1 バイトを指していなければならない。呼び出し元に戻る前に、この関数は、サイド情報の次のバイトを参照するように、ポインタ **ppBitStream* を更新する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *ppBitStream*、*pDstSideInfo*、*pDstMainDataBegin*、*pDstPrivateBits*、*pDstScfsi*、*pFrameHeader*、または *ppBitStream* のうち少なくとも 1 つが NULL。
ippStsMP3FrameHeaderErr	エラー。MP3 フレーム・ヘッダ構造体のいくつかの要素が無効。 *pFrameHeader*->id != 0 *pFrameHeader*->id != 1 *pFrameHeader*->layer != 1 *pFrameHeader*->mode < 0 *pFrameHeader*->mode > 3
ippStsMP3SideInfoErr	エラー。*window_switching_flag* フラグがセットされている場合に *block_type* の値がゼロ。

UnpackScaleFactors_MP3

スケール係数をアンパックする。

```
IppStatus ippsUnpackScaleFactors_MP3_1u8s(Ipp8u** ppBitStream,
    int* pOffset, Ipp8s* pDstScaleFactor, IppMP3SideInfo* pSideInfo,
    int* pScfsi, IppMP3FrameHeader* pFrameHeader, int granule,
    int channel);
```

引数

ppBitStream	現在のフレーム、最小単位、チャネルのスケール係数に関連するビットストリーム・バッファの第 1 バイトへのポインタへのポインタ。この関数は、このパラメータを更新する。
pOffSet	**ppBitStream* によって参照されるバイト内の次のビットへのポインタ。このパラメータは、0 〜 7 の範囲で有効である。ここで、0 は最上位ビット、7 は最下位ビットに対応する。この関数は、このパラメータを更新する。
pSideInfo	現在の最小単位およびチャネルに関連する MP3 サイド情報構造体へのポインタ。
pScfsi	現在のチャネルのスケール係数選択情報へのポインタ。
channel	チャネル・インデックス。値は 0 または 1 。
granule	最小単位インデックス。値は 0 または 1。
pFrameHeader	現在のフレームの MP3 フレーム・ヘッダ構造体へのポインタ。
pDstScaleFactor	ロング・ブロックまたはショート・ブロック、あるいはその両方のスケール係数ベクトルへのポインタ。

説明

関数 ippsUnpackScaleFactors_MP3 は、ippac.h ファイルで宣言される。この関数は、1 つのチャネルの 1 つの最小単位のショート・ブロックまたはロング・ブロック、あるいはその両方のスケール係数をアンパックし、その結果をベクトル *pDstScaleFactor* に格納する。呼び出し元に戻る前に、この関数は、入力ビットストリーム内の次の利用可能なビットを指すように、**ppBitStream* および **pOffset* ポインタを更新する。

注 : MPEG-2 強度モード : 強度位置がその最大値に等しいか、または位置が無効な場合、この無効な位置は負に設定される。その結果、再量子化モジュール内では、負の位置は無効な位置を示す。強度位置として扱われないスケール係数は、使用する前に正の位置に設定する必要がある。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。*ppBitStream*、*pOffset*、*pDstScaleFactor*、*pSideInfo*、*pScfsi*、*ppBitStream*、または *pFrameHeader* が NULL。
ippStsBadArgErr	エラー。 • *pOffset* > 7 • *pOffset* < 0 • 最小単位またはチャネル、あるいはその両方が [0, 1] を超えている。
ippStsMP3FrameHeaderErr	エラー。 • *pFrameHeader->id* が [0, 1] を超えている。 • *pFrameHeader->modeExt* が [0..3] を超えている。
ippStsMP3SideInfoErr	エラー。 • *pSideInfo->blockType* が [0,3] を超えている。 • *pSideInfo->mixedBlock* が [0,1] を超えている。 • *pFrameHeader* が MPEG-1 ビットストリームを指定している場合に、*pSideInfo->sfCompress* が [0, 15] を超えているか、*pScfsi*[0..3] が [0, 1] を超えている。 • *pFrameHeader* が MPEG-2 ビットストリームを指定している場合に、*pSideInfo->sfCompress* が [0, 511] を超えている。

HuffmanDecode_MP3
HuffmanDecodeSfb_MP3
HuffmanDecodeSfbMbp_MP3

ハフマン・データをデコードする。

```
IppStatus ippsHuffmanDecode_MP3_1u32s(Ipp8u** ppBitStream,
    int* pOffset, Ipp32s* pDstIs, int* pDstNonZeroBound,
    IppMP3SideInfo* pSideInfo, IppMP3FrameHeader* pFrameHeader, int
    hufSize);
IppStatus ippsHuffmanDecodeSfb_MP3_1u32s(Ipp8u** ppBitStream, int*
    pOffset, Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo*
    pSideInfo, IppMP3FrameHeader* pFrameHeader, int hufSize,
    IppMP3ScaleFactorBandTableLong pSfbTableLong);
IppStatus, ippsHuffmanDecodeSfbMbp_MP3_1u32s (Ipp8u** ppBitStream, int*
    pOffset, Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo*
    pSideInfo, IppMP3FrameHeader* pFrameHeader, int hufSize,
    IppMP3ScaleFactorBandTableLong pSfbTableLong,
    IppMP3ScaleFactorBandTableShort pSfbTableShort,
    IppMP3MixedBlockPartitionTable pMbpTable);
```

引数

ppBitStream	現在の最小単位およびチャネルに関連するハフマン・コード・ワードを格納するビットストリームの第 1 バイトへのポインタへのポインタ。この関数は、このパラメータを更新する。
pOffset	**ppBitStream* によって参照されるビットストリーム内の開始ビットの位置へのポインタ。このパラメータは、0 ～ 7 の範囲で有効である。ここで、0 は最上位ビット、7 は最下位ビットに対応する。この関数は、このパラメータを更新する。
pSideInfo	現在の最小単位およびチャネルに関連するサイド情報を格納する MP3 構造体へのポインタ。
pFrameHeader	現在のフレームに関連するヘッダを格納する MP3 構造体へのポインタ。
pDstIs	現在の最小単位およびチャネルに関連する 576 個のスペクトル係数の量子化値の計算に使用される、デコードされたハフマン符号のベクトルへのポインタ。

pDstNonZeroBound	これより上ではすべての係数がゼロに設定されるスペクトル領域へのポインタ。
hufSize	現在の最小単位およびチャネルに関連するハフマン・コード・ビットの数。
pSfbTableLong	ロング・ブロックのスケール係数バンド・テーブルへのポインタ。
pSfbTableShort	ショート・ブロックのスケール係数バンド・テーブルへのポインタ。また、MPEG-1 または MPEG-2 規格のデフォルト・テーブルを使用することもできる。
pMbpTable	複合ブロック・パーティション・テーブルへのポインタ。

説明

関数 `ippsHuffmanDecode_MP3`、`ippsHuffmanDecodeSfb_MP3`、および `ippsHuffmanDecodeSfbMbp_MP3_1u32s` は、`ippac.h` ファイルで宣言される。これらの関数は、1 つのチャネルの 1 つの最小単位に関連する 576 個のスペクトル係数について、ハフマン符号をデコードする。呼び出し元に戻る前に、この関数は、デコードされたハフマン・コードのブロックに続く最初の新しいビットを格納するビットストリーム内のバイトを指すように、ポインタ **ppBitStream* を更新する。

関数 `ippsHuffmanDecodeSfb_MP3` は、ロング・ブロックで定義済みのスケール係数バンド・テーブルを使用できるようにする。あるいは、MPEG-1 または MPEG-2 規格のデフォルト・テーブルも使用できる。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *ppBitStream*、*pOffset*、*pDstIs*、*pDstNonZeroBound*、*pSideInfo*、*pFrameHeader*、または *ppBitStream* のうち少なくとも 1 つが `NULL`、または *pOffset* `< 0` か *pOffset* `> 7`。
`ippStsMP3FrameHeaderErr`	エラー。MP3 フレーム・ヘッダ構造体のいくつかの要素が無効。
`ippStsMP3SideInfoErr`	エラー。MP3 サイド情報構造体のいくつかの要素が無効、または *hufSize* `< 0` か *hufSize* `>` *pSideInfo->part23Len*。
`ippStsErr`	未知のエラー。

ReQuantize_MP3
ReQuantizeSfb_MP3

デコードされたハフマン符号を再量子化する。

```
IppStatus ippsReQuantize_MP3_32s_I(Ipp32s* pSrcDstIsXr, int*
    pNonZeroBound, Ipp8s* pScaleFactor, IppMP3SideInfo* pSideInfo,
    IppMP3FrameHeader* pFrameHeader, Ipp32s* pBuffer);
IppStatus ippsReQuantizeSfb_MP3_32s_I(Ipp32s* pSrcDstIsXr, int*
    pNonZeroBound, Ipp8s* pScaleFactor, IppMP3SideInfo* pSideInfo,
    IppMP3FrameHeader* pFrameHeader, Ipp32s* pBuffer,
    IppMP3ScaleFactorBandTableLong pSfbTableLong,
    IppMP3ScaleFactorBandTableShort pSfbTableShort);
```

引数

pSrcDstIsXr	デコードされたハフマン符号のベクトルへのポインタ。ステレオ・モードおよびデュアル・チャネル・モードでは、右チャネルのデータはアドレス &(*pSrcDstIsXr*[576]) から始まる。この関数は、このベクトルを更新する。
pNonZeroBound	これより上ではすべての係数がゼロに設定されるスペクトル範囲へのポインタ。ステレオ・モードおよびデュアル・チャネル・モードでは、左チャネルの境界は pNonZeroBound [0]、右チャネルの境界は pNonZeroBound [1] となる。
pScaleFactor	スケール係数バッファへのポインタ。ステレオ・モードおよびデュアル・チャネル・モードでは、右チャネルのスケール係数は &(*pScaleFactor* [IPP_MP3_SF_BUF_LEN]) から始まる。
pSideInfo	現在の最小単位のサイド情報へのポインタ。
pFrameHeader	現在のフレームのフレーム・ヘッダへのポインタ。
pBuffer	ワークスペース・バッファへのポインタ。バッファの長さは 576 サンプルでなければならない。
pSfbTableLong	ロング・ブロックのスケール係数バンド・テーブルへのポインタ。

pSfbTableShort	ショート・ブロックのスケール係数バンド・テーブルへのポインタ。

説明

関数 ippsReQuantize_MP3 と ippsReQuantizeSfb_MP3 は、ippac.h ファイルで宣言される。これらの関数は、デコードされたハフマン符号を再量子化する。合成フィルタ・バンクのスペクトル・サンプルは、ISO 規格で指定された再量子化の式を使用して、デコードされたハフマン符号から計算される。

必要に応じて、ステレオ Mid/Side（M/S）デコードまたはインテンシティ・デコード、あるいはその両方が適用される。この関数は、再量子化されたスペクトル・サンプルを *pSrcDstIsXr* に返す。

ショート・ブロックの場合、順序変更操作を実行できる。ユーザは、再量子化関数を呼び出す前に、*pBuffer* によって指定されるワークスペース・バッファを事前に割り当てる必要がある。ショート・ブロックの場合、*pNonZeroBound* によって指定される値は、順序変更されたシーケンスに従って再計算される。

また、関数 ippsReQuantizeSfb_MP3 は、ロング・ブロックまたはショート・ブロックで定義済みのスケール係数バンド・テーブルを使用することもできる。あるいは、MPEG-1 または MPEG-2 規格のデフォルト・テーブルも使用できる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pSrcDstIsXr*、*pNonZeroBound*、*pScaleFactor*、*pSideInfo*、*pFrameHeader*、または *pBuffer* が NULL。
ippStsBadArgErr	エラー。*pNonZeroBound* が [0, 576] を超えている。
ippStsMP3FrameHeaderErr	エラー。*pFrameHeader->id* が [0, 1] を超えている。*pFrameHeader->samplingFreq* が [0, 2] を超えている。*pFrameHeader->mode* が [0,3] を超えている。*pFrameHeader->modeExt* が [0,3] を超えている。

`ippStsMP3SideInfoErr`	エラー。ステレオ・モードのビットストリームで、左のブロック・タイプと右のタイプが異なる場合。 *pSideInfo*[ch].blockType が [0, 3] を超えている。 *pSideInfo*[ch].mixedBlock が [0, 1] を超えている。 *pSideInfo*[ch].globGain が [0, 255] を超えている。*pSideInfo*[ch].sfScale が [0, 1] を超えている。 *pSideInfo*[ch].preFlag が [0, 1] を超えている。 *pSideInfo*[ch].pSubBlkGain[w] が [0, 7] を超えている。ここで、ch は 0 ～ 1 の範囲内、*w* は 0 ～ 2 の範囲内である。
`ippStsErr`	未知のエラー。

MDCTInv_MP3

ハイブリッド合成フィルタ・バンクの
第 2 ステージを実行する。

```
IppStatus ippsMDCTInv_MP3_32s(Ipp32s* pSrcXr, Ipp32s* pDstY,
    Ipp32s* pSrcDstOverlapAdd, int nonZeroBound,  int* pPrevNumOfImdct,
    int blockType, int mixedBlock);
```

引数

pSrcXr	Q5.26 形式で表現される、現在のチャネルおよび最小単位の再量子化されたスペクトル・サンプルのベクトルへのポインタ。
pDstY	PQMF バンクへの入力となる、Q7.24 形式の IMDCT 出力のベクトルへのポインタ。
pSrcDstOverlapAdd	Overlap-Add バッファへのポインタ。このバッファは、前の最小単位の IMDCT 出力（Q7.24 形式）の重複する部分を格納する。この関数は、このバッファを更新する。

nonZeroBound	スペクトル係数の制限境界。現在の最小単位およびチャネルで、この境界を超えるのすべてのスペクトル係数はゼロになる。
pPrevNumOfImdct	直前の最小単位の現在のチャネルについて計算されるIMDCT の数へのポインタ。この関数は、現在の最小単位の IMDCT の数を参照するように、このパラメータを更新する。
blockType	ブロック・タイプのインジケータ。
mixedBlock	複合ブロックのインジケータ。

説明

関数 ippsMDCTInv_MP3 は、ippac.h ファイルで宣言される。この関数は、ハイブリッド合成フィルタ・バンクの第 1 ステージを実行する。次の操作が実行される。

- エイリアス・リダクション
- ブロック・サイズ指示子と複合ブロック・モードに基づく逆変形離散コサイン変換 (IMDCT)
- IMDCT 出力のオーバーラップ追加
- PQMF（Pseudo-Quadrature Mirror Synthesis Filter）バンクに先立つ周波数反転

IMDCT は重複変換であるため、ユーザは IMDCT Overlap-Add ステートを維持するために、*pSrcDstOverlapAdd* によって参照されるバッファを事前に割り当てる必要がある。

このバッファは、576 個の要素を格納しなければならない。合成フィルタ・バンクを最初に呼び出す前に、Overlap-Add バッファのすべての要素がゼロに設定されていなければならない。それ以降のすべての呼び出しの間、MP3 アプリケーションは、Overlap-Add バッファの内容を維持しなければならない。

ippsMDCTInv_MP3_32s への入口点で、Overlap-Add バッファには、前の最小単位の処理によって生成された IMDCT 出力が格納されている必要がある。ippsMDCTInv_MP3_32s の終了時に、Overlap-Add バッファには、現在の最小単位の処理によって生成される出力の重複する部分が格納される。

この関数からのリターン時に、IMDCT サブバンド出力サンプルは、次のように構成される。

pDstY[*j**32+subband]。これは、*j* = 0 から 17、subband = 0 から 31 の場合である。

Q5.26 は、固定小数点位置の前の 5 ビットと後の 26 ビットを使用して、固定小数点形式の 32 ビット値を表すように指定する。

Q7.24 は、固定小数点位置の前の 7 ビットと後の 24 ビットを使用して、固定小数点形式の 32 ビット値を表すように指定する。

注： ポインタ *pSrcXr* と *pDstY* は、異なるバッファを参照しなければならない。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsBadArgErr`	1 つ以上の指定されたポインタが NULL の場合のエラー状態を示す。
`ippStsErr`	次の入力データ・エラーのうち 1 つ以上が検出された場合のエラーを示す。 *blockType* が [0,3] を超えている、 または *mixedBlock* が [0,1] を超えている、 または *nonZeroBound* が [0,576] を超えている、 または **pPrevNumOfImdct* が [0,32] を超えている。

SynthPQMF_MP3

ハイブリッド合成フィルタ・バンクの
第 2 ステージを実行する。

```
IppStatus ippsSynthPQMF_MP3_32s16s(Ipp32s* pSrcY, Ipp16s* pDstAudioOut,
    Ipp32s* pVBuffer, int* pVPosition, int mode);
```

引数

pSrcY	Q7.24 形式の 32 個の IMDCT サブバンド入力サンプルのブロックへのポインタ。
pDstAudioOut	16 ビット符号付きリトルエンディアン形式の 32 個の再構築された PCM 出力サンプルのブロックへのポインタ。左チャネルと右チャネルは、*mode* フラグに従ってインターリーブされる。
pVBuffer	Q7.24 データを格納する入力ワークスペース・バッファへのポインタ。この関数は、このパラメータを更新する。

pVPosition	内部ワークスペース・インデックスへのポインタ。この関数は、このパラメータを更新する。
mode	PCM オーディオ出力チャネルをインターリーブするかどうかを指定するフラグ。この値が 1 の場合はインターリーブしない。値が 2 の場合はインターリーブする。

説明

関数 ippsSynthPQMF_MP3 は、ippac.h ファイルで宣言される。この関数は、ハイブリッド合成フィルタ・バンクの第 2 ステージを実行する。このステージは、IMDCT 出力の 32 サンプルの入力ブロックに対して 32 個の時間領域出力サンプルを生成する、クリティカルにサンプリングされた 32 チャネル PQMF 合成バンクである。

各入力ブロックについて、PQMF は、16 ビット符号付きリトル・エンディアン PCM サンプルの出力シーケンスを、*pDstAudioOut* によって指定されるベクトルに出力する。

mode が 2 の場合、左チャネルの出力サンプルと右チャネルの出力サンプルはインターリーブされ（すなわち、LRLRLR）、左チャネルのデータは次のように構成される。

pDstAudioOut [2**i*]。これは、*i* = 0 から 31 の場合である。

mode が 1 の場合、左チャネルの出力と右チャネルの出力はインターリーブされない。

PQMF バンクにはメモリが含まれるため、MP3 アプリケーションは、この関数の各呼び出しの間、2 つのステート変数を維持する必要がある。

最初に、アプリケーションは、PQMF 計算のために、サイズが 512x（チャネル数）のワークスペース・バッファを事前に割り当てる必要がある。このバッファは、ポインタ *pVBuffer* によって参照される。関数を最初に呼び出す前に、このバッファの要素はゼロに初期化されていなければならない。それ以降の呼び出しの間、現在の呼び出しに対する *pVBuffer* 入力には、前回の呼び出しによって生成された *pVBuffer* 出力が格納されていなければならない。

MP3 アプリケーションは、*pVBuffer* 以外に、ステート変数 *pVPosition* の値をゼロに初期化し、その後も維持する必要がある。MP3 アプリケーションは、デコーダのリセット時にのみ、*pVbuffer* または *pVPosition* に格納された値を変更する。このリセット値は、常にゼロでなければならない。

戻り値

ippStsNoErr	エラーなし。
ippNullPtrErr	エラー。ポインタ *pSrcY*、*pDstAudioOut*、*pVBuffer*、または *pVPosition* のうち少なくとも 1 つが NULL。

`ippStsBadArgErr`	エラー。指定されたポインタのうち少なくとも 1 つが NULL、または *mode* の値が 1 より小さいか 2 より大きい。あるいは、*pVPosition* の値が [0, 15] を超えている。
`ippStsErr`	未知のエラー。

アドバンスト・オーディオ符号化関数

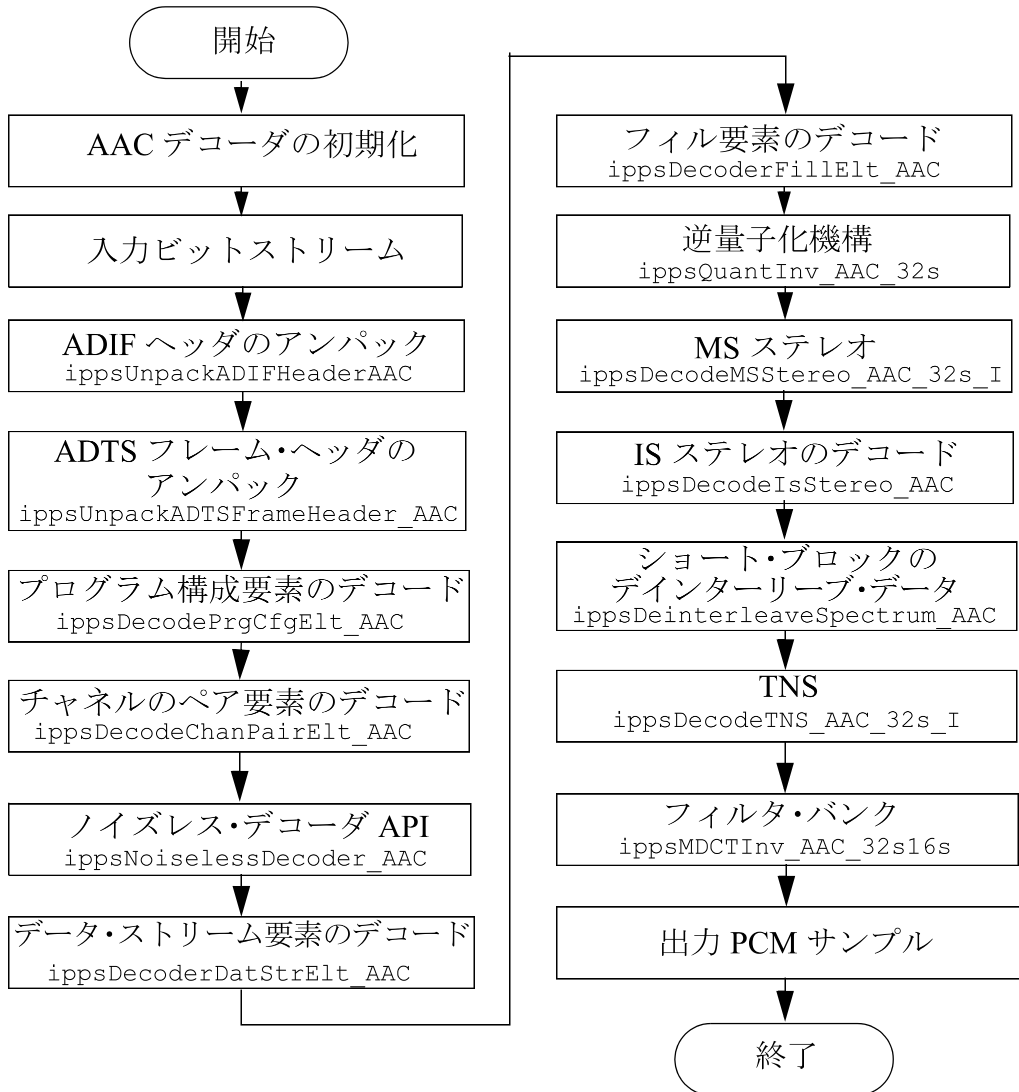
ISO/IEC 13818-7 MPEG-2 AAC（アドバンスト・オーディオ符号化）アルゴリズムは、5 チャネルの信号（左前方、右前方、中央、左後方、右後方）によるサラウンド信号の効率的な符号化手法である。

MPEG-2 AAC は、8 kHz から 96 kHz までのサンプリング周波数で最大 48 個のメイン・オーディオ・チャネルをサポートする。MPEG 形式テストの結果、AAC は 5 チャネルのオーディオ信号において ITU-R の品質条件を満たし、320 Kbps の AAC の音声品質は、640Kbps の MPEG-2 オーディオ BC（バックワード・コンパチブル）の音声品質より若干上回る。

その高い符号化効率により、AAC はデジタル放送システムの最有力候補とされ、DRM（Digital Radio Mondiale）システムによって採用されている。また、AAC は高品質の音楽をインターネットで配信する役割も果たす。さらに、AAC は次世代の「グローバル・マルチメディア言語」と呼ばれる MPEG-4 規格に準拠した唯一の高品質オーディオ符号化スキームである。

AAC デコーダ API は、ビットストリームのアンパックや AAC コア・デコード関数など、さまざまな AAC LC デコーダ関数を提供する。これにより、ユーザはより柔軟性に優れたデコーダ・システムの設定を行うことができる。ISO/IEC 13818-7:1997 を参照のこと。

図 10-4　AAC フローチャート



グローバル・マクロ

表 10-4 は、グローバル・マクロの定義を示す。

表 10-4　グローバル・マクロの定義

マクロのグローバル名	定義	注
IPP_AAC_FRAME_LEN	1024	1 フレームのデータ・サイズ
IPP_AAC_SF_LEN	120	スケール係数バッファのサイズ
IPP_AAC_GROUP_NUM_MAX	8	1 フレームの最大グループ数
IPP_AAC_TNS_COEF_LEN	60	TNS 係数バッファのサイズ
IPP_AAC_TNS_FILT_MAX	8	1 フレームの TNS フィルタの最大数
IPP_AAC_PRED_SFB_MAX	41	1 フレームの推定スケール係数バンドの最大数
IPP_AAC_ELT_NUM	16	1 プログラムの最大要素数
IPP_AAC_LFE_ELT_NUM	4	1 プログラムの低周波数拡張要素の最大数
IPP_AAC_DATA_ELT_NUM	8	1 プログラムのデータ要素の最大数
IPP_AAC_COMMENTS_LEN	256	コメント・フィールドの最大サイズ (バイト単位)

データ・タイプと構造体

この項では、インテル ® IPP のアドバンスト・オーディオ・エンコーダのデータ・タイプと構造体について説明する。

ADIF ヘッダ

```
typedef struct {
    Ipp32u  ADIFId;/* 32-bit, "ADIF" ASCII code */
    int     copyIdPres;         /* copy id flag: 0: off, 1: on */
    int     originalCopy;       /* original bitstream or copy, 0:
                                copy 1: original */
    int     home;
    int     bitstreamType;      /* bitstream flag: 0: constant
                                rate bitstream, 1: variable rate
                                bitstream */
    int     bitRate;            /* bit rate.if 0, unknown bit
                                rate */
```

```
    int     numPrgCfgElt;   /* number of program configure
                            elements */
    int     pADIFBufFullness[IPP_AAC_ELT_NUM];/* buffer
                                              fullness */
    Ipp8u   pCopyId[9];         /* 72-bit copy id */
} IppAACADIFHeader;
```

ADTS フレーム・ヘッダ

```
typedef struct {
    /* ADTS fixed header */
    int id;                     /* ID 1*/
    int layer;                  /* layer index 0x3: Layer I
                                //             0x2: Layer II
                                //             0x1: Layer III */
    int protectionBit;          /* CRC flag 0: CRC on, 1: CRC off */
    int profile;                /* profile: 0:MP, 1:LC, 2:SSR */
    int samplingRateIndex;      /* sampling rate index */
    int privateBit;             /* private_bit, no use  */
    int chConfig;               /* channel configure */
int originalCopy;               /* original bitstream or copy, 0:
                                copy, 1: original */
    int home;
    int emphasis;               /* not used by ISO/IEC 13818-7,
                                but used by 14496-3 */

    /* ADTS variable header */
    int cpRightIdBit;           /* copyright id bit */
    int cpRightIdStart;         /* copyright id start */
    int frameLen;               /* frame length in bytes */
    int ADTSBufFullness;        /* buffer fullness */
    int numRawBlock;            /* number of raw data blocks in
                                the frame */

    /* ADTS CRC error check, 16bits   */
    int CRCWord;                    /* CRC-check word */
} IppAACADTSFrameHeader;
```

単一チャネル・サイド情報

```
typedef struct {
    /* unpacked from the bitstream */
    int     icsReservedBit;     /* reserved bit */
    int     winSequence;        /* window sequence flag */
    int     winShape;           /* window shape flag, 0: sine
                                window, 1: KBD window */
    int     maxSfb;         /* maximum effective scale factor
                            bands */
    int     sfGrouping;         /* scale factor grouping
                                information */
    int     predDataPres;       /* prediction data present flag
                                for one frame, 0: prediction
                                off, 1: prediction on */
    int     predReset;          /* prediction reset flag, 0:
                                reset off, 1: reset on */
    int     predResetGroupNum; /* prediction reset group number */
    Ipp8u pPredUsed[IPP_AAC_PRED_SFB_MAX+3];/* prediction flag
                                buffer for each scale factor
                                band: 0: off, 1: on buffer
                                length 44 bytes, 4-byte align*/
    /* decoded from the above info */
    int     numWinGrp;          /* group number */
    int     pWinGrpLen[IPP_AAC_GROUP_NUM_MAX];/* buffer for
                                number of windows in each group */
} IppAACIcsInfo;
```

AAC スケーラブル・メイン要素ヘッダ

```
typedef struct{
    int windowSequence; //the windows is short or long type
    int windowShape;//what window is used for the right hand
//part of this analysis window
    int maxSfb;//number of scale factor band transmitted
    int sfGrouping;//grouping of short spectral data


    int numWinGrp;//window group number
    int pWinGrpLen[IPP_AAC_GROUP_NUM_MAX];//length of every
group
int msMode;// MS stereo flag: 0 - none, 1 - different // for
every sfb, 2 - all
```

```
    Ipp8u (*ppMsMask)[IPP_AAC_SF_MAX];//if MS's used in one
                                       sfb, when msMode ==1
    IppAACTnsInfo pTnsInfo[IPP_AAC_CHAN_NUM];//TNS structure
                                              for two channels
    IppAACLtpInfo pLtpInfo[IPP_AAC_CHAN_NUM];//LTP structure
                                              for two channels
}IppAACMainHeader;
```

AAC スケーラブル拡張要素ヘッダ

```
typedef struct{
    int msMode;//0,non; 1,part; 2,all
    int maxSfb;// number of scale factor band for extension layer
    Ipp8u (*ppMsMask)[IPP_AAC_SF_MAX]; //if ms is used
IppAACTnsInfo pTnsInfo[IPP_AAC_CHAN_NUM]; // TNS structure for
                                          Stereo
int pDiffControlLr[IPP_AAC_CHAN_NUM][IPP_AAC_PRED_SFB_MAX];
//FSS information for stereo
}IppAACExtHeader;
```

1 レイヤの TNS 構造体

```
typedef struct{
    int tnsDataPresent;
    int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX];
// TNS number filter buffer
    int pTnsFiltCoefRes[IPP_AAC_GROUP_NUM_MAX];
// TNS coef resolution flag
    int pTnsRegionLen[IPP_AAC_TNS_FILT_MAX];
// TNS filter length
    int pTnsFiltOrder[IPP_AAC_TNS_FILT_MAX];
// TNS filter order
    int pTnsDirection[IPP_AAC_TNS_FILT_MAX];
// TNS filter direction flag
    int pTnsCoefCompress[IPP_AAC_GROUP_NUM_MAX];
// The most significant bit of the coefficients of the //noise
shaping filter in window w is omitted or not
    Ipp8s pTnsFiltCoef[IPP_AAC_TNS_COEF_LEN];
// Coefficients of  one noise shaping filter applied to
//window w
}IppAACTnsInfo;
```

LTP 構造体

```
typedef struct{
    int ltpDataPresent;//if ltp is used
    int ltpLag;//the optimal delay from 0 to 2047
    Ipp16s ltpCoef;//indicate the  LTP coefficient
    int pLtpLongUsed[IPP_AAC_MAX_LTP_SFB]; // if long block use ltp
    int pLtpShortUsed[IPP_AAC_WIN_MAX]; //if short block use ltp
    int pLtpShortLagPresent[IPP_AAC_WIN_MAX];
//if short lag is transmitted
    int pLtpShortLag[IPP_AAC_WIN_MAX];
//relative delay for short window
}IppAACLtpInfo;
```

チャネルのペア要素

```
typedef struct {
    int     commonWin;          /* common window flag, 0: off,
                                1: on */
    int     msMaskPres;         /* MS stereo mask present flag */
Ipp8u       pMsUsed[IPP_AAC_SF_LEN];   /* MS stereo flag buffer
                                       for each scale factor
                                       band */
} IppAACChanPairElt;
```

チャネル情報

```
typedef struct {
    int     tag;
    int     id;             /* element id */
    int     samplingRateIndex; /* sampling rate index */
    int     predSfbMax;     /* maximum prediction scale factor
                            bands */
    int     preWinShape;    /* previous block window shape */

    int     winLen;         /* 128: if short window, 1024:
                            others */
    int     numWin;         /* 1 for long block, 8 for short
                            block */
    int     numSwb;         /* decided by sampling frequency and
                            block type */
```

```
    /* unpacking from the bitstream */
    int     globGain;       /* global gain */
    int     pulseDataPres;  /* pulse data present flag, 0: off,
                            1: on */
    int     tnsDataPres;    /* TNS data present flag, 0: off, 1:
                            on */
    int     gainContrDataPres; /* gain control data present
                               flag, 0: off, 1: on */

    /* icsInfo pointer */
    IppAACIcsInfo *pIcsInfo;   /* pointer to IppAACIcsInfo
                               structure */

    /* channel pair pointer */
    IppAACChanPairElt *pChanPairElt;  /* pointer to
                                      IppAACChanPairElt
                                      structure */

    /* section data */
    Ipp8u pSectCb[IPP_AAC_SF_LEN];    /* section code book
                                      buffer */
    Ipp8u pSectEnd[IPP_AAC_SF_LEN];   /* the end of scale
                                      factor offset in each
                                      section */
    int pMaxSect[IPP_AAC_GROUP_NUM_MAX];  /* maximum section
                                          number for each group
                                          */
    int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX];/*TNS number filter
                                           number buffer*/
    /* TNS data */
    int pTnsFiltCoefRes[IPP_AAC_GROUP_NUM_MAX];/* TNS
                                               coefficients
                                               resolution flag
                                               */
    int pTnsRegionLen[IPP_AAC_TNS_FILT_MAX];  /* TNS filter
                                              length */
    int pTnsFiltOrder[IPP_AAC_TNS_FILT_MAX];  /* TNS filter
                                              order */
    int pTnsDirection[IPP_AAC_TNS_FILT_MAX];  /* TNS filter
                                              direction flag */
}IppAACChanInfo;
```

MPEG-2 AAC プリミティブ

この項では、パラメータ *maxSfb* は、グループごとに転送され、ビットストリームからアンパックされるスケール係数バンドの数を指定する。パラメータ *numSwb* は、ショート・ブロックまたはロング・ブロックにおけるスケール係数窓バンドの数を示す。これらの値は、サンプリング・レートとブロック・タイプに基づいて計算される。

ISO/IEC 13818-7:1997 の条項 8.3.1 を参照のこと。

UnpackADIFHeader_AAC

AAC ADIF フォーマット・ヘッダを取得する。

```
IppStatus ippsUnpackADIFHeader_AAC (Ipp8u **ppBitStream,
    IppAACADIFHeader *pADIFHeader, IppAACPrgCfgElt *pPrgCfgElt, int
    prgCfgEltMax);
```

引数

ppBitStream	ADIF ヘッダより前の、現在のバイトへの二重ポインタ。
prgCfgEltMax	プログラム設定要素の最大数。この値は、[1, 16] の範囲でなければならない。
ppBitStream	ADIF ヘッダより後の、現在のバイトへの二重ポインタ。
pADIFHeader	IppACCADIFHeader 構造体へのポインタ。
pPrgCfgElt	IppAACPrgCfgElt 構造体へのポインタ。バッファには prgCfgEltMax 個の要素がなければならない。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、プログラム設定要素を含む AAC ADIF 形式のヘッダを入力ビットストリームから取得する。ISO/IEC 13818-7:1997 の表 6.2 および 6.21 を参照のこと。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *ppBitStream*、*pADIFHeader*、*pPrgCfgElt*、または **ppBitStream* のうち少なくとも 1 つが NULL。
ippStsAacPrgNumErr	エラー。デコードした時に *pADIFHeader->numPrgCfgElt* > *prgCfgEltMax*、または *prgCfgEltMax* が [1,IPP_AAC_MAX_ELT_NUM] の範囲外。

注： pADIFHeader->numPrgCfgElt は、ビットストリームから直接アンパックした数に 1 を追加した値である。

UnpackADTSFrameHeader_AAC

入力ビットストリームから ADTS フレーム・ヘッダを取得する。

```
IppStatus ippsUnpackADTSFrameHeader_AAC (Ipp8u **ppBitStream,
    IppAACADTSFrameHeader *pADTSFrameHeader);
```

引数

ppBitStream	現在のバイトへの二重ポインタ。
ppBitStream	ADTS フレーム・ヘッダをアンパックした後の、現在のバイトへの二重ポインタ。
pADTSFrameHeader	IppAACADTSFrameHeader 構造体へのポインタ。

説明

この関数は、入力ビットストリームから ADTS フレーム・ヘッダを取得する。CRC ワードが適用される場合、16 ビット CRC ワードの第 1 バイトは、pADTSFrameHeader->CRCWord[15:8] に格納され、第 2 バイトは pADTSFrameHeader->CRCWord[7:0] に格納される。この関数は、ヘッダが破壊されているかをチェックしない。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *ppBitStream*、*pOffest*、*pPrgCfgElt*、または **ppBitStream* のうち少なくとも 1 つが NULL。
ippStsAacAdtsSyncWordErr	同期ワード・エラー。

DecodePrgCfgElt_AAC

入力ビットストリームからプログラム構成要素を取得する。

```
IppStatus ippsDecodePrgCfgElt_AAC (Ipp8u **ppBitStream, int *pOffset,
    IppAACPrgCfgElt *pPrgCfgElt);
```

引数

ppBitStream	現在のバイトへの二重ポインタ。
pOffset	**ppBitStream* によって参照されるバイト内のビット位置へのポインタ。0 から 7 の範囲で有効である。 0 は、最上位ビットを示す。 7 は、最下位ビットを示す。
ppBitStream	プログラム設定要素をデコードした後の、現在のバイトへの二重ポインタ。
pOffset	**ppBitStream* によって参照されるバイト内のビット位置へのポインタ。0 から 7 の範囲で有効である。 0 は、最上位ビットを示す。 7 は、最下位ビットを示す。
pPrgCfgElt	IppAACPrgCfgElt 構造体へのポインタ。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、入力ビットストリームからプログラム設定要素を取得する。ISO/IEC 13818-7 の条項 8.5 を参照のこと。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *ppBitStream*、*pOffset*、*pPrgCfgElt*、または **ppBitStream* のうち少なくとも 1 つが NULL。
ippStsAacBitOffsetErr	エラー。*pOffset* が [0,7] の範囲外。

DecodeChanPairElt_AAC

入力ビットストリームから *channel_pair_element* を取得する。

```
IppStatus ippsDecodeChanPairElt_AAC (Ipp8u **ppBitStream, int *pOffset,
    IppAACIcsInfo *pIcsInfo, IppAACChanPairElt *pChanPairElt, int
    predSfbMax);
```

引数

ppBitStream	現在のバイトへの二重ポインタ。
pOffset	**ppBitStream* によって参照されるバイト内のビット位置へのポインタ。0 から 7 の範囲で有効である。 0 は、最上位ビットを示す。 7 は、最下位ビットを示す。
predSfbMax	最大推定スケール係数バンド。LC　プロファイルでは、*predSfbMax* = 0 にセットする。
ppBitStream	チャネルのペア要素をデコードした後の、現在のバイトへの二重ポインタ。
pOffset	**ppBitStream* によって参照されるバイト内のビット位置へのポインタ。0 から 7 の範囲で有効である。 0 は、最上位ビットを示す。 7 は、最下位ビットを示す。
pIcsInfo	IppAACIcsInfo 構造体へのポインタ。 pIcsInfo->predDataPres = 0 の場合、*pIcsInfo->predReset* = 0 にセットする。

	pIcsInfo->pWinGrpLen に含まれた最初の *pIcsInfo->numWinGrp* 要素にのみ意味がある。表 10-5 に示すように、構造体のメンバを変更してはならない。
pChanPairElt	IppAACChanPairElt 構造体へのポインタ。表 10-6 に示すように、構造体のメンバを変更してはならない。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、入力ビットストリームからチャネルのペア要素を取得する。単一チャネル・ストリームは含まれていない。

入力ビットストリームからデコードした *common_window* フラグが 0 の場合、*pIcsInfo* と *pChanPairElt* のすべてのメンバは *pChanPairElt->commonWin* を除いて変更されない。

ISO/IEC 13818-7:1997 の条項 8.3、表 6.10、および 6.11 を参照のこと。

表 10-5　変更されない *pIcsInfo* のメンバ

変更されないメンバ	条件
sfGrouping	*pIcsInfo->winSequence* != 2
predResetGroupNum	*pIcsInfo->predDataPres* == 0 || *pIcsInfo->predReset* == 0
pPredUsed[sfb]	*pIcsInfo->predDataPres* == 0

表 10-6　変更されない *pChanPairElt* のメンバ

メンバ	条件
pMsUsed[sfb]	*pChanPairElt*->msMaskPres != 1

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *ppBitStream*、*pOffset*、**ppBitStream*、*pIcsInfo*、または *pChanPairElt* のうち少なくとも 1 つが NULL。
ippStsAacBitOffsetErr	エラー。*pOffset* が [0,7] の範囲外。

`ippStsAacMaxSfbErr`	エラー。ビットストリームからデコードした *pIcsInfo->maxSfb* が IPP_AAC_MAX_SFB（すべてのサンプリング周波数の最大スケール係数バンド）より大きい。

NoiselessDecoder_LC_AAC

1 つのチャネルのすべてのデータをデコードする。

```
IppStatus ippsNoiselessDecoder_LC_AAC (Ipp8u **ppBitStream, int
    *pOffset, int commonWin, IppAACChanInfo *pChanInfo, Ipp16s
    *pDstScalefactor, Ipp32s *pDstQuantizedSpectralCoef, Ipp8u
    *pDstSfbCb, Ipp8s *pDstTnsFiltCoef);
```

引数

ppBitStream	現在のバイトへの二重ポインタ。
pOffset	1 バイトのオフセットへのポインタ。
pChanInfo	チャネル情報 IppAACChanInfo 構造体へのポインタ。メンバ *samplingRateIndex* と predSfbMax は、入力として扱われる。
commonWin	共通の窓インジケータ。
ppBitStream	ビットストリーム・バッファへの二重ポインタ。
pOffset	1 バイトのオフセットへのポインタ。
pChanInfo	チャネル情報へのポインタ。IppAACChanInfo 構造体。表 10-7 に示すように、*pIcsInfo* は *pChanInfo*->pIcsInfo を意味する。
pDstScalefactor	スケール係数または強度位置バッファへのポインタ。バッファ・サイズは 120 以上である。各グループには *maxSfb* 要素のみ格納される。シーケンス・グループの間には、スペースを含めない。

pDstQuantizedSpectralCoef 量子化されたスペクトル係数データへのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上である。

pDstSfbCb スケール係数バンド・コードブックへのポインタ。バッファ・サイズは 120 以上でなければならない。各グループの *maxSfb* 要素を格納する。シーケンス・グループの間には、スペースを含めない。

pDstTnsFiltCoef TNS 係数へのポインタ。バッファ・サイズは 60 以上でなければならない。格納シーケンスは、各窓のそれぞれのフィルタの TNS 順序要素である。対応する TNS がゼロの場合、要素は変更されない。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、1 つのチャネルからすべてのデータをデコードする。これらのデータには、スケール係数、強度位置、スペクトル係数、TNS 係数、および LC プロファイルに関連するサイド情報が含まれる。

この関数を呼び出す前に、*pChanInfo*->pIcsInfo、*pChanInfo*->*samplingRateIndex*、*pChanInfo*->predSfbMax をセットしてポインタの値を修正する必要がある。

表 10-7 *pChanInfo* の入力 / 出力メンバ一覧

メンバ	出力
Tag	使用しない。
id	使用しない。
preWinShape	使用しない。
pChanPairElt	使用しない。
samplingRateIndex	入力として使用する。変更されない。
predSfbMax	入力として使用する。ゼロでなければならず、変更されない。
winLen	出力として使用する。デコードした *pIcsInfo*->*winSequence* がショート・ブロックの場合、128 にセットされる。それ以外の場合、1024 にセットされる。
numWin	出力として使用する。デコードした pIcsInfo->*winSequence* がショート・ブロックの場合、8 にセットされる。それ以外の場合、1 にセットされる。

表 10-7 *pChanInfo* の入力 / 出力メンバ一覧

メンバ	出力
numSwb	出力として使用する。*samplingRateIndex* and *pIcsInfo->winSequence* に基づいて、各グループのスケール係数窓バンドの最大数をセットする。ISO/IEC 13818-7:1997 の表 8.4-8.1 を参照のこと。
globGain *pulseDataPres* *tnsDataPres* *gainContrDataPres*	出力として使用する。ビットストリームからアンパックする。
pMaxSect	出力として使用する。各グループの最大セクション番号へのポインタ。バッファ内の *pIcsInfo->numWinGrp* 要素にのみ意味がある。
pSectCb	出力として使用する。セクション・コードブックへのポインタ。各グループには *pMaxSect*[g] 要素のみ格納される。シーケンス・グループの間には、スペースを含めない。
pTnsRegionLen	出力として使用する。各窓で 1 つのフィルタが適用されるスケール係数バンド単位の領域の長さへのポインタ。
pTnsFiltOrder	出力として使用する。各窓に適用される TNS フィルタの順序へのポインタ。
pTnsDirection	出力として使用する。フィルタが上方向または下方向に適用されることを示すトークンへのポインタ。0 は上方向、1 は下方向を示す。
pIcsInfo	*commonWin* == 1 の場合、入力として使用する。 *commonWin* == 0 の場合、出力として使用する。 *pIcsInfo->predDataPres* == 0 の場合、 *pIcsInfo->predReset* = 0 にセットする。 *pIcsInfo-> pWinGrpLen* に含まれた最初の *pIcsInfo->numWinGrp* 要素にのみ意味がある。特別な条件では、構造体のいくつかのメンバは変更してはならない。表 10-5 を参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *ppBitStream*、*pOffset*、*pChanInfo*、*pDstScalefactor*、*pDstQuantizedSpectralCoef*、*pDstSfbCb*、*pDstTnsFiltCoef*、*pChanInfo->pIcsInfo*、または **ppBitStream* のうち少なくとも 1 つが NULL。
ippStsAacBitOffsetErr	エラー。**pOffset* が [0,7] の範囲外。
ippStsAacComWinErr	エラー。*commonWin* が [0,1] を超えている。
ippStsAacSmplRateIdxErr	エラー。*pChanInfo->samplingRateIndex* が [0,11] を超えている。
ippStsAacPredSfbErr	エラー。*pChanInfo->predSfbMax* がゼロでない。
ippStsAacMaxSfbErr	エラー。*pChanInfo->pIcsInfo->maxSfb* > *pChanInfo->numSwb*。

ippStsAacSectCbErr	エラー。*pChanInfo->pSectCb* が参照したコードブックが無効、または *(*pChanInfo->pSectCb*)==12, 13。 現在のチャネルがペア・チャネルの右チャネルでない場合、**pSectCb* = 14, 15 も無効である。
ippStsAacPlsDataErr	エラー。*pChanInfo->pIcsInfo->winSequence* がショート・シーケンスを示しており、*pChanInfo->pulsePres* がパルス・データの存在を示している。 パルス・データ >= *pChanInfo->numSwb* またはパルス・データ位置 *offset* >= *winLen* のスケール係数バンドを開始する。
ippStsAacGainCtrErr	エラー。*pChanInfo->gainControlPres* が 1。つまり、ゲイン・コントロール・データが存在している。ゲイン・コントロール・データは現在サポートされていない。
ippStsAacCoefValErr	エラー。*pDstCoef* が参照する量子化した係数値が [-8191, 8191] を超えている。

DecodeDatStrElt_AAC

入力ビットストリームからデータ・
ストリーム要素を取得する。

```
IppStatus ippsDecodeDatStrElt_AAC (Ipp8u **ppBitStream, int *pOffset,
    int *pDataTag, int *pDataCnt, Ipp8u * pDstDataElt);
```

引数

ppBitStream	現在のバイトへの二重ポインタ。
pOffset	**ppBitStream* によって参照されるバイト内のビット位置へのポインタ。0 から 7 の範囲で有効である。 0 は、最上位ビットを示す。 7 は、最下位ビットを示す。
ppBitStream	データ・ストリーム要素をデコードした後の、現在のバイトへの二重ポインタ。

pOffset	**ppBitStream* によって参照されるバイト内のビット位置へのポインタ。0 から 7 の範囲で有効である。 0 は、最上位ビットを示す。 7 は、最下位ビットを示す。
pDataTag	*element_instance_tag* へのポインタ。ISO/IEC 13818-7:1997 の表 6.20 を参照のこと。
pDataCn	データ・サイズの値へのポインタ（バイト単位）。
pDstDataElt	入力ビットストリームから抽出されたデータ・ストリームを含むデータ・ストリーム・バッファへのポインタ。バッファには、*pDstDataElt* が参照する 512 個の要素がある。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、入力ビットストリームからデータ・ストリーム要素を取得する。

ISO/IEC 13818-7:1997 の条項 8.6、表 6.20 を参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *ppBitStream*、*pOffset*、**ppBitStream*、*pDataTag*、*pDataCnt*、または *pDstDataElt* のうち少なくとも 1 つが NULL。
ippStsAacBitOffsetErr	エラー。**pOffset* が [0,7] の範囲外。

DecodeFillElt_AAC

入力ビットストリームからフィル要素を取得する。

```
IppStatus ippsDecodeFillElt_AAC (Ipp8u **ppBitStream, int *pOffset,
    int *pFillCnt, Ipp8u * pDstFillElt);
```

引数

ppBitStream	現在のバイトへのポインタへのポインタ。

pOffset	**ppBitStream* によって参照されるバイト内のビット位置へのポインタ。 0 から 7 の範囲で有効である。 0 は、最上位ビットを示す。 7 は、最下位ビットを示す。
ppBitStream	フィル要素をデコードした後の、現在のバイトへのポインタへのポインタ。
pOffset	**ppBitStream* が参照するバイト内のビット位置へのポインタ。 0 から 7 の範囲で有効である。 0 は、最上位ビットを示す。 7 は、最下位ビットを示す。
pFillCnt	全フィル・データのサイズの値へのポインタ（バイト単位）。
pDstFillElt	フィル・データ・バッファへのポインタ。バッファの長さは 270 以上でなければならない。

説明

この関数は、`ippac.h` ファイルで宣言される。この関数は、入力ビットストリームからフィル要素を取得する。

ISO/IEC 13818-7:1997 の条項 8.7、表 6.22 を参照のこと。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。ポインタ *ppBitStream*、*pOffset*、**ppBitStream*、*pFillCnt*、または *pDstFillElt* のうち少なくとも 1 つが `NULL`。
`ippStsAacBitOffsetErr`	エラー。**pOffset* が `[0,7]` の範囲外。

QuantInv_AAC

現在のチャネルのハフマン符号の逆量子化を実行する（インプレース方式）。

```
IppStatus ippsQuantInv_AAC_32s_I (Ipp32s *pSrcDstSpectralCoef, const
    Ipp16s *pScalefactor, int numWinGrp, const int *pWinGrpLen, int
    maxSfb, const Ipp8u *pSfbCb, int samplingRateIndex, int winLen);
```

引数

pSrcDstSpectralCoef	入力では、量子化された入力係数へのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上でなければならない。 出力では、Q13.18 形式の逆量子化されたデスティネーション係数へのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。 バッファ・サイズは 1024 以上でなければならない。出力 *pSrcDstSpectralCoef*[*i*] の最大エラーは、表 10-8 に示す。
pScalefactor	スケール係数バッファへのポインタ。バッファ・サイズは 120 以上でなければならない。
numWinGrp	グループ番号。
pWinGrpLen	各グループにある窓の数へのポインタ。バッファ・サイズは 8 以上でなければならない。
maxSfb	現在のブロックのスケール係数バンドの最大数。
pSfbCb	スケール係数バンド・コードブックへのポインタ。バッファ・サイズは 120 以上でなければならない。各グループの *maxSfb* 要素にのみ意味がある。シーケンス・グループの間には、スペースを含めない。
samplingRateIndex	サンプリング・レート・インデックス。[0, 11] の範囲で有効である。ISO/IEC 13818-7:1997 の表 6.5 を参照のこと。
winLen	1 つの窓のデータ番号。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、次の式に従って、現在のチャネルのハフマン符号の逆方向の量子化を実行する。

$$pSrcDst[i] = sign(pSrcDst[i])*(pSrcDst[i])^{\frac{4}{3}} *2^{\left(\frac{1}{2}(pScalefactor[sfb]-100)\right)}$$

ISO/IEC 13818-7:1997 の条項 10 を参照のこと。

表 10-8 *pSrcDstSpectralCoef* の計算エラー一覧

出力	条件	
max(error (*pSrcDstSpectralCoef* [*i*]))	Input abs (*pSrcDstSpectralCoef* [*i*])	Output abs (*pSrcDstSpectralCoef*[*i*])
3	<= 128	< 2 ^ 29
3	129~8191	<= 2 ^ 25
7	129~8191	< 2 ^ 29

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDstSpectralCoef*、*pScalefactor*、*pSfbCb*、*pWinGrpLen* のうち少なくとも 1 つが NULL。
ippStsAacSmplRateIdxErr	エラー。*pChanInfo->samplingRateIndex* が [0, 11] を超えている。
ippStsAacMaxSfbErr	エラー。*maxSfb* が [0,IPP_AAC_MAX_SFB] を超えている。
ippStsAacWinGrpErr	エラー。*numWinGrp* がロング窓で [0, 8] を超えているか、またはショート窓で 1 でない。
ippStsAacWinLenErr	エラー。*winLen* が 128 または 1024 でない。
ippStsAacCoefValErr	エラー。*pSrcDstSpectralCoef* が参照する量子化した係数値が [-8191,8191] を超えている。

DecodeMsStereo_AAC

ペア・チャネルの M/S ステレオを処理する
（インプレース方式）。

```
IppStatus ippsDecodeMsStereo_AAC_32s_I (Ipp32s *pSrcDstL, Ipp32s
    *pSrcDstR, int msMaskPres, const Ipp8u *pMsUsed, Ipp8u *pSfbCb, int
    numWinGrp, const int *pWinGrpLen, int maxSfb, int
    samplingRateIndex, int winLen);
```

引数

pSrcDstL 入力では、Q13.18 形式の左チャネル・データへのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上でなければならない。

出力では、Q13.18 形式の左チャネル・データへのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上でなければならない。

pSrcDstR 入力では、Q13.18 形式の右チャネル・データへのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上でなければならない。

出力では、Q13.18 形式の右チャネル・データへのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上でなければならない。

msMaskPres MS ステレオ・マスク・フラグ。

- 0: MS オフ
- 1: MS オン
- 2: MS すべてのバンドがオン。

pMsUsed MS ステレオ・フラグ・バッファへのポインタ。バッファ・サイズは 120 以上でなければならない。

pSfbCbPointer スケール係数バンド・コードブックへのポインタ。バッファ・サイズは 120 以上でなければならない。各グループの *maxSfb* 要素を格納する。シーケンス・グループの間には、スペースを含めない。

numWinGrp	グループ番号。
pWinGrpLen	各グループにある窓の数へのポインタ。バッファ・サイズは 8 以上でなければならない。
maxSfb	現在のブロックのスケール係数バンドの最大数。
samplingRateIndex	サンプリング・レート・インデックス。[0, 11] の範囲で有効である。ISO/IEC 13818-7:1997 の表 6.5 を参照のこと。
winLen	1 つの窓のデータ番号。
pSrcDstR	Q13.18 形式の右チャネル・データへのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上でなければならない。
pSfbCb	スケール係数バンド・コードブックへのポインタ。 invert_intensity (group, sfb) = -1 および **pSfbCb* = INTERITY_HCB の場合、 **pSfbCb* = INTERITY_HCB2 とする。 **pSfbCb* = INTERITY_HCB2 の場合、 **pSfbCb* = INTERITY_HCB とする。 バッファ・サイズは 120 以上でなければならない。各グループの *maxSfb* 要素を格納する。シーケンス・グループの間には、スペースを含めない。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、ペア・チャネルの M/S ステレオ処理を行い、同時に invert_intensity(group, sfb) 関数を実行して値を *pSfbCb* バッファに格納する。

MS ステレオ・フラグがオンの場合、次の式に従って、*pSrcDstL*[i] と *pSrcDstR*[i] を処理する。

$$pSrcDstL'[i] = pSrcDstL[i] + PsrcDstR[i]$$
$$pSrcDstR'[i] = pSrcDstL[i] - pSrcDstR[i].$$

ISO/IEC 13818-7:1997 の条項 12 を参照のこと。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSrcDstL*、*pSrcDstR*、*pMsUsed*、*pWinGrpLen*、または *pSfbCb* のうち少なくとも 1 つが NULL。
ippStsAacMaxSfbErr	エラー。各窓で *samplingFreqIndex* と *maxSfb* から計算された係数インデックスが *winLen* を超えている。
ippStsAacSamplRateIdxErr	エラー。*pChanInfor->samplingRateIndex* が [0,11] を超えている。
ippStsAacWinGrpErr	エラー。ロング窓で *numWinGrp* が [0,8] を超えているか、またはショート窓で 1 に等しくない。
ippStsAacWinLenErr	エラー。*winLen* が 128 または 1024 でない。
ippStsStereoMaskErr	エラー。ステレオ・マスク・フラグが 1 または 2 でない。

DecodeIsStereo_AAC

ペア・チャネルのインテンシティ・ステレオを処理する。

```
IppStatus ippsDecodeIsStereo_AAC_32s (const Ipp32s *pSrcL, Ipp32s
    *pDstR, const Ipp16s *pScalefactor, const Ipp8u *pSfbCb, int
    numWinGrp, const int *pWinGrpLen, int maxSfb, int
    samplingRateIndex, int winLen);
```

引数

pSrcL	Q13.18 形式の左チャネル・データへのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上でなければならない。
pScalefactor	スケール係数バッファへのポインタ。バッファ・サイズは 120 以上でなければならない。

pSfbCb	スケール係数バンド・コードブックへのポインタ。バッファ・サイズは 120 以上でなければならない。各グループの *maxSfb* 要素を格納する。シーケンス・グループの間には、スペースを含めない。 1、-1、または 0 に等しい *pSfbCb*[sfb] の各値は、インテンシティ・ステレオ・モード（直接、逆方向、なし）を示す。
numWinGrp	グループ番号。
pWinGrpLen	各グループにある窓の数へのポインタ。バッファ・サイズは 8 以上でなければならない。
maxSfbMax	現在のブロックのスケール係数バンドの最大数。
samplingRateIndex	サンプリング・レート・インデックス。[0, 11] の範囲で有効である。ISO/IEC 13818-7:1997 の表 6.5 を参照のこと。
winLen	1 つの窓のデータ番号。
pDstR	Q13.18 形式の右チャネル・データへのポインタ。ショート・ブロックの場合、係数は各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上でなければならない。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、ペア・チャネルのインテンシティ・ステレオを処理する。

次の式に従って、*pSrcL*[*i*] と *pDstR*[*i*] を処理する。

$$pDstR[i] = pSrc[i] * is_intensity(g, sfb) * 2^{\left(-\frac{1}{4}pScalefactor[sfb]\right)}$$

注： invert_intensity(g, sfb) は、MS ステレオ処理プリミティブで既にデコードされ、pSfbCb[sfb] に格納されているため、この式では使用されない。ISO/IEC 13818-7:1997 の条項 12 を参照のこと。

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pSrcDstL*、*pSrcDstR*、*pScaleFactor*、*pWinGrpLen*、または *pSfbCb* のうち少なくとも 1 つが NULL。
ippStsAacMaxSfbErr	エラー。各窓で *samplingFreqIndex* と *maxSfb* から計算された係数インデックスが *winLen* を超えている。
ippStsAacSamplRateIdxErr	エラー。*pChanInfor->samplingRateIndex* が [0,11] を超えている。
ippStsAacWinGrpErr	エラー。ロング窓で *numWinGrp* が [0,8] を超えているか、またはショート窓で 1 に等しくない。
ippStsAacWinLenErr	エラー。*winLen* が 128 または 1024 でない。

DeinterleaveSpectrum_AAC

ショート・ブロックの係数を
デインターリーブする。

```
IppStatus ippsDeinterleaveSpectrum_AAC_32s (const Ipp32s *pSrc, Ipp32s
    *pDst, int numWinGrp, const int *pWinGrpLen, int maxSfb, int
    samplingRateIndex, int winLen);
```

引数

pSrc	ソース係数バッファへのポインタ。係数は、各グループ内のスケール係数窓バンドによってインターリーブされる。バッファ・サイズは 1024 以上でなければならない。
numWinGrp	グループ番号。
pWinGrpLen	各グループにある窓の数へのポインタ。バッファ・サイズは 8 以上でなければならない。
maxSfbMax	現在のブロックのスケール係数バンドの最大数。
samplingRateIndex	サンプリング・レート・インデックス。[0, 11] の範囲で有効である。ISO/IEC 13818-7:1997 の表 6.5 を参照のこと。

winLen	1 つの窓のデータ番号。
pDst	係数の出力へのポインタ。 データ・シーケンスは、*pDst*[*w**128+*sfb***sfbWidth*[*sfb*]+*i*] の順序になる。ここで、*w* は窓インデックス、*sfb* はスケール係数バンド・インデックス、*sfbWidth* はスケール係数バンド幅テーブル、*i* はスケール係数バンド内のインデックスを示す。 バッファ・サイズは 1024 以上でなければならない。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、ショート・ブロックの係数をデインターリーブする。

ISO/IEC 13818-7:1997 の条項 8.3.5 を参照のこと。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、または *pWinGrpLen* のうち少なくとも 1 つが NULL。
ippStsAacMaxSfbErr	エラー。各窓で *samplingFreqIndex* と *maxSfb* から計算された係数インデックスが *winLen* を超えている。
ippStsAacSamplRateIdxErr	エラー。*pChanInfor->samplingRateIndex* が [0,11] を超えている。
ippStsAacWinGrpErr	エラー。*numWinGrp* が [0,8] を超えている。
ippStsAacWinLenErr	エラー。*winLen* が 128 でない。

DecodeTNS_AAC

TNS（Temporal Noise Shaping）をデコードする（インプレース方式）。

```
IppStatus ippsDecodeTNS_AAC_32s_I (Ipp32s *pSrcDstSpectralCoefs, const
    int *pTnsNumFilt, const int *pTnsRegionLen, const int
    *pTnsFiltOrder, const int *pTnsFiltCoefRes, const Ipp8s
    *pTnsFiltCoef, const int *pTnsDirection, int maxSfb, int profile,
    int samplingRateIndex, int winLen);
```

引数

pSrcDstSpectralCoefs 入力では、全極型フィルタでフィルタされる Q13.18 形式の入力スペクトル係数へのポインタ。バッファには、*pSrcDstSpectralCoefs* が参照する 1024 個の要素がある。

出力では、全極型フィルタでフィルタされた Q13.18 形式の出力スペクトル係数へのポインタ。

倍精度データに対する計算エラーについては、表 10-9 を参照のこと。

pTnsNumFilt 現在のフレームの各窓で使用するノイズ・シェーピング・フィルタへのポインタ。バッファには、*pTnsNumFilt* が参照する 8 個の要素がある。これらの要素は次のように配列される。

pTnsNumFilt[*w*]: 窓 *w* で使用したノイズ・シェーピング・フィルタの数（*w* = 0 から *numWin*-1 まで）。

pTnsRegionLen 現在のフレームの各窓で 1 つのフィルタが適用されるスケール係数バンド単位の領域の長さへのポインタ。

バッファには、*pTnsRegionLen* が参照する 8 個の要素がある。これらの要素は次のように配列される。

pTnsRegionLen[*i*]: 窓で適用されるフィルタ filt の領域の長さ。

$$w, i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt, w = 0$$

から *numWin*-1、*filt*=0 から *pTnsNumFilt*[*w*]-1 まで。

pTnsFiltOrder	現在のフレームの各窓に適用される1つのノイズ・シェーピング・フィルタの順序へのポインタ。バッファには、*pTnsFiltOrder*が参照する8個の要素がある。これらの要素は次のように配列される。 *pTnsFiltOrder*[i]：1つのノイズ・シェーピング・フィルタ filt の順序へのポインタ。このフィルタは窓 w に適用される。 $i= w, i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt, w = 0$ から *numWin*-1、*filt*=0 から *pTnsNumFilt*[w]-1 まで。
pTnsFiltCoefRes	現在のフレームの各窓で転送したフィルタ係数の 3 ビットまたは4ビットの解像度へのポインタ。バッファには、*pTnsFiltCoefRes*が参照する8個の要素がある。これらの要素は次のように配列される。 *pTnsFiltCoefRes*[w]：窓 *w* で転送したフィルタ係数の解像度。*w* = 0 から *numWin*-1 まで。
pTnsFiltCoef	現在のフレームの各窓に適用される1つのノイズ・シェーピング・フィルタの係数へのポインタ。バッファには、*pTnsFiltCoef*が参照する60個の要素がある。これらの要素は次のように配列される。 *pTnsFiltCoef*[*i*],*pTnsFiltCoef*[*i*+1], …, *pTnsFiltCoef*[i+order-1]：窓 w に適用される1つのノイズ・シェーピング・フィルタ *filt* の係数。 この順序は、ノイズ・シェーピング・フィルタ *filt* が窓 w に適用される順序と同じである。w = 0 から numWin-1、filt=0 から *pTnsNumFilt*[w]-1 まで。 例えば、*pTnsFiltCoef*[0]、*pTnsFiltCoef*[1]、…、*pTnsFiltCoef*[order0-1] は、窓0に適用されるノイズ・シェーピング・フィルタ0の係数である。 また、*pTnsFiltCoef*[*order0*]、*pTnsFiltCoef*[*order0*+1]、…、*pTnsFiltCoef*[order0+order1-1] は、window 0 に適用されるノイズ・シェーピング・フィルタ1となる。 order0 は、窓0に適用されるノイズ・シェーピング・フィルタ0と同じである。order1 は、窓0に適用されるノイズ・シェーピング・フィルタ1と同じである。

	窓 0 が処理された後は、窓 1、窓 2 と処理され、*numWin* 番目の窓まで処理される。
pTnsDirection	フィルタが上方向または下方向に適用されることを示すトークンへのポインタ。 0 は上方向、1 は下方向を示す。 バッファには、*pTnsDirection* が参照する 8 個の要素がある。これらの要素は次のように配列される。 *pTnsDirection*[i]：フィルタ filt が上方向または下方向に適用されることを示すトークン。このフィルタは窓 w に適用される。 $i = w, i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt, w = 0$ から numWin-*1*、filt=0 から *pTnsNumFilt*[*w*]-1 まで。
maxSfb	現在のフレームの各窓グループごとに転送されるスケール係数バンドの数。
profile	ISO/IEC 13818-7:1997 の表 7.1 に示されているプロファイル・インデックス。
samplingRateIndex	現在のフレームのサンプリング・レートを示すインデックス。
winLen	1 つの窓のデータ番号。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、TNS（Temporal Noise Shaping）のデコード処理を実行する。これは、各変換窓内の量子化ノイズの継時形状を制御する。

TNS デコード処理は、全極型フィルタを選択したスペクトラル係数の領域に適用することで、現在のフレームの各窓ごとにそれぞれ実行される。

表 10-9 *pSrcDstSpectralCoefs* の計算エラー一覧

MAX(error(*pSrcDstSpectralCoefs*[*i*]))	条件
4095	8 == numWin
32767	1 == numWin

numWin は、現在のフレームの窓シーケンスにある窓の数である。窓シーケンスが EIGHT_SHORT_SEQUENCE の場合、*numWin* は 8 に等しい。それ以外の場合、1 となる。

numSwb は、実際の窓タイプ（現在のフレームのロング窓またはショート窓）のスケール係数窓バンドの合計数である。

注 : この関数は、LC プロファイルのみをサポートする。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDstSpectralCoefs*、*pTnsNumFilt*、*pTnsRegionLen*、*pTnsFiltOrder*、*pTnsFiltCoefRes*、*pTnsFiltCoef*、または *pTnsDirection* のうち少なくとも 1 つが NULL。
IppStsTnsProfielErr	エラー。*profile* != 1。
ippStsAacTnsNumFiltErr	エラー。データ・エラーが発生。 • ショート窓シーケンスの場合、*pTnsNumFilt*[*w*] が [0,1] を超えている。 • ロング窓シーケンスの場合、*pTnsNumFilt*[*w*] が [0,3] を超えている。
ippStsAacTnsLenErr	エラー。**pTnsRegionLen* が [0, *numSwb*] を超えている。
ippStsAacTnsOrderErr	エラー。データ・エラーが発生。 • ショート窓シーケンスの場合、**pTnsFiltOrder* が [0,7] を超えている。 • ロング窓シーケンスの場合、**pTnsFiltOrder* が [0,12] を超えている。

ippStsAacTnsCoefResErr	エラー。*pTnsFiltCoefRes*[w] が [3, 4] を超えている。
ippStsAacTnsCoefErr	エラー。**pTnsFiltCoef* が [-8, 7] を超えている。
ippStsAacTnsDirectErr	エラー。**pTnsDirection* が [0, 1] を超えている。

注： *numWin* は、現在のフレームの窓シーケンスにある窓の数である。窓シーケンスが EIGHT_SHORT_SEQUENCE の場合、*numWin* は 8 に等しい。それ以外の場合、1 となる。

注： *numSwb* は、実際の窓タイプ（現在のフレームのロング窓またはショート窓）のスケール係数窓バンドの合計数である。

MDCTInv_AAC

時間−周波数領域信号をマップし、1024 個の 16 ビット符号付きリトル・エンディアン PCM サンプルを再構築する。

```
IppStatus ippsMDCTInv_AAC_32s16s (Ipp32s *pSrcSpectralCoefs, Ipp16s
    *pDstPcmAudioOut, Ipp32s *pSrcDstOverlapAddBuf, int winSequence,
    int winShape, int prevWinShape, int pcmMode);
```

引数

pSrcSpectralCoefs	Q13.18 形式の入力時間−周波数領域におけるサンプル。バッファには、*pSrcDstSpectralCoef* が参照する 1024 個の要素がある。
pSrcDstOverlapAddBuf	Q13.18 形式の直前のブロックにおける窓処理されたシーケンスの後半が格納された Overlap-Add バッファへのポインタ。バッファには 1024 個の要素がある。
winSequence	現在のブロックで使用される窓シーケンスを示すフラグ。

winShape	現在のブロックで選択された窓関数を示すフラグ。
prevWinShape	直前のブロックで選択された窓関数を示すフラグ。
pcmMode	PCM オーディオ出力チャネルがインターリーブ（すなわち、LRLRLR）されるかどうかを指定するフラグ。1 の場合、インターリーブされない。2 の場合、インターリーブされる。
pDstPcmAudioOut	Q15 形式の再構築された 1024 個の 16 ビット符号付きリトル・エンディアン出力 PCM サンプルへのポインタ。必要に応じてインターリーブされる。*pDstPcmAudioOut* の最大計算エラーは、各ベクトル要素に対して 1 より小さい。ベクトルの 2 次エラーの合計は、96 より小さい。
pSrcDstOverlapAddBuf	Q13.18 形式の直前のブロックにおける窓処理されたシーケンスの後半が格納された Overlap-Add バッファへのポインタ。*pDstPcmAudioOut* の最大計算エラーは、各ベクトル要素に対して 4 より小さい。ベクトルの 2 次エラーの合計は、1536 より小さい。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、時間－周波数領域信号をマップし、1024 個の 16 ビット符号付きリトル・エンディアン PCM サンプルを各チャネルの出力として再構築する。

このモジュールは、次のように構成されている。

- IMDCT 変換
- 窓処理
- Overlap-Add 操作

フィルタバンクの時間 / 周波数の解像度を入力信号の特性に適応するために、ブロック・スイッチ・ツールも適用される。各チャネルでは、時間－周波数領域における 1024 個のサンプルが IMDCT によって時間領域に変換される。

窓操作を適用した後は、窓処理されたシーケンスの前半が、直前のブロックにおける窓処理されたシーケンスの後半に追加され、各チャネルごとに 1024 個のサンプルを再構築する。出力は、*pcmMode* に基づいてインターリーブできる。

pcmMode が 2 の場合、出力のシーケンスは *pDstPcmAudioOut*[2*i]、i=0 ～ 1023 となる。つまり、1024 個の出力サンプルが、*pDstPcmAudioOut*[0], *pDstPcmAudioOut*[2], *pDstPcmAudioOut*[4],..., *pDstPcmAudioOut*[2046] のシーケンスで格納される。

pcmMode が 1 の場合、出力のシーケンスは *pDstPcmAudioOut*[i]、i=0 ～ 1023 となる。

また、*pSrcDstOverlapAddBuf* が参照する Overlap-Add 操作の入出力バッファを事前に割り当てる必要がある。

最初に呼び出す前にこのバッファをゼロにリセットし、現在の呼び出しの出力を、同じチャネルの次の呼び出しの入力として使用する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcSpectralCoefs*、*pSrcDstOverlapAddBuf*、および *pDstPcmAudioOut* のうち少なくとも 1 つが NULL。
ippStsAacWinSeqErr	エラー。*winSequence* が [0,3] を超えている。
ippStsAacWinShapeErr	エラー。*winShape* または *prevWinShape* が [0,1] を超えている。
ippStsAacPcmModeErr	エラー。*pcmMode* が [1,2] を超えている。

MPEG-4 AAC プリミティブ

この項では、MPEG-4 AAC 操作のプリミティブについて説明する。

DecodeMainHeader_AAC

ビットストリームからメイン・ヘッダ情報とメイン・レイヤ情報を取得する。

```
IppStatus ippsDecodeMainHeader_AAC(Ipp8u **ppBitStream, int *pOffset,
    IppAACMainHeader *pAACMainHeader, int channelNum, int
    monoStereoFlag);
```

引数

ppBitStream	ビットストリーム・バッファへの二重ポインタ。
pOffset	1 バイトのオフセットへのポインタ。
channelNum	チャネル数。
monoStereoFlag	現在のフレームのレイヤがモノおよびステレオであることを示す。
ppBitStream	メイン要素をデコードした後の、ビットストリーム・バッファへの二重ポインタ。
pOffset	メイン要素をデコードした後の、1 バイトのオフセットへのポインタ。
pAACMainHeader	メイン要素ヘッダへのポインタ。

説明

この関数は、`ippac.h` ファイルで宣言される。この関数は、ビットストリームからメイン・ヘッダ情報とメイン・レイヤ情報を取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *ppBitStream*、*pAACMainHeader*、**ppBitStream*、または *pOffset* のうち少なくとも 1 つが NULL。
ippStsAacBitOffsetErr	エラー。*pOffset* が [0,7] を超えている。

intel®

ippStsAacChanErr	エラー。*channelNum* が [1,2] を超えている。
ippStsAacMonoStereoErr	エラー。*monoStereoFlag* が [0,1] を超えている。

DecodeExtensionHeader_AAC

ビットストリームから拡張ヘッダ情報と
拡張レイヤ情報を取得する。

```
IppStatus ippsDecodeExtensionHeader_AAC(Ipp8u **ppBitStream, int
    *pOffset, IppAACExtHeader *pAACExtHeader, int monoStereoFlag, int
    thisLayerStereo, int monoLayerFlag, int preStereoMaxSfb, int
    hightstMonoMaxSfb, int winSequence);
```

引数

ppBitStream	ビットストリーム・バッファへの二重ポインタ。
pOffset	1 バイトのオフセットへのポインタ。
monoStereoFlag	現在のフレームのレイヤがモノおよびステレオであることを示すフラグ。
thisLayerStereo	現在のレイヤがステレオであることを示すフラグ。
monoLayerFlag	現在のレイヤがモノであることを示すフラグ。
preStereoMaxSfb	直前のステレオ・レイヤ *maxSfb*。
hightstMonoMaxSfb	最後のモノ・レイヤ *maxSfb*。
winSequence	窓のタイプ（ショートまたはロング）。
pAACExtHeader	拡張要素ヘッダへのポインタ。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、ビットストリームから拡張ヘッダ情報と拡張レイヤ情報を取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsBadArgErr	エラー。ポインタ *ppBitStream*、*pAACExtHeader*、または *pOffset* のうち少なくとも 1 つが NULL。
ippStsAacBitOffsetErr	エラー。**pOffset* が [0,7] の範囲外。

ippStsAacStereoLayerErr	エラー。*thisLayerStereo* が [0,1] を超えている。
ippStsAacMonoLayerErr	エラー。*monoLayerFlag* が [0,1] を超えている。
ippStsAacMaxSfbErr	エラー。*maxSfb* が [0,IPP_AAC_MAX_SFB] を超えている。
ippStsAacMonoStereoErr	エラー。*monoStereoFlag* が [0,1] を超えている。
ippStsAacWinSeqErr	エラー。*winSequence* が [0,3] を超えている。

DecodePNS_AAC

個々のチャネル・ストリーム（ICS）内で
知覚ノイズ置換（PNS）符号化を実装する。

```
IppStatus ippsDecodePNS_AAC_32s(Ipp32s *pSrcDstSpec, int
    *pSrcDstLtpFlag, Ipp8u *pSfbCb, Ipp16s *pScaleFactor, int
    maxSfb,int numWinGrp, int *pWinGrpLen,int samplingFreqIndex, int
    winLen,int *pRandomSeed);
```

引数

pSrcDstSpec	知覚ノイズ置換（PNS）のスペクトル係数へのポインタ。
pSrcDstLtpFlag	長期予測（LTP）フラグへのポインタ。
pSfbCb	スケール係数コードブックへのポインタ。
pScaleFactor	スケール係数値へのポインタ。
maxSfb	このレイヤで使用するスケール係数バンドの数。
numWinGrp	窓グループの数。
pWinGrpLen	各窓グループの長さへのポインタ。
samplingFreqIndex	サンプリング周波数のインデックス。
winLen	窓の長さ。ロング窓は 1024、ショート窓は 128 である。
pRandomSeed	PNS のランダム・シード。
pSrcDstSpec	知覚ノイズで置換した出力スペクトルへのポインタ。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、個々のチャネル・ストリーム（ICS）内で知覚ノイズ置換（PNS）符号化を実装する。特定のスペクトル係数のセットは、ハフマン符号化による符号および逆量子化処理からではなく、ランダム・ベクトルから計算される。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDstSpec*、*pSfbCb*、*pScaleFactor*、*pRandomSeed*、*pWinGrpLen* または *pSrcDstLtpFlag* のうち少なくとも 1 つが NULL。
ippStsAacMaxSfbErr	エラー。*maxSfb* が [0,IPP_AAC_MAX_SFB] を超えている。
ippStsAacSmplRateIdxErr	エラー。*pChanInfo->samplingRateIndex* が [0,12] を超えている。
ippStsAacWinLenErr	エラー。*winLen* が 128 または 1024 でない。

LongTermReconstruct_AAC

LTR（Long Term Reconstruct、長期再構築）を使用して、連続した符号化フレーム間における信号の冗長性を削減する。

```
IppStatus ippsLongTermReconstruct_AAC_32s(Ipp32s *pSrcEstSpec, Ipp32s
    *pSrcDstSpec, int *pLtpFlag, int winSequence, int
    samplingFreqIndex);
```

引数

pSrcDstSpec	LTP のスペクトル係数へのポインタ。
pSrcEstSpec	周波数領域ベクトルへのポインタ。
winSequence	窓のタイプ（ショートまたはロング）。
samplingFreqIndex	サンプリング周波数のインデックス。
pLtpFlag	LTP フラグへのポインタ。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、長期再構築（LTP）を使用して、連続した符号化フレーム間における信号の冗長性を削減する。

LTP は、順方向アダプティブ予測機構である。これは、デコーダの数値丸めエラーまたは転送したスペクトル係数の重複エラーに対して本質的に精密ではない。

デコードしたスペクトル係数のベクトルおよび対応する周波数領域ベクトルを追加して、再構築したスペクトル係数のベクトルを取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDstSpec*、*pSrcEstSpec*、および *pLtpFlag* のうち少なくとも 1 つが NULL。
ippStsAacSmplRateIdxErr	エラー。*pChanInfo->samplingRateIndex* が [0,12] を超えている。
ippStsAacWinSeqErr	エラー。*winSequence* が [0,3] を超えている。

MDCTFwd_AAC

PCM サンプルのスペクトル係数を生成する。

```
IppStatus ippsMDCTFwd_AAC_32s(Ipp32s *pSrc, Ipp32s *pDst, Ipp32s
    *pSrcDstOverlapAdd, int winSequence, int winShape, int preWinShape,
    Ipp32s *pWindowedBuf);
```

引数

pSrc	MDCT を実行するための継時信号へのポインタ。
pSrcDstOverlapAdd	オーバーラップ・バッファへのポインタ。MPEG-4 AAC のデコードでは使用しない。
winSequence	ブロックがロングまたはショートであることを示す窓シーケンス。
winShape	現在の窓の形状。
preWinShape	直前の窓の形状。
pWindowedBuf	MDCT の作業バッファ。バッファのサイズは 2048 ワード以上でなければならない。

pSrcDstOverlapAdd	オーバーラップ・バッファへのポインタ。MPEG-4 AAC のデコードでは使用しない。
pDst	MDCT の出力。これは、PCM サンプルのスペクトル係数。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、MDCT 長期再構築（LTP）ループ内の PCM サンプルのスペクトル係数を生成する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc*、*pDst*、*pWindowedBuf*、または *pSrcDstOverlapAdd* のうち少なくとも 1 つが NULL。
ippStsAacWinSeqErr	エラー。*winSequence* が [0,3] を超えている。
ippStsAacWinShapeErr	エラー。*preWinShape* が [0,1] を超えている。

EncodeTNS_AAC

長期再構築ループで逆 TNS を実行する（インプレース方式）。

```
IppStatus ippsEncodeTNS_AAC_32s_I(Ipp32s *pSrcDst, const int
    *pTnsNumFilt, const int *pTnsRegionLen, const int *pTnsFiltOrder,
    const int *pTnsFiltCoefRes, const Ipp8s *pTnsFiltCoef, const int
    *pTnsDirection, int maxSfb, int profile, int samplingFreqIndex, int
    winLen);
```

引数

pSrcDst	入力では、TNS エンコード操作のスペクトル係数へのポインタ。 出力では、TNS エンコード操作後のスペクトル係数へのポインタ。
pTnsNumFilt	TNS フィルタの数へのポインタ。
pTnsRegionLen	TNS フィルタの長さへのポインタ。
pTnsFiltOrder	TNS フィルタの順序へのポインタ。

pTnsFiltCoefRes	TNS 係数の解像度フラグへのポインタ。
pTnsFiltCoef	TNS フィルタ係数へのポインタ。
pTnsDirection	TNS 方向フラグへのポインタ。
maxSfb	スケール係数の最大数。
profile	自動プロファイル。
samplingFreqIndex	サンプリング周波数のインデックス。
winLen	窓の長さ。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、LTP ループまたは ATNS（Analysis Temporal Noise Shaping）で逆 TNS を行う（インプレース方式）。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrcDstSpectralCoef*、*pTnsNumFilt*、*pTnsRegionLen*、*pTnsFiltOrder*、*pTnsFiltCoefRes*、pTnsFiltCoef、または *pTnsDirection* のうち少なくとも 1 つが NULL。
ippStsTnsProfileErr	エラー。*profile* != 1。
ippStsAacTnsNumFiltErr	エラー。**pTnsNumFilt* がショート窓シーケンスで [0, 1] を超えているか、またはロング窓シーケンスで [0, 3] を超えている。
ippStsAacTnsLenErr	エラー。**pTnsRegionLen* が [0,*numSwb*] を超えている。
ippStsAacTnsOrderErr	エラー。**pTnsFiltOrder* がショート窓シーケンスで [0, 7] を超えているか、またはロング窓シーケンスで [0, 12] を超えている。
ippStsAacTnsCoefResErr	エラー。**pTnsFiltCoefRes* が [3, 4] を超えている。
ippStsAacTnsCoefErr	エラー。**pTnsFiltCoef* が [-8, 7] を超えている。

ippStsAacTnsDirectErr	エラー。*pTnsDirection* が [0,1] を超えている。
ippStsAacSmplRateIdxErr	エラー。*samplingRateIndex* が [0, 12] を超えている。
ippStsAacWinLenErr	エラー。*winLen* が 128 または 1024 でない。

LongTermPredict_AAC

予測された時間領域信号を長期再構築（LTP）ループから取得する。

```
IppStatus ippsLongTermPredict_AAC_32s(Ipp32s *pSrcTimeSignal, Ipp32s
    *pDstEstTimeSignal, IppAACLtpInfo *pAACLtpInfo, int winSequence);
```

引数

pSrcTimeSignal	継時領域内で予測される継時信号へのポインタ。
pDstEstTimeSignal	LTP 後のサンプルの出力へのポインタ。
pAACLtpInfo	LTP 情報へのポインタ。
winSequence	窓のタイプ（ショートまたはロング）。
pDstEstTimeSignal	時間領域の予測出力へのポインタ。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、予測された時間領域信号を LTP ループから取得する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上のポインタが NULL。
ippStsAacWinSeqErr	エラー。*winSequence* が [0,3] を超えている。

NoiseLessDecode_AAC

ノイズレス・デコードを実行する。

```
IppStatus ippsNoiseLessDecode_AAC (Ipp8u **ppBitStream, int *pOffset,
    IppAACMainHeader *pAACMainHeader,Ipp16s *pDstScaleFactor, Ipp32s
    *pDstQuantizedSpectralCoef,  Ipp8u *pDstSfbCb, Ipp8s
    *pDstTnsFiltCoef, IppAACChanInfo *pChanInfo, int winSequence, int
    maxSfb, int commonWin, int scaleFlag, int audioObjectType);
```

引数

ppBitStream	分析されるビットストリームへの二重ポインタ。
pOffset	1 バイトのオフセットへのポインタ。
pAACMainHeader	メイン・ヘッダ情報へのポインタ。スケーラブル・オブジェクトでは使用しない。 *commonWin* == 0 && *scaleFlag*==0 の場合、LTP 情報をデコードして、*pAACMainHeader->pLtpInfo*[] に保存する必要がある。
pChanInfo	チャネル情報構造体へのポインタ。
windowSequence	窓のタイプ（ショートまたはロング）。
maxSfb	スケール係数バンドの数。
commonWin	チャネル・ペアが同じ ICS 情報を使用するかを示す。
scaleFlag	スケーラブルなタイプが使用されるかどうかを示すフラグ。
audioObjectType	オーディオ・オブジェクト・タイプのインジケータ。 • 1 はメイン・タイプを示す。 • 2 は LC タイプを示す。 • 6 はスケーラブル・モードを示す。

ppBitStream	分析されたビットストリームへの二重ポインタ。
pOffset	1 バイトのオフセットへのポインタ。
pChanInfo	チャネル情報構造体へのポインタ。
pDstScaleFactor	分析されたスケール係数へのポインタ。
pDstQuantizedSpectralCoef	ハフマン・デコードした後の、量子化されたスペクトル係数へのポインタ。
pDstSfbCb	スケール係数コードブックのインデックスへのポインタ。
pDstTnsFiltCoef	TNS フィルタ係数へのポインタ。スケーラブル・オブジェクトでは使用しない。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、MPEG-2 および MPEG-4 オブジェクト用の一般的なノイズレス・デコード・モジュールである。

1 つのスケール係数バンドが MPEG-4 AAC スケーラブル・オブジェクトで PNS を使用する場合、**pDstScaleFactor* にはこのスケール係数バンドのノイズ・エネルギーが格納され、*pDstSfbCb*[sfb] は NOISE_HCB(13) となる。このスケール係数バンドのスペクトルには、ハフマン・デコードを行う必要はなく、このスケール係数バンドの *pDstQuantizedSpectralCoef* にはゼロを指定できる。

AAC スケーラブル・オブジェクトでは、*pDstTnsFiltCoef* と *pAACMainHeader* は使用されない。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *ppBitStream*、*pOffset*、**ppBitStream*、*pAACMainHeader*、*pDstScaleFactor*、*pDstTnsFiltCoef*、*pDstQuantizedSpectralCoef*、*pChanInfo*、または *pDstSfbCb* のうち少なくとも 1 つが NULL。
ippStsAacBitOffsetErr	エラー。**pOffset* が [0,7] を超えている。

ippStsAacComWinErr	エラー。*commonWin* が [0,1] を超えている。
ippStsAacMaxSfbErr	エラー。*maxSfb* が [0,IPP_AAC_MAX_SFB] を超えている。
ippStsAacSmplRateIdxErr	エラー。*pChanInfo->samplingRateIndex* が [0,11] を超えている。
ippStsAacCoefValErr	エラー。*pDstCoef* が参照する量子化した係数値が [-8191,8191] を超えている。

LtpUpdate_AAC

必要なバッファの更新を長期再構築（LTP）ループで実行する。

```
IppStatus ippsLtpUpdate_AAC_32s (Ipp32s *pSpecVal, Ipp32s *pLtpSaveBuf,
    int winSequence, int winShape, int preWinShape, Ipp32s *pWorkBuf);
```

引数

pSpecVal	LTP ループで TNS デコーダ後のスペクトル値へのポインタ。
pLtpSaveBuf	LTP の保存バッファへのポインタ。バッファのサイズは、3**frameLength* でなければならない。
winSequence	窓のタイプ。 • 0 はロングを示す。 • 1 はロング・スタートを示す。 • 2 はショートを示す。 • 3 はロング・ストップを示す。
winShape	窓の形状（KBD または SIN）。
preWinShape	直前の窓の形状。
pWorkBuf	LTP 更新の作業バッファ。*pWorkBuf* のサイズは 2048*3 = 6144 ワード以上でなければならない。

pLtpSaveBuf	LTP　の保存バッファへのポインタ。バッファのサイズは、3**frameLength* でなければならない。値は次のフレーム用に保存される。

説明

この関数は、ippac.h ファイルで宣言される。この関数は、必要なバッファの更新を長期再構築（LTP）ループで実行する。この操作には、IMDCT と保存バッファの更新が含まれる。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。*pLtpSaveBuf*、*pWorkBuf*、または *pSpecVal* のうち少なくとも 1 つが NULL。
ippStsAacWinSeqErr	エラー。*winSequence* が [0,3] を超えている。
ippStsAacWinShapeErr	エラー。*winShape* または *preWinShape* が [0,1] を超えている。

文字列関数

11

本章では、文字列演算を実行するインテル® IPP 関数について説明する。インテル IPP の文字列関数は、ゼロを文字列の終端記号として認識しないが、文字列の長さ (要素の数) を明示的に指定する必要がある。
また、文字列の重複はサポートされていない (インプレース演算以外の場合)。
インテル IPP の文字列関数は、`Ipp8u` および `Ipp16u` という 2 つのデータ・タイプを使用する。

このグループのすべての関数を表 11-1 に示す。

表 11-1　インテル® IPP の文字列関数

関数の基本名	操作
Find, FindRev	指定された文字列に一致する最初のサブ文字列を検索する。
FindC FindRevC	指定された要素に一致する最初の要素をソース文字列から検索する。
FindCAny FindRevCAny	指定された配列のいずれかの要素に一致する最初の要素をソース文字列から検索する。
Insert	指定された文字列を他の文字列に挿入する。
Remove	指定された数の要素を文字列から削除する。
Compare	2 つの固定長文字列を比較する。
CompareIgnore, CompareIgnoreLatin	大文字・小文字を無視して 2 つの固定長文字列を比較する。
Equal	2 つの固定長文字列が等しいかを比較する。
TrimC	指定された文字を文字列の最初と最後からすべて削除する。
TrimCAny	指定されたいずれかの文字に一致する要素をソース文字列の最初と最後からすべて削除する。
TrimStartCAny TrimEndCAny	指定されたいずれかの文字に一致する要素をソース文字列の最初または最後からすべて削除する。
ReplaceC	ソース文字列の指定された要素を他の要素に置換する。
Uppercase, UppercaseLatin	文字列のアルファベット文字をすべて大文字に変換する。
Lowercase, LowercaseLatin	文字列のアルファベット文字をすべて小文字に変換する。
Hash	文字列のハッシュ値を計算する。
Concat	複数の文字列を連結する。

表 11-1　インテル® IPP の文字列関数

関数の基本名	操作
ConcatC	複数の文字列を連結し、各文字列間に区切り文字を挿入する。
SplitC	ソース文字列を分割する。

Find, FindRev

指定された文字列に一致する最初の
サブ文字列を検索する。

```
IppStatus ippsFind_8u(const Ipp8u* pSrc, int len, const Ipp8u*
      pFind,
      int lenFind, int* pIndex);
IppStatus ippsFind_16u(const Ipp16u* pSrc, int len, const Ipp16u*
      pFind,
      int lenFind, int* pIndex);
IppStatus ippsFindRev_8u(const Ipp8u* pSrc, int len, const Ipp8u*
      pFind,
      int lenFind, int* pIndex);
IppStatus ippsFindRev_16u(const Ipp16u* pSrc, int len, const
      Ipp16u* pFind,
      int lenFind, int* pIndex);
```

引数

pSrc	ソース文字列へのポインタ。
len	ソース文字列の要素数。
pFind	参照文字列へのポインタ。
lenFind	参照文字列の要素数。
pIndex	結果インデックスへのポインタ。

説明

関数 ippsFind と ippsFindRev は、ippch.h ファイルで宣言される。これらの関数は、指定された参照文字列 *pFind* に一致する要素のサブ文字列を、ソース文字列 *pSrc* から検索する。最初に一致したサブ文字列の開始位置は *pIndex* に格納さ

れる。一致するサブ文字列が見つからなかった場合、*pIndex* に -1 が設定される。
関数 ippsFindRev は、ソース文字列を逆順で検索する。
検索では、大文字・小文字を区別する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* または *lenFind* の値が負。

FindC
FindRevC

指定された要素に一致する最初の要素を
ソース文字列から検索する。

```
IppStatus ippsFindC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind,
      int* pIndex);
IppStatus ippsFindC_16u(const Ipp16u* pSrc, int len, Ipp16u
      valFind,
      int* pIndex);
IppStatus ippsFindRevC_8u(const Ipp8u* pSrc, int len, Ipp8u
      valFind,
      int* pIndex);
IppStatus ippsFindRevC_16u(const Ipp16u* pSrc, int len, Ipp16u
      valFind,
      int* pIndex);
```

引数

pSrc	ソース文字列へのポインタ。
len	ソース文字列の要素数。
valFind	指定された要素の値。
pIndex	結果インデックスへのポインタ。

説明

関数 ippsFindC と ippsFindRevC は、ippch.h ファイルで宣言される。これらの関数は、指定された値 *valFind* を格納した要素に一致する最初の要素を、ソース文字列 *pSrc* から検索する。この要素の位置は *pIndex* に格納される。一致する要

素が見つからなかった場合、*pIndex* に -1 が設定される。関数 ippsFindRev は、ソース文字列を逆順で検索する。
検索では、大文字・小文字を区別する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* の値が負。

FindCAny, FindRevCAny

指定された配列のいずれかの要素に一致する最初の要素をソース文字列から検索する。

```
IppStatus ippsFindCAny_8u(const Ipp8u* pSrc, int len,
     const Ipp8u* pAnyOf, int lenAnyOf, int* pIndex);
IppStatus ippsFindCAny_16u(const Ipp16u* pSrc, int len,
     const Ipp16u* pAnyOf, int lenAnyOf, int* pIndex);
IppStatus ippsFindRevCAny_8u(const Ipp8u* pSrc, int len,
     const Ipp8u* pAnyOf, int lenAnyOf, int* pIndex);
IppStatus ippsFindRevCAny_16u(const Ipp16u* pSrc, int len,
     const Ipp16u* pAnyOf, int lenAnyOf, int* pIndex);
```

引数

pSrc	ソース文字列へのポインタ。
len	ソース文字列の要素数。
pAnyOf	参照要素が格納された配列へのポインタ。
lenAnyOf	配列内の要素の数。
pIndex	結果インデックスへのポインタ。

説明

関数 ippsFindCAny と ippsFindRevCAny は、ippch.h ファイルで宣言される。これらの関数は、指定された配列 *pAnyOf* に格納されたいずれかの参照要素に一致する最初の要素を、ソース文字列 *pSrc* から検索する。この要素の位置は *pIndex* に

格納される。一致する要素が見つからなかった場合、*pIndex* に -1 が設定される。
関数 ippsFindRevCAny は、ソース文字列を逆順で検索する。
検索では、大文字・小文字を区別する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* または *lenAnyOf* の値が負。

Insert

指定された文字列を他の文字列に挿入する。

```
IppStatus ippsInsert_8u(const Ipp8u* pSrc, int srcLen, const
      Ipp8u* pInsert, int insertLen, Ipp8u* pDst, int
      startIndex);
IppStatus ippsInsert_16u(const Ipp16u* pSrc, int srcLen, const
      Ipp16u* pInsert, int insertLen, Ipp16u* pDst, int
      startIndex);
IppStatus ippsInsert_8u_I(const Ipp8u* pInsert, int insertLen,
      Ipp8u* pSrcDst, int* pSrcDstLen, int startIndex);
IppStatus ippsInsert_16u_I(const Ipp16u* pInsert, int
      insertLen, Ipp16u* pSrcDst, int* pSrcDstLen, int
      startIndex);
```

引数

pSrc	ソース文字列へのポインタ。
srcLen	ソース文字列の要素数。
pInsert	挿入される文字列へのポインタ。
insertLen	挿入される文字列の要素数。
pDst	デスティネーション文字列へのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション文字列へのポインタ。
pSrcDstLen	インプレース演算用のソースとデスティネーション文字列の要素数へのポインタ。
startIndex	挿入位置のインデックス。

説明

関数 ippsInsert は、ippch.h ファイルで宣言される。この関数は、長さ *srcLen* のソース文字列 *pSrc* に、*insertLen* 個の要素を含む文字列 *pInsert* を挿入する。挿入位置は *startIndex* で指定される。結果は、*pDst* に格納される。

ippsInsert はインプレース演算で、ソース文字列 *pSrcDst* に文字列 *pInsert* を挿入し、その結果をデスティネーション文字列 *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*srcLen*、*insertLen*、*pSrcSdtLen*、*startIndex* のいずれかの値が負、または *startIndex* の値が *srcLen* か *pSrcDstLen* の値より大きい。

Remove

指定された数の要素を文字列から削除する。

```
IppStatus ippsRemove_8u(const Ipp8u* pSrc, int srcLen, Ipp8u*
      pDst, int startIndex, int len);
IppStatus ippsRemove_16u(const Ipp16u* pSrc, int srcLen,
      Ipp16u* pDst, int startIndex, int len);
IppStatus ippsRemove_8u_I(Ipp8u* pSrcDst, int* pSrcDstLen, int
      startIndex, int len);
IppStatus ippsRemove_16u_I(Ipp16u* pSrcDst, int* pSrcDstLen,
      int startIndex, int len);
```

引数

pSrc	ソース文字列へのポインタ。
srcLen	ソース文字列の要素数。
pDst	デスティネーション文字列へのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション文字列へのポインタ。

pSrcDstLen	インプレース演算用のソースとデスティネーション文字列の要素数へのポインタ。
startIndex	開始位置のインデックス。
len	削除する要素数。

説明

関数 ippsRemove は、ippch.h ファイルで宣言される。この関数は、長さ *srcLen* のソース文字列 *pSrc* から *len* 個の要素を削除する。開始位置は *startIndex* で指定される。結果は *pDst* に格納される。

ippsRemove はインプレース演算で、ソース文字列 *pSrcDst* から *len* 個の要素を削除し、その結果をデスティネーション文字列 *pSrcDst* に格納する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*srcLen*、*len*、*pSrcSdtLen*、*startIndex* のいずれかの値が負、または *(startIndex+len)* の値が *srcLen* か *pSrcDstLen* の値より大きい。

Compare

2 つの固定長文字列を比較する。

```
IppStatus ippsCompare_8u(const Ipp8u* pSrc1, const Ipp8u*
     pSrc2, int len, int *pResult);
IppStatus ippsCompare_16u(const Ipp16u* pSrc1, const Ipp16u*
     pSrc2, int len, int *pResult);
```

引数

pSrc1	先頭のソース文字列へのポインタ。
pSrc2	2 番目のソース文字列へのポインタ。
len	比較する要素の最大数。
pResult	結果へのポインタ。

説明

関数 ippsCompare は、ippch.h ファイルで宣言される。この関数は、2 つの文字列 *pSrc1* と *pSrc2* の最初の *len* 個の要素を比較する。
値 *pResult* = *pSrc1*[i]-*pSrc2*[i] では、i 番目の要素を 0 から *len*-1 までの i の値に対して逐次的に計算する。一致しない要素が発生した場合 (つまり、*pResult* がゼロではない場合)、関数は操作を中止し、*pResult* の値を返す。*pSrc1*[i]>*pSrc2*[i] の場合、正の値が返される。*pSrc1*[i]<*pSrc2*[i] の場合、負の値が返される。文字列が一致した場合、関数は *pResult* = 0 を返す。
比較では、大文字・小文字を区別する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* の値が負。

CompareIgnoreCase, CompareIgnoreCaseLatin

大文字・小文字を無視して 2 つの
固定長文字列を比較する。

```
IppStatus ippsCompareIgnoreCase_16u(const Ipp16u* pSrc1,
       const Ipp16u* pSrc2, int len, int* pResult);
IppStatus ippsCompareIgnoreCaseLatin_8u(const Ipp8u* pSrc1,
       const Ipp8u* pSrc2, int len, int* pResult);
IppStatus ippsCompareIgnoreCaseLatin_16u(const Ipp16u* pSrc1,
       const Ipp16u* pSrc2, int len, int* pResult);
```

引数

pSrc1	先頭のソース文字列へのポインタ。
pSrc2	2 番目のソース文字列へのポインタ。
len	比較する要素の最大数。
pResult	結果へのポインタ。

説明

関数 ippsCompareIgnoreCase と ippsCompareIgnoreCaseLatin は、ippch.h ファイルで宣言される。これらの関数は、2 つの文字列 *pSrc1* と *pSrc2* の最初の *len* 個の要素を比較する。すべての文字列が一致した場合、関数は *pResult* = 0 を返す。一致しない要素が i 番目で発生した場合、関数は操作を中止し、*pResult* を返す。*pSrc1*[i]>*pSrc2*[i] の場合、正の値が返される。*pSrc1*[i]<*pSrc2*[i] の場合、負の値が返される。
比較では、大文字・小文字を区別しない。

関数 ippsCompareIgnore は、Unicode 文字を比較する。
関数 ippsCompareIgnoreLatin は、ASCII 文字を比較する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* の値が負。

Equal

2 つの固定長文字列が等しいかを比較する。

```
IppStatus ippsEqual_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2,
      int len, int* pResult);
IppStatus ippsEqual_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
      int len, int* pResult);
```

引数

pSrc1	先頭のソース文字列へのポインタ。
pSrc2	2 番目のソース文字列へのポインタ。
len	比較する要素の最大数。
pResult	結果へのポインタ。

説明

関数 ippsEqual は、ippch.h ファイルで宣言される。この関数は、2 つの文字列 *pSrc1* と *pSrc2* の最初の *len* 個の要素を比較する。先頭の文字列の各要素は、2 番目の文字列で対応する要素と比較される。一致しない要素が発生した場合、関数は

操作を中止し、*pResult* に 0 を格納する。文字列が一致した場合、関数は *pResult* に 1 を格納する。
比較では、大文字・小文字を区別する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* の値が負。

TrimC

指定された文字を文字列の最初と最後からすべて削除する。

```
IppStatus ippsTrimC_8u(const Ipp8u* pSrc, int srcLen, Ipp8u odd,
      Ipp8u* pDst, int* pDstLen);
IppStatus ippsTrimC_16u(const Ipp16u* pSrc, int srcLen, Ipp16u odd,
      Ipp16u* pDst, int* pDstLen);
IppStatus ippsTrimC_8u_I(Ipp8u* pSrcDst, int* pLen, Ipp8u odd);
IppStatus ippsTrimC_16u_I(Ipp16u* pSrcDst, int* pLen, Ipp16u odd);
```

引数

pSrc	ソース文字列へのポインタ。
srcLen	ソース文字列の要素数。
pSrcDst	インプレース演算用のソースとデスティネーション文字列へのポインタ。
pDst	デスティネーション文字列へのポインタ。
pDstLen	デスティネーション文字列の要素数へのポインタ。
pLen	インプレース演算用のソースとデスティネーション文字列の要素数へのポインタ。
odd	削除する文字。

説明

関数 ippsTrimC は、ippch.h ファイルで定義される。この関数は、*srcLen* 個の要素を含むソース文字列 *pSrc* の最初と最後に、指定された文字 *odd* が見つかった場合、その文字をすべて削除する。関数は、*pDstLen* 個の要素を含む結果の文字列を *pDst* に格納する。

ippsTrimC はインプレース演算で、*pLen* 個の要素を含む文字列 *pSrcDst* の最初と最後に、指定された文字 *odd* が見つかった場合、その文字をすべて削除する。これらの関数は、*pLen* 個の要素を含む結果の文字列を *pSrcDst* に格納する。

この操作では、大文字・小文字を区別する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定 Š れたポインタが NULL。
ippStsLengthErr	エラー。*srcLen* または *pLen* の値が負。

TrimCAny
TrimStartCAny
TrimEndCAny

指定されたいずれかの文字に一致する要素をソース文字列の最初と最後からすべて削除する。

```
IppStatus ippsTrimCAny_8u(const Ipp8u* pSrc, int srcLen,
      const Ipp8u* pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);
IppStatus ippsTrimCAny_16u(const Ipp16u* pSrc, int srcLen,
      const Ipp16u* pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);

IppStatus ippsTrimStartCAny_8u(const Ipp8u* pSrc, int srcLen,
      const Ipp8u* pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);
IppStatus ippsTrimStartCAny_16u(const Ipp16u* pSrc, int srcLen,
      const Ipp16u* pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);

IppStatus ippsTrimEndCAny_8u(const Ipp8u* pSrc, int srcLen,
      const Ipp8u* pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);
```

```
IppStatus ippsTrimEndCAny_16u(const Ipp16u* pSrc, int srcLen,
      const Ipp16u* pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);
```

引数

pSrc	ソース文字列へのポインタ。
srcLen	ソース文字列の要素数。
pTrim	指定された要素が格納された配列へのポインタ。
trimLen	配列内の要素の数。
pDst	デスティネーション文字列へのポインタ。
pDstLen	デスティネーション文字列の要素数へのポインタ。

説明

関数 ippsTrimCAny、ippsTrimStartCAny、ippsTrimEndCAny は、ippch.h ファイルで宣言される。

関数 ippsTrimCAny は、配列 *pTrim* にある指定された要素のいずれかが、ソース文字列 *pSrc* の最初および最後に見つかった場合、その要素をすべて削除し、*pDstLen* 個の要素を含む結果の文字列を *pDst* に格納する。一致しない要素が発生した場合、関数は操作を中止する。

関数 ippsTrimStartCAny はソース文字列 *pSrc* の最初に、ippsTrimEndCAny は最後にこの操作を行う。

この操作では、大文字・小文字を区別する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*srcLen* または *trimLen* の値が負。

ReplaceC

ソース文字列の指定された要素を
他の要素に置換する。

```
IppStatus ippsReplaceC_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,
      Ipp8u oldVal, Ipp8u newVal);
```

```
IppStatus ippsReplaceC_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len,
      Ipp16u oldVal, Ipp16u newVal);
```

引数

pSrc	ソース文字列へのポインタ。
len	ソース文字列の要素数。
pDst	デスティネーション文字列へのポインタ。
oldVal	置換される要素。
newVal	*oldVal* を置換する要素。

説明

関数 ippsReplaceC は、ippch.h ファイルで宣言される。この関数はソース文字列 *pSrc* の指定された要素 *oldVal* を、指定された要素 *newVal* に置換し、新しい文字列を *pDst* に格納する。
この操作では、大文字・小文字を区別する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* の値が負。

Uppercase, UppercaseLatin

文字列のアルファベット文字を
すべて大文字に変換する。

```
IppStatus ippsUppercase_16u(const Ipp16u* pSrc, Ipp16u* pDst,
      int len);
IppStatus ippsUppercase_16u_I(Ipp16u* pSrcDst, int len);

IppStatus ippsUppercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst,
      int len);
IppStatus ippsUppercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst,
      int len);
IppStatus ippsUppercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
```

```
IppStatus ippsUppercaseLatin_16u_I(Ipp16u* pSrcDst, int len);
```

引数

pSrc	ソース文字列へのポインタ。
pDst	デスティネーション文字列へのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション文字列へのポインタ。
len	文字列の要素数。

説明

関数 ippsUppercase と ippsUppercaseLatin は、ippch.h ファイルで宣言される。これらの関数は、ソース文字列 *pSrc* に含まれたアルファベット文字を大文字に変換し、結果を *pDst* に格納する。

これらの関数はインプレース演算で、ソース文字列 *pSrcDst* に含まれたアルファベット文字を大文字に変換し、結果を *pSrcDst* に格納する。

関数 ippsUppercase は、Unicode 文字を変換する。
関数 ippsUppercaseLatin は、ASCII 文字を変換する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* の値が負。

Lowercase, LowercaseLatin

文字列のアルファベット文字を
すべて小文字に変換する。

```
IppStatus ippsLowercase_16u(const Ipp16u* pSrc, Ipp16u* pDst, int
      len);
IppStatus ippsLowercase_16u_I(Ipp16u* pSrcDst, int len);

IppStatus ippsLowercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst,
      int len);
```

```
IppStatus ippsLowercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst,
      int len);
IppStatus ippsLowercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsLowercaseLatin_16u_I(Ipp16u* pSrcDst, int len);
```

引数

pSrc	ソース文字列へのポインタ。
pDst	デスティネーション文字列へのポインタ。
pSrcDst	インプレース演算用のソースとデスティネーション文字列へのポインタ。
len	文字列の要素数。

説明

関数 ippsLowercase と ippsLowercaseLatin は、ippch.h ファイルで宣言される。これらの関数は、ソース文字列 *pSrc* に含まれたアルファベット文字を小文字に変換し、結果を *pDst* に格納する。

これらの関数はインプレース演算で、ソース文字列 *pSrcDst* に含まれたアルファベット文字を小文字に変換し、結果を *pSrcDst* に格納する。

関数 ippsLowercase は、Unicode 文字を変換する。
関数 ippsLowercaseLatin は、ASCII 文字を変換する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* の値が負。

Hash

文字列のハッシュ値を計算する。

```
IppStatus ippsHash_8u32u(const Ipp8u* pSrc, int len, Ipp32u*
      pHashVal);
IppStatus ippsHash_16u32u(const Ipp16u* pSrc, int len, Ipp32u*
      pHashVal);
```

引数

pSrc	ソース文字列へのポインタ。
len	文字列の要素数。
pHashVal	結果の値へのポインタ。

説明

関数 ippsHash は、ippch.h ファイルで宣言される。この関数は、指定された文字列 *pSrc* のハッシュ値 *pHasVal* を生成する。ハッシュ値は、各文字列に対して一意である。2 つの文字列のハッシュ値が異なる場合、文字列もまた異なり、ハッシュ値が同じ場合、文字列も同じである。ハッシュ値 *pHashVal*[i] = 2**pHashVal*[i-1]^*pSrc*[i] では、文字列の i 番目の要素を、初期値の 0 から i-1 までの i の値に対して逐次的に計算する。^ はビット単位 XOR (排他 OR) 演算子を意味する。最後の要素のハッシュ値は、文字列全体のハッシュ値を示す。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len* の値が負。

Concat

複数の文字列を連結する。

```
IppStatus ippsConcat_8u_D2L(const Ipp8u* const pSrc[],
     const int* pSrcLen[], int numSrc, Ipp8u* pDst);
IppStatus ippsConcat_16u_D2L(const Ipp16u* const pSrc[],
     const int* pSrcLen[], int numSrc, Ipp16u* pDst);
IppStatus ippsConcat_8u(const Ipp8u* pSrc1, int len1, const Ipp8u*
     pSrc2, int len2, Ipp8u* pDst);
IppStatus ippsConcat_16u(const Ipp16u* pSrc1, int len1, const Ipp16u*
     pSrc2, int len2, Ipp16u* pDst);
```

引数

pSrc1	先頭のソース文字列へのポインタ。
len1	先頭の文字列の要素数。

pSrc2	2 番目のソース文字列へのポインタ。
len2	2 番目の文字列の要素数。
pSrc	ソース文字列が格納された配列へのポインタ。
pSrcLen	ソース文字列の長さが格納された配列へのポインタ。
numSrc	ソース文字列の数。
pDst	デスティネーション文字列へのポインタ。

説明

関数 ippsConcat は、ippch.h ファイルで宣言される。この関数は、複数の文字列を連結する。この関数に D2L サフィックスを付けると、*numSrc* 個の文字列 *pSrc*[] を操作する。このサフィックスが付いていない場合、2 つの文字列 *pSrc1* と *pSrc2* のみを操作する。結果の文字列は *pDst* に格納される。デスティネーション文字列用のメモリ・ブロックは、関数が呼び出される前に割り当てる必要がある。連結する文字列の長さの合計は、IPP_MAX_32S 以下にする必要がある。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。*len1* か *len2* の値が負、または i < *numSrc* の場合に *srcLen*[i] の値が負。
ippStsSizeErr	エラー。*numSrc* の値が 0 以下。

ConcatC

複数の文字列を連結し、各文字列間に区切り文字を挿入する。

```
IppStatus ippsConcatC_8u_D2L(const Ipp8u* const pSrc[],
      const int* pSrcLen[], int numSrc, Ipp8u delim, Ipp8u* pDst);
IppStatus ippsConcatC_16u_D2L(const Ipp16u* const pSrc[],
      const int* pSrcLen[], int numSrc, Ipp16u delim, Ipp16u* pDst);
```

引数

pSrc	ソース文字列が格納された配列へのポインタ。
pSrcLen	ソース文字列の長さが格納された配列へのポインタ。

numSrc	ソース文字列の数。
delim	区切り文字。
pDst	デスティネーション文字列へのポインタ。

説明

関数 ippsConcatC は、ippch.h ファイルで宣言される。この関数は、*numSrc* 個の文字列 *pSrc* を連結し、指定された区切り文字 *delim* を、結果の文字列 *pDst* の各文字間に挿入する。

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。1 つ以上の指定されたポインタが NULL。
ippStsLengthErr	エラー。i < *numSrc* の時に *srcLen*[i] の値が負。
ippStsSizeErr	エラー。*numSrc* の値が 0 以下。

SplitC

ソース文字列を分割する。

```
IppStatus ippsSplitC_8u_D2L(const Ipp8u* pSrc, int srcLen, Ipp8u
     delim, Ipp8u* pDst[], int* pDstLen[], int* pNumDst);
IppStatus ippsSplitC_16u_D2L(const Ipp16u* pSrc, int srcLen, Ipp16u
     delim, Ipp16u* pDst[], int* pDstLen[], int* pNumDst);
```

引数

pSrc	ソース文字列へのポインタ。
srcLen	ソース文字列の要素数。
delim	区切り文字。
pDst	デスティネーション文字列が格納された配列へのポインタ。
pDstLen	デスティネーション文字列の長さが格納された配列へのポインタ。
pNumDst	デスティネーション文字列の数。

intel®

説明

関数 `ippsSplitC` は、`ippch.h` ファイルで宣言される。この関数は、区切り文字 *delim* を使用して、ソース文字列 *pSrc* を *nNumDst* 個の文字列 *pDst* に分割する。
指定された n 個の区切り文字がソース文字列の最初または最後にある場合、n 個の空文字列がデスティネーション文字列の配列に格納される。
指定された n 個の区切り文字がソース文字列の他の場所にある場合、(n-1) 個の空文字列がデスティネーション文字列の配列に格納される。n は、その位置で見つかった区切り文字の数を示す。

戻り値

`ippStsNoErr`	エラーなし。
`ippStsNullPtrErr`	エラー。1 つ以上の指定されたポインタが NULL。
`ippStsLengthErr`	エラー。i < *pNumDst* の場合に *srcLen* の値が負、または *dstLen* の値が負。
`ippStsSizeErr`	エラー。*pNumDst* の値が 0 以下。
`ippStsOvermatchStrings`	警告。出力文字列が、*pNumDst* で指定された数を超えている。この場合、奇数の文字列が破棄される。
`ippStsOverlongString`	警告。いくつかの出力文字数の要素数が、*pDstLen* で指定された値を超えている。この場合、対応する文字列は切り捨てられる。

固定精度算術関数 12

本章では、ベクトル引数を処理するインテル® IPP 固定精度超越数値演算関数について説明する。これらの関数は、入力ベクトルを引数として使用して、各基本関数の値を要素ごとに計算し、結果を出力ベクトルに返す。

関数の仕様は、インテル IPP の共通 API 規定に準拠しているが、科学算術関数に必要不可欠な新しい機能が含まれている。主な新機能は、共通の定義とは異なる詳細な精度の仕様である。単精度および倍精度データ形式に基づく元の精度レベル以外に、新しい精度レベルがいくつか追加されている。

固定精度ベクトル関数は、すべての関数型で IEEE-754 規格をサポートしている。これは次のことを意味する。

- すべての関数は、すべての引数値について、厳密に指定および保証される精度レベルを持つ。
- 特殊な値の処理および例外処理に関するすべての必要条件が満たされる。したがって、精度が標準レベルより低い場合、その関数は他のすべての点では IEEE-754 の必要条件に適合する。

精度のレベルは、実際の経験とアプリケーションの必要条件に基づいて選択する必要がある。利用可能な精度のオプションは、関数名のサフィックスで指定される。単精度データ形式では `A11`、`A21`、または `A24` を使用でき、倍精度データ形式では `A50` または `A53` を使用できる。関数型 `A11`、`A21`、`A50` は、それぞれおおよそ 3 桁、6 桁、15 桁の正確な 10 進数が保証される。関数型 `A24` および `A53` では、最大保証誤差は 1 ulp 以内で、通常は 0.55ulp を超えない。

インテル IPP の固定精度算術関数は、インテル® 数値演算ライブラリ (インテル® MKL) の対応する部分と同じ機能を持つ。

ただし、インテル IPP は、より低レベルの超越関数を提供する。これらの関数は、動作モードとデータ・タイプごとに別々の関数型を持ち、リアルタイム・アプリケーションのマルチメディアおよび信号処理に最適である。すべての固定精度算術関数を表 12-1 に示す。.

表 12-1 インテル® IPP 固定精度算術関数

関数の短縮名	データ・タイプ	説明
累乗関数と根関数		
Inv	32f, 64f	各ベクトル要素の逆数を計算する。
Div	32f, 64f	1 つのベクトル要素をそれに対応する別のベクトルの要素で割る。
Sqrt	32f, 64f	各ベクトル要素の平方根を計算する。
InvSqrt	32f, 64f	各ベクトル要素の逆平方根を計算する。
Cbrt	32f, 64f	各ベクトル要素の立方根を計算する。
InvCbrt	32f, 64f	各ベクトル要素の逆立方根を計算する。
Pow	32f, 64f	1 つのベクトルの各要素をもう 1 つのベクトルの対応する要素で累乗する。
Powx	32f, 64f	ベクトルの各要素を定数で累乗する。
指数関数と対数関数		
Exp	32f, 64f	*e* を各ベクトル要素で累乗する。
Ln	32f, 64f	各ベクトル要素の自然対数関数を計算する。
Log10	32f, 64f	各ベクトル要素の常用対数を計算する。
三角関数		
Cos	32f, 64f	各ベクトル要素の余弦を計算する。
Sin	32f, 64f	各ベクトル要素の正弦を計算する。
SinCos	32f, 64f	各ベクトル要素の正弦および余弦を計算する。
Tan	32f, 64f	各ベクトル要素の正接を計算する。
Acos	32f, 64f	各ベクトル要素の逆余弦を計算する。
Asin	32f, 64f	各ベクトル要素の逆正弦を計算する。
Atan	32f, 64f	各ベクトル要素の逆正接を計算する。
Atan2	32f, 64f	2 つのベクトルの要素の 4 象限逆正接を計算する。
双曲線関数		
Cosh	32f, 64f	各ベクトル要素の双曲線余弦を計算する。
Sinh	32f, 64f	各ベクトル要素の双曲線正弦を計算する。
Tanh	32f, 64f	各ベクトル要素の双曲線正接を計算する。
Acosh	32f, 64f	各ベクトル要素の逆 (負ではない) 双曲線余弦を計算する。
Asinh	32f, 64f	各ベクトル要素の逆双曲線正弦を計算する。
Atanh	32f, 64f	各ベクトル要素の逆双曲線正接を計算する。
特殊関数		
Erf	32f, 64f	誤差関数の値を計算する。
Erfc	32f, 64f	相補誤差関数の値を計算する。

注 : 本章で説明する固定精度算術関数と、第 5 章で説明した算術関数を混同してはならない。第 5 章の関数は、本章で説明する関数と同じ機能を持つが、異なる精度仕様に従う。

intel®

インテル IPP 固定精度算術関数は、正常な実行 (IppStsNoErr)、エラー状態 IppStsSizeErr、IppStsNullPtrErr (第 2 章の 表 2-2 を参照)、各種の警告 IppStsDomain、IppStsSingularity、IppStsOverflow、IppStsUnderflow を示すステータス・コードを返すことがある。警告の場合は、戻り値は正の値になり、計算は続行される。

表 12-2 は、警告を表すステータス・コードと対応するメッセージの一覧である。

表 12-2 固定精度算術関数の警告ステータス・コード

ステータス	値	メッセージ
IppStsOverflow	12	演算中にオーバーフローが発生した
IppStsUnderflow	17	演算中にアンダーフローが発生した
IppStsSingularity	18	演算中に特異点が発生した
IppStsDomain	19	引数が関数領域外

引数の値が関数の定義の範囲から外れている場合の関数の動作については、付録 A「特殊な事例の処理」を参照のこと。

累乗関数と根関数

Inv

各ベクトル要素の逆数を計算する。

```
IppStatus ippsInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsInv は、ippvm.h ファイルで宣言される。この関数は、ベクトル *pSrc* の各要素の 逆数 を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsInv_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsInv_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsInv_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsInv_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsInv_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst[n] = \frac{1}{pSrc[n]}, 0 \le n < len$$

例 12-1 は、関数 ippsInv の使用方法を示している。

例 12-1 ippsInv 関数の使用

```
IppStatus ippsInv_32f_A21_sample(void)
{
const Ipp32f x[4] = {-9.975, 1.272, -6.134, 6.175};
Ipp32f      y[4];

    IppStatus st = ippsInv_32f_A21( x, y, 4 );

    printf(" ippsInv_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsInv_32f_A21:
 x = -9.975 1.272 -6.134 6.175
 y = -0.100 0.786 -0.163 0.162
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsSingularity	警告。引数が特異点、すなわち、*pSrc* の少なくとも 1 つの要素がゼロ。

Div

1 番目のベクトルの要素をそれに対応する 2 番目のベクトルの要素で割る。

```
IppStatus ippsDiv_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsDiv_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f*  pDst, int len);
IppStatus ippsDiv_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f*  pDst, int len);
IppStatus ippsDiv_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsDiv_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
```

引数

pSrc1	1 番目のソース・ベクトルへのポインタ。
pSrc2	2 番目のソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsDiv は、ippvm.h ファイルで宣言される。この関数は、ベクトル *pSrc1* の各要素をそれに対応するベクトル *pSrc2* の要素で割り、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsDiv_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsDiv_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsDiv_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsDiv_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 `ippsDiv_64f_A53` は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。 $pDst[n] = \frac{pSrc1[n]}{pSrc2[n]}, 0 \le n < len$

例 12-2 は、関数 `ippsDiv` の使用方法を示している。

例 12-2 ippsDiv 関数の使用

```
IppStatus ippsDiv_32f_A21_sample(void)
{
const Ipp32f x1[4] = {599.088, 735.034, 572.448, 151.640};
const Ipp32f x2[4] = {385.297, 609.005, 361.403, 225.182};
Ipp32f       y[4];

    IppStatus st = ippsDiv_32f_A21( x1, x2, y, 4 );

    printf(" ippsDiv_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}

Output results:

 ippsDiv_32f_A21:
 x1 = 599.088 735.034 572.448 151.640
 x2 = 385.297 609.005 361.403 225.182
 y  = 1.555 1.207 1.584 0.673
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

IppStsSingularity	警告。引数が特異点、すなわち、*pSrc2* の少なくとも 1 つの要素がゼロ。

Sqrt

各ベクトル要素の平方根を計算する。

```
IppStatus ippsSqrt_32f_A11 ( const Ipp32f* pSrc, Ipp32f* pDst, int len );
IppStatus ippsSqrt_32f_A21 ( const Ipp32f* pSrc, Ipp32f* pDst, int len );
IppStatus ippsSqrt_32f_A24 ( const Ipp32f* pSrc, Ipp32f* pDst, int len );
IppStatus ippsSqrt_64f_A50 ( const Ipp64f* pSrc, Ipp64f* pDst, int len );
IppStatus ippsSqrt_64f_A53 ( const Ipp64f* pSrc, Ipp64f* pDst, int len );
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsSqrt は、ippvm.h ファイルで宣言される。この関数は、*pSrc* の各要素の平方根を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsSqrt_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsSqrt_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsSqrt_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsSqrt_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsSqrt_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = \sqrt{pSrc[n]}, 0 \le n < len$

例 12-3 は、関数 ippsSqrt の使用方法を示している

例 12-3 ippsSqrt 関数の使用

```
IppStatus ippsSqrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {5850.093, 4798.730, 3502.915, 8959.624};
Ipp32f      y[4];

    IppStatus st = ippsSqrt_32f_A21( x, y, 4 );

    printf(" ippsSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsSqrt_32f_A21:
 x = 5850.093 4798.730 3502.915 8959.624
 y = 76.486 69.273 59.185 94.655
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素がゼロより小さい。

InvSqrt

各ベクトル要素の逆平方根を計算する。

```
IppStatus ippsInvSqrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsInvSqrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvSqrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInvSqrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsInvSqrt` は、`ippvm.h` ファイルで宣言される。この関数は、*pSrc* の各要素の逆平方根を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsInvSqrt_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsInvSqrt_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsInvSqrt_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsInvSqrt_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 `ippsInvSqrt_64f_A53` は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst[n] = \frac{1}{\sqrt{pSrc[n]}}, 0 \le n < len$$

例 12-4 は、関数 `ippsInvSqrt` の使用方法を示している。

例 12-4 ippsInvSqrt 関数の使用

```
IppStatus ippsInvSqrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {7105.043, 5135.398, 3040.018, 149.944};
```

例 12-4 ippsInvSqrt 関数の使用 （続き）

```
Ipp32f      y[4];

    IppStatus st = ippsInvSqrt_32f_A21( x, y, 4 );

    printf(" ippsInvSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsInvSqrt_32f_A21:
 x = 7105.043 5135.398 3040.018 149.944
 y = 0.012 0.014 0.018 0.082
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素がゼロより小さい。
IppStsSingularity	警告。引数が特異点、すなわち、*pSrc* の少なくとも 1 つの要素がゼロ。

Cbrt

各ベクトル要素の立方根を計算する。

```
IppStatus ippsCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。

len ベクトル内の要素の数。

説明

関数 ippsCbrt は、ippvm.h ファイルで宣言される。この関数は、*pSrc* の各要素の立方根を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsCbrt_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsCbrt_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsCbrt_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsCbrt_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsCbrt_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst[n] = \sqrt[3]{pSrc[n]},\ 0 \le n < len$$

例 12-5 は、関数 ippsCbrt の使用方法を示している。

例 12-5 ippsCbrt 関数の使用

```
IppStatus ippsCbrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {6456.801, 4932.096, -6517.838, 7178.869};
Ipp32f      y[4];

    IppStatus st = ippsCbrt_32f_A21( x, y, 4 );

    printf(" ippsCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsCbrt_32f_A21:
 x = 6456.801 4932.096 -6517.838 7178.869
```

例 12-5　ippsCbrt 関数の使用 （続き）

```
y = 18.621 17.022 -18.680 19.291
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

InvCbrt

各ベクトル要素の逆立方根を計算する。

```
IppStatus ippsInvCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsInvCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsInvCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsInvCbrt は、ippvm.h ファイルで宣言される。この関数は、*pSrc* の各要素の逆立方根を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsInvCbrt_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsInvCbrt_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsInvCbrt_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsInvCbrt_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsInvCbrt_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst[n] = \frac{1}{\sqrt[3]{pSrc[n]}}, 0 \le n < len$$

例 12-6 は、関数 ippsInvCbrt の使用方法を示している。

例 12-6 ippsInvCbrt 関数の使用

```
IppStatus ippsInvCbrt_32f_A21_sample(void)
{
const Ipp32f x[4] = {914.120, 3644.584, 1473.214, 1659.070};
Ipp32f       y[4];

    IppStatus st = ippsInvCbrt_32f_A21( x, y, 4 );

    printf(" ippsInvCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsInvCbrt_32f_A21:
 x = 914.120 3644.584 1473.214 1659.070
 y = 0.103 0.065 0.088 0.084
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsSingularity	警告。引数が特異点、すなわち、*pSrc* の少なくとも 1 つの要素がゼロ。

Pow

1 番目のベクトルの各要素を 2 番目
ベクトルの対応する要素で累乗する。

```
IppStatus ippsPow_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
    Ipp32f* pDst, int len);
IppStatus ippsPow_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
IppStatus ippsPow_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2,
    Ipp64f* pDst, int len);
```

引数

pSrc1	1 番目のソース・ベクトルへのポインタ。
pSrc2	2 番目のソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsPow` は、`ippvm.h` ファイルで宣言される。この関数は、ベクトル *pSrc1* の各要素をベクトル *pSrc2* の対応する要素で累乗し、結果を *pDst* の対応する要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsPow_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsPow_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsPow_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsPow_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsPow_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = (pSrc1[n])^{pSrc2[n]}, 0 \le n < len$

例 12-7 は、関数 ippsPow の使用方法を示している。

例 12-7 ippsPow 関数の使用

```
IppStatus ippsPow_32f_A21_sample(void)
{
const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32f x2[4] = {0.823, 0.991, 0.411, 0.692};
Ipp32f       y[4];

    IppStatus st = ippsPow_32f_A21( x1, x2, y, 4 );

    printf(" ippsPow_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}


Output results:

 ippsPow_32f_A21:
 x1  = 0.483 0.565 0.776 0.252
 x2  = 0.823 0.991 0.411 0.692
 y   = 0.549 0.568 0.901 0.386
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、少なくとも 1 組のソース要素が次の条件を満たす。*pSrc1* の要素がゼロより小さい有限数であり、*pSrc2* の要素が整数でない有限数である。

`IppStsSingularity`	警告。引数が特異点、すなわち、少なくとも 1 組の要素が次の条件を満たす。*pSrc1* の要素がゼロであり、*pSrc2* の要素がゼロより小さい整数である。

Powx

ベクトルの各要素を定数で累乗する。

```
IppStatus ippsPowx_32f_A11 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
     Ipp32f* pDst, int len);
IppStatus ippsPowx_32f_A21 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
     Ipp32f* pDst, int len);
IppStatus ippsPowx_32f_A24 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
     Ipp32f* pDst, int len);
IppStatus ippsPowx_64f_A50 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
     Ipp64f* pDst, int len);
IppStatus ippsPowx_64f_A53 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
     Ipp64f* pDst, int len);
```

引数

pSrc1	ソース・ベクトルへのポインタ。
ConstValue	定数値。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsPowx` は、`ippvm.h` ファイルで宣言される。この関数は、ベクトル *pSrc1* の各要素を定数 *ConstValue* で累乗し、結果を *pDst* の対応する要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsPowx_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsPowx_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsPowx_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsPowx_64f_A50は、正しく丸められた50ビットの仮数、または4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsPowx_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst[n] = (pSrc1[n])^{ConstValue}, 0 \le n < len$$

例 12-8 は、関数 ippsPowx の使用方法を示している。

例 12-8 ippsPowx 関数の使用

```
IppStatus ippsPowx_32f_A21_sample(void)
{
const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
const Ipp32f x2 = 0.823;
Ipp32f       y[4];

    IppStatus st = ippsPowx_32f_A21( x1, x2, y, 4 );

    printf(" ippsPowx_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f \n", x2);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}


Output results:

 ippsPowx_32f_A21:
 x1 = 0.483 0.565 0.776 0.252
 x2 = 0.823
 y  = 0.549 0.568 0.901 0.386
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

ippStsDomain	警告。引数が関数領域外、すなわち、少なくとも 1 組のソース要素が次の条件を満たす。*pSrc1* の要素がゼロより小さい有限数であり、*ConstValue* が整数でない有限数である。
IppStsSingularity	警告。引数が特異点、すなわち、少なくとも 1 組の要素が次の条件を満たす。*pSrc1* の要素がゼロであり、*ConstValue* がゼロより小さい整数である。

指数関数と対数関数

Exp

e を各ベクトル要素で累乗する。

```
IppStatus ippsExp_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExp_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExp_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsExp は、ippvm.h ファイルで宣言される。この関数は、*e* を *pSrc* の各要素で累乗し、結果を *pDst* の対応する要素に格納する。
単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsExp_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsExp_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsExp_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsExp_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsExp_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = e^{pSrc[n]}, 0 \leq n < len$

例 12-9 は、関数 ippsExp の使用方法を示している。

例 12-9 ippsExp 関数の使用

```
IppStatus ippsExp_32f_A21_sample(void)
{
const Ipp32f x[4] = {4.885, -0.543, -3.809, -4.953};
Ipp32f      y[4];

    IppStatus st = ippsExp_32f_A21( x, y, 4 );

    printf(" ippsExp_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsExp_32f_A21:
 x = 4.885 -0.543 -3.809 -4.953
 y = 132.324 0.581 0.022 0.007
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsOverflow	警告。関数のオーバーフロー、すなわち、*pSrc* の少なくとも 1 つの要素が Ln (FPMAX) より大きい。FPMAX は、表現可能な浮動小数点数の最大値である。

`IppStsUnderflow`	警告。関数のアンダーフロー、すなわち、*pSrc* の少なくとも 1 つの要素が `Ln (FPMIN)` より小さい。`FPMIN` は、正の浮動小数点数の最小値である。

Ln

各ベクトル要素の自然対数関数を計算する。

```
IppStatus ippsLn_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsLn` は、`ippvm.h` ファイルで宣言される。この関数は *pSrc* の各要素の自然対数を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsLn_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsLn_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsLn_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsLn_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsLn_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = \log_e(pSrc[n]),\ 0 \le n < len$

例 12-10 は、関数 ippsLn の使用方法を示している。

例 12-10 ippsLn 関数の使用

```
IppStatus ippsLn_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.188, 3.841, 5.363, 5.755};
Ipp32f       y[4];

    IppStatus st = ippsLn_32f_A21( x, y, 4 );

    printf(" ippsLn_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsLn_32f_A21:
 x = 0.188 3.841 5.363 5.755
 y = -1.670 1.346 1.680 1.750
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素がゼロより小さい。
IppStsSingularity	警告。引数が特異点、すなわち、*pSrc* の少なくとも 1 つの要素がゼロ。

Log10

各ベクトル要素の常用対数を計算する。

```
IppStatus ippsLog10_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLog10_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLog10_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsLog10` は、`ippvm.h` ファイルで宣言される。この関数は *pSrc* の各要素の自然対数を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsLog10_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsLog10_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsLog10_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsLog10_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 `ippsLog10_64f_A53` は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = \log_{10}(pSrc[n]),\ 0 \le n < len$

例 12-11 は、関数 ippsLog10 の使用方法を示している。

例 12-11 ippsLog10 関数の使用

```
IppStatus ippsLog10_32f_A21_sample(void)
{
const Ipp32f x[4] = {6.057, 6.111, 1.746, 6.664};
Ipp32f      y[4];

    IppStatus st = ippsLog10_32f_A21( x, y, 4 );

    printf(" ippsLog10_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsLog10_32f_A21:
 x = 6.057 6.111 1.746 6.664
 y = 0.782 0.786 0.242 0.824
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素がゼロより小さい。
IppStsSingularity	警告。引数が特異点、すなわち、*pSrc* の少なくとも 1 つの要素がゼロ。

三角関数

Cos

各ベクトル要素の余弦を計算する。

```
IppStatus ippsCos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsCos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsCos` は、`ippvm.h` ファイルで宣言される。この関数は *pSrc* の各要素の余弦を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsCos_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsCos_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsCos_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsCos_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 `ippsCos_64f_A53` は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst[n] = \cos(pSrc[n]), 0 \le n < len$$

例 12-12 は、関数 `ippsCos` の使用方法を示している。

例 12-12 ippsCos 関数の使用

```
IppStatus ippsCos_32f_A21_sample(void)
{
const Ipp32f x[4] = {-984.222, -2957.549, -8859.218, 2153.691};
Ipp32f      y[4];
```

例 12-12 ippsCos 関数の使用 （続き）

```
    IppStatus st = ippsCos_32f_A21( x, y, 4 );

    printf(" ippsCos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsCos_32f_A21:
 x = -984.222 -2957.549 -8859.218 2153.691
 y = -0.619 -0.258 0.997 0.129
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素が ± INF。

Sin

各ベクトル要素の正弦を計算する。

```
IppStatus ippsSin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsSin は、ippvm.h ファイルで宣言される。この関数は *pSrc* の各要素の正弦を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsSin_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsSin_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsSin_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsSin_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsSin_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = \sin(pSrc[n])\ ,\ 0 \leq n < len$

例 12-13 は、関数 ippsSin の使用方法を示している。

例 12-13 ippsSin 関数の使用

```
IppStatus ippsSin_32f_A21_sample(void)
{
const Ipp32f x[4] = {5666.372, 6052.125, 397.656, -3960.997};
Ipp32f       y[4];

    IppStatus st = ippsSin_32f_A21( x, y, 4 );

    printf(" ippsSin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsSin_32f_A21:
 x = 5666.372 6052.125 397.656 -3960.997
 y = -0.873 0.988 0.970 -0.524
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素が ± INF。

SinCos

各ベクトル要素の正弦および余弦を計算する。

```
IppStatus ippsSinCos_32f_A11 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_32f_A21 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_32f_A24 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A50 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);
IppStatus ippsSinCos_64f_A53 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
    pDst2, int len);
```

引数

pSrc	1 番目のソース・ベクトルへのポインタ。
pDst1	正弦値用のデスティネーション・ベクトルへのポインタ。
pDst2	余弦値用のデスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsSinCos は、ippvm.h ファイルで宣言される。この関数は、*pSrc* の各要素の正弦を計算し、結果をそれに対応する *pDst1* の要素に格納する。*pSrc* の各要素の余弦を計算し、結果をそれに対応する *pDst2* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsSinCos_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsSinCos_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsSinCos_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsSinCos_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsSinCos_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst1[n] = \sin(pSrc[n]) , pDst2[n] = \cos(pSrc[n]) , 0 \le n < len$$

例 12-14 は、関数 ippsSinCos の使用方法を示している。

例 12-14 ippsSinCos 関数の使用

```
IppStatus ippsSinCos_32f_A21_sample(void)
{
const Ipp32f x[4] = {3857.845, -3939.024, -1468.856, -8592.486};
Ipp32f      y1[4];
Ipp32f      y2[4];

    IppStatus st = ippsSinCos_32f_A21( x, y1, y2, 4 );

    printf(" ippsSinCos_32f_A21:\n");
    printf(" x  = %.3f %.3f %.3f %.3f \n", x[0],  x[1],  x[2],  x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
    return st;
}

Output results:

 ippsSinCos_32f_A21:
 x  = 3857.845 -3939.024 -1468.856 -8592.486
 y1 = -0.031 0.508 0.987 0.228
 y2 = 1.000 0.861 0.161 -0.974
```

戻り値

ippStsNoErr	エラーなし。

ippStsNullPtrErr	エラー。ポインタ *pDst1*、*pDst2*、または *pSrc* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素が ± INF。

Tan

各ベクトル要素の正接を計算する。

```
IppStatus ippsTan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsTan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsTan は、ippvm.h ファイルで宣言される。この関数は *pSrc* の各要素の正接を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsTan_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsTan_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsTan_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsTan_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsTan_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。*pDst*[*n*] = tan(*pSrc*[*n*]) , 0 ≤ *n* < *len*

例 12-15 は、関数 ippsTan の使用方法を示している。

例 12-15 ippsTan 関数の使用

```
IppStatus ippsTan_32f_A21_sample(void)
{
const Ipp32f x[4] = {7519.456, 4533.524, 9118.015, 8514.359};
Ipp32f       y[4];

    IppStatus st = ippsTan_32f_A21( x, y, 4 );

    printf(" ippsTan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsTan_32f_A21:
 x = 7519.456 4533.524 9118.015 8514.359
 y = -18.656 0.209 2.028 0.750
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素が ± INF。

Acos

各ベクトル要素の逆余弦を計算する。

```
IppStatus ippsAcos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsAcos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsAcos は、ippvm.h ファイルで宣言される。この関数は、*pSrc* の各要素の逆余弦を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAcos_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsAcos_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsAcos_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAcos_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsAcos_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = \mathrm{acos}(pSrc[n])\ ,\ 0 \le n < len$

例 12-16 は、関数 ippsAcos の使用方法を示している。

例 12-16 ippsAcos 関数の使用

```
IppStatus ippsAcos_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.079, -0.715, -0.076, -0.529};
Ipp32f       y[4];
```

例 12-16 ippsAcos 関数の使用 （続き）

```
    IppStatus st = ippsAcos_32f_A21( x, y, 4 );

    printf(" ippsAcos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAcos_32f_A21:
 x = 0.079 -0.715 -0.076 -0.529
 y = 1.492 2.368 1.647 2.129
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素の絶対値が 1 より大きい。

Asin

各ベクトル要素の逆正弦を計算する。

```
IppStatus ippsAsin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsAsin` は、`ippvm.h` ファイルで宣言される。この関数は *pSrc* の各要素の逆正弦を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsAsin_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsAsin_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsAsin_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsAsin_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 `ippsAsin_64f_A53` は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。$pDst[n] = \mathrm{asin}(pSrc[n])$, $0 \le n < len$

例 12-17 は、関数 `ippsAsin` の使用方法を示している。

例 12-17 ippsAsin 関数の使用

```
IppStatus ippsAsin_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.724, -0.581, 0.559, 0.687};
Ipp32f      y[4];

    IppStatus st = ippsAsin_32f_A21( x, y, 4 );

    printf(" ippsAsin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsAsin_32f_A21:
 x = 0.724 -0.581 0.559 0.687
 y = 0.810 -0.620 0.594 0.758
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素の絶対値が 1 より大きい。

Atan

各ベクトル要素の逆正接を計算する。

```
IppStatus ippsAtan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAtan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsAtan は、ippvm.h ファイルで宣言される。この関数は *pSrc* の各要素の逆正接を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAtan_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsAtan_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsAtan_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAtan_64f_A50は、正しく丸められた50ビットの仮数、または4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsAtan_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

pDst[*n*] = atan(*pSrc*[*n*]) , 0 ≤ *n* < *len* .

例 12-18 は、関数 ippsAtan の使用方法を示している。

例 12-18　ippsAtan 関数の使用

```
IppStatus ippsAtan_32f_A21_sample(void)
{
const Ipp32f x[4] = {0.994, 0.999, 0.223, -0.215};
Ipp32f      y[4];

    IppStatus st = ippsAtan_32f_A21( x, y, 4 );

    printf(" ippsAtan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAtan_32f_A21:
 x = 0.994 0.999 0.223 -0.215
 y = 0.782 0.785 0.219 -0.212
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Atan2

2 つのベクトルの要素の 4 象限逆正接を計算する。

```
IppStatus ippsAtan2_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
     pDst, int len);
IppStatus ippsAtan2_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
     pDst, int len);
IppStatus ippsAtan2_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
     pDst, int len);
IppStatus ippsAtan2_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
     pDst, int len);
IppStatus ippsAtan2_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
     pDst, int len);
```

引数

pSrc1	1 番目のソース・ベクトルへのポインタ。
pSrc2	2 番目のソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsAtan2` は、`ippvm.h` ファイルで宣言される。この関数は、*pSrc1* の各要素を Y (縦座標)、*pSrc2* の対応する要素を X (横座標) に使用して、原点から点 (X,Y) までの直線と X 軸の角度を計算し、結果を *pDst* の対応する要素に格納する。結果の角度値の範囲は、-π ～ +π である。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsAtan2_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsAtan2_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsAtan2_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAtan2_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsAtan2_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = arc\tan2(pSrc1[n], pSrc2[n]), 0 \le n < len$

例 12-19 は、関数 ippsAtan2 の使用方法を示している。

例 12-19 ippsAtan2 関数の使用

```
IppStatus ippsAtan2_32f_A21_sample(void)
{
const Ipp32f x1[4] = {1.492, 1.700, 1.147, 1.142};
const Ipp32f x2[4] = {1.064, 1.505, 1.950, 1.905};
Ipp32f       y[4];

    IppStatus st = ippsAtan2_32f_A21( x1, x2, y, 4 );

    printf(" ippsAtan2_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}


Output results:

 ippsAtan2_32f_A21:
 x1 = 1.492 1.700 1.147 1.142
 x2 = 1.064 1.505 1.950 1.905
 y  = 0.951 0.846 0.532 0.540
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc1*、*pSrc2*、または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

双曲線関数

Cosh

各ベクトル要素の双曲線余弦を計算する。

```
IppStatus ippsCosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsCosh` は、`ippvm.h` ファイルで宣言される。この関数は、*pSrc* の各要素の双曲線余弦を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsCosh_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsCosh_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsCosh_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsCosh_64f_A50は、正しく丸められた50ビットの仮数、または4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsCosh_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = \cosh(pSrc[n]), 0 \leq n < len$

例 12-20 は、関数 ippsCosh の使用方法を示している。

例 12-20 ippsCosh 関数の使用

```
IppStatus ippsCosh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-4.676, -4.054, 6.803, -9.525};
Ipp32f      y[4];

    IppStatus st = ippsCosh_32f_A21( x, y, 4 );

    printf(" ippsCosh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsCosh_32f_A21:
 x = -4.676 -4.054 6.803 -9.525
 y = 53.661 28.833 450.219 6849.870
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsOverflow	警告。関数のオーバーフロー、すなわち、*pSrc* の少なくとも 1 つの要素の絶対値が Ln(FPMAX) + Ln(2) より大きい。FPMAX は、表現可能な浮動小数点数の最大値である。

Sinh

各ベクトル要素の双曲線正弦を計算する。

```
IppStatus ippsSinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsSinh` は、`ippvm.h` ファイルで宣言される。この関数は、*pSrc* の各要素の双曲線正弦を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsSinh_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsSinh_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsSinh_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsSinh_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 `ippsSinh_64f_A53` は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = \sinh(pSrc[n])$, $0 \le n < len$

intel®

例 12-21 は、関数 ippsSinh の使用方法を示している。

例 12-21 ippsSinh 関数の使用

```
IppStatus ippsSinh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-2.483, -8.148, 3.544, -8.876};
Ipp32f      y[4];

    IppStatus st = ippsSinh_32f_A21( x, y, 4 );

    printf(" ippsSinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsSinh_32f_A21:
 x = -2.483 -8.148 3.544 -8.876
 y = -5.945 -1727.412 17.290 -3577.970
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsOverflow	警告。関数のオーバーフロー、すなわち、*pSrc* の少なくとも 1 つの要素の絶対値が Ln(FPMAX) + Ln(2) より大きい。FPMAX は、表現可能な浮動小数点数の最大値である。

Tanh

各ベクトル要素の双曲線正接を計算する。

```
IppStatus ippsTanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsTanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsTanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsTanh` は、`ippvm.h` ファイルで宣言される。この関数は、*pSrc* の各要素の双曲線正接を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsTanh_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsTanh_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsTanh_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsTanh_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 `ippsTanh_64f_A53` は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[x] = \tanh(pSrc[n])\ ,\ 0 \le n < len$

例 12-22 は、関数 `ippsTanh` の使用方法を示している。

例 12-22 ippsTanh 関数の使用

```
IppStatus ippsTanh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f       y[4];

    IppStatus st = ippsTanh_32f_A21( x, y, 4 );

    printf(" ippsTanh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
```

例 12-22 ippsTanh 関数の使用 (続き)

```
}

Output results:

 ippsTanh_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Acosh

各ベクトル要素の逆 (負ではない) 双曲線余弦を計算する。

```
IppStatus ippsAcosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAcosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAcosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsAcosh は、ippvm.h ファイルで宣言される。この関数は、*pSrc* の各要素の逆 (負ではない) 双曲線余弦 を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAcosh_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsAcosh_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsAcosh_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAcosh_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsAcosh_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst[x] = \mathrm{acosh}(pSrc[n]),\ 0 \le n < len$$

例 12-23 は、関数 ippsAcosh の使用方法を示している。

例 12-23 ippsAcosh 関数の使用

```
IppStatus ippsAcosh_32f_A21_sample(void)
{
const Ipp32f x[4] = {588.321, 691.492, 837.773, 726.767};
Ipp32f      y[4];

    IppStatus st = ippsAcosh_32f_A21( x, y, 4 );

    printf(" ippsAcosh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAcosh_32f_A21:
 x = 588.321 691.492 837.773 726.767
 y = 7.070 7.232 7.424 7.282
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。

ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素が 1 より小さい。

Asinh

各ベクトル要素の逆双曲線正弦を計算する。

```
IppStatus ippsAsinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAsinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAsinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsAsinh は、ippvm.h ファイルで宣言される。この関数は、*pSrc* の各要素の逆双曲線正弦を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAsinh_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsAsinh_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsAsinh_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsAsinh_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 `ippsAsinh_64f_A53` は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = \mathrm{asinh}(pSrc[n])$, $0 \leq n < len$

例 12-24 は、関数 `ippsAsinh` の使用方法を示している。

例 12-24 ippsAsinh 関数の使用

```
IppStatus ippsAsinh_32f_A21_sample(void)
{
const Ipp32f x[4] = {-30.122, -589.282, 487.472, -63.082};
Ipp32f      y[4];

    IppStatus st = ippsAsinh_32f_A21( x, y, 4 );

    printf(" ippsAsinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAsinh_32f_A21:
 x = -30.122 -589.282 487.472 -63.082
 y = -4.099 -7.072 6.882 -4.838
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Atanh

各ベクトル要素の逆双曲線正接を計算する。

```
IppStatus ippsAtanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
```


intel®

```
IppStatus ippsAtanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAtanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAtanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsAtanh は、ippvm.h ファイルで宣言される。この関数は、*pSrc* の各要素の逆双曲線正接を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAtanh_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsAtanh_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsAtanh_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

関数型 ippsAtanh_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsAtanh_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$pDst[n] = \mathrm{atanh}(pSrc[n])\ ,\ 0 \le n < len$

例 12-25 は、関数 ippsAtanh の使用方法を示している。

例 12-25 ippsAtanh 関数の使用

```
IppStatus ippsAtanh_32f_A21_sample(void)
{
```

例 12-25　ippsAtanh 関数の使用 （続き）

```
const Ipp32f x[4] = {-0.076, 0.808, 0.440, -0.705};
Ipp32f      y[4];

    IppStatus st = ippsAtanh_32f_A21( x, y, 4 );

    printf(" ippsAtanh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

Output results:

 ippsAtanh_32f_A21:
 x = -0.076 0.808 0.440 -0.705
 y = -0.076 1.123 0.472 -0.877
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsDomain	警告。引数が関数領域外、すなわち、*pSrc* の少なくとも 1 つの要素の絶対値が 1 より大きい。
IppStsSingularity	警告。引数が特異点、すなわち、*pSrc* の少なくとも 1 つの要素の絶対値が 1。

特殊関数

Erf

誤差関数の値を計算する。

```
IppStatus ippsErf_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErf_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErf_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErf_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsErf_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 `ippsErf` は、`ippvm.h` ファイルで宣言される。この関数は、*pSrc* の各要素の誤差関数値を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 `ippsErf_32f_A11` は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 `ippsErf_32f_A21` は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 `ippsErf_32f_A24` は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

`ippsErf_64f_A50` は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 `ippsErf_64f_A53` は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst[n] = \mathrm{erf}(pSrc[n]),\ 0 \le n < len,\quad \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

例 12-26 は、関数 `ippsErf` の使用方法を示している。

例 12-26 ippsErf 関数の使用

```
IppStatus ippsErf_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f       y[4];

    IppStatus st = ippsErf_32f_A21( x, y, 4 );

    printf(" ippsErf_32f_A21:\n");
```

例 12-26 ippsErf 関数の使用 （続き）

```
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsErf_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。

Erfc

相補誤差関数の値を計算する。

```
IppStatus ippsErfc_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsErfc_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsErfc_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, int len);
```

引数

pSrc	ソース・ベクトルへのポインタ。
pDst	デスティネーション・ベクトルへのポインタ。
len	ベクトル内の要素の数。

説明

関数 ippsErfc は、ippvm.h ファイルで宣言される。この関数は、*pSrc* の各要素の相補誤差関数値を計算し、結果をそれに対応する *pDst* の要素に格納する。

単精度データの場合、各関数型の精度は次のようになる。

関数型 ippsErfc_32f_A11 は、正しく丸められた 11 ビットの仮数、または少なくとも 3 桁の正確な 10 進数を保証する。

関数型 ippsErfc_32f_A21 は、正しく丸められた 21 ビットの仮数、または 4 ulp、あるいはおおよそ 6 桁の正確な 10 進数を保証する。

関数型 ippsErfc_32f_A24 は、最大保証誤差 1 ulp 以内で、正しく丸められた 24 ビットの仮数 (暗黙ビットを含む) を保証する。

倍精度データの場合、各関数型の精度は次のようになる。

ippsErfc_64f_A50 は、正しく丸められた 50 ビットの仮数、または 4 ulp、あるいはおおよそ 15 桁の正確な 10 進数を保証する。

関数型 ippsErfc_64f_A53 は、最大保証誤差 1 ulp 以内で、正しく丸められた 53 ビットの仮数 (暗黙ビットを含む) を保証する。

計算は、次のように行われる。

$$pDst[n] = \mathrm{erfc}(pSrc[n]),\ 0 \le n < len,\quad \mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt\ .$$

例 12-27 は、関数 ippsErfc の使用方法を示している。

例 12-27 ippsErfc 関数の使用

```
IppStatus ippsErfc_32f_A21_sample(void)
{
const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
Ipp32f      y[4];

    IppStatus st = ippsErfc_32f_A21( x, y, 4 );

    printf(" ippsErfc_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}


Output results:

 ippsErfc_32f_A21:
 x = -0.982 0.838 -0.448 -0.454
 y = -0.754 0.685 -0.420 -0.425
```

戻り値

ippStsNoErr	エラーなし。
ippStsNullPtrErr	エラー。ポインタ *pSrc* または *pDst* が NULL。
ippStsSizeErr	エラー。*len* がゼロ以下。
IppStsUnderflow	警告。関数のアンダーフロー、すなわち、*pSrc* 要素の少なくとも 1 つが一部のしきい値より小さい。関数結果はターゲット精度では正の浮動小数点数の最小値より小さい。

特殊な事例の処理

インテル® IPP に実装されている数値演算関数の中には、すべての引数について定義されていないものがある。ここでは、入力引数が関数定義の範囲を外れているとき、あるいはあいまいな出力結果をまねく可能性があるときに、信号処理データ領域で使用されるインテル IPP 画像処理関数がその状況をどのように処理するかについて説明する。

以下の 表 A-1 に、第 5 章で説明されている汎用ベクトル関数について特殊な事例をまとめ、その関数から返ってくるステータス・コードと結果値の一覧を示す。

表 A-1 インテル® IPP 信号処理関数についての特殊な事例

関数の基本名	データ・タイプ	特殊な事例	結果値	ステータス・コード
ippsSqrt	16s	Sqrt (x <0)	0	ippStsSqrtNegArg
	32f	Sqrt (x <0)	NAN_32F	ippStsSqrtNegArg
	64s	Sqrt (x <0)	0	ippStsSqrtNegArg
	64f	Sqrt (x <0)	NAN_64F	ippStsSqrtNegArg
ippsDiv	8u	Div (0/0)	0	ippStsDivByZero
		Div (x/0)	IPP_MAX_8U	ippStsDivByZero
	16s	Div (0/0)	0	ippStsDivByZero
		Div (x/0), x>0	IPP_MAX_16S	ippStsDivByZero
		Div (x/0), x<0	IPP_MIN_16S	ippStsDivByZero
	16sc	Div (0/0)	0	ippStsDivByZero
		Div (x/0)	0	ippStsDivByZero
ippsDiv	32f	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0), x>0	INF_32F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_32F	ippStsDivByZero
	32fc	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0)	NAN_32F	ippStsDivByZero
				ippStsDivByZero
	64f	Div (0/0)	NAN_64F	
		Div (x/0), x>0	INF_64F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_64F	ippStsDivByZero
	64fc	Div (0/0)	NAN_64F	ippStsDivByZero
		Div (x/0)	NAN_64F	ippStsDivByZero
ippsDivC	All	Div(x/0)	x	ippStsDivByZeroErr

表 A-1 インテル® IPP 信号処理関数についての特殊な事例 （続き）

関数の基本名	データ・タイプ	特殊な事例	結果値	ステータス・コード
ippsLn	16s, 32s	Ln (0) Ln (x<0)	0 0	ippStsLnZeroArg ippStsLnNegArg
	32f	Ln (x<0) Ln(x<IPP_MINABS_32F)	NAN_32F INF_NEG_32F	ippStsLnNegArg ippStsLnZeroArg
	64f	 Ln (x<0) Ln(x<IPP_MINABS_64F	NAN_64F INF_NEG_64F	ippStsLnNegArg ippStsLnZeroArg
ippsExp	16s 32s 64s 32f 64f	overflow overflow overflow overflow overflow	IPP_MAX_16S IPP_MAX_32S IPP_MAX_64S INF_32F INF_64F	ippStsNoErr ippStsNoErr ippStsNoErr ippStsNoErr ippStsNoErr
ippsThreshold, ippsThresholdLT, ippsThresholdGT, ippsThresholdLTVal, ippsThresholdGTVal	16sc 32fc 64fc	level<0 level<0 level<0	x x x	ippStsThreshNegLevelErr ippStsThreshNegLevelErr ippStsThreshNegLevelErr
ippsThresholdLTInv	32f	level<0 level=0, x=0	x INF_32F	ippStsThreshNegLevelErr ippStsInvZero
	32fc	level<0 level=0, x=0	x INF_32F	ippStsThreshNegLevelErr ippStsInvZero
	64f	level<0 level=0, x=0	x INF_64F	ippStsThreshNegLevelErr ippStsInvZero
	64fc	level<0 level=0, x=0	x INF_64F	ippStsThreshNegLevelErr ippStsInvZero

ここで、x は入力値を示す。それぞれの定数の定義については、第 2 章の データ範囲 を参照のこと。

別々のデータ・タイプを、同じ数値演算関数のいくつかの変種で処理すると、引数値が同じでも、異なる結果の出る場合がある。ただし、特定の関数と固定データ・タイプについて言えば、特殊な事例の処理は、さまざまな記述子をその名前に含んでいる関数のどの変種でも同じである。例えば、16s データを操作する対数関数 ippsLn におけるゼロ引数値の扱いは、そのすべての変種 ippsLn_16s_Sfs、ippsLn_16s_ISfs で同じである。

以下の表 A-2 に、第 12 章の「固定精度算術関数」で説明した固定精度算術ベクトル関数について特殊な事例をまとめている。

intel®

表 A-2 インテル® IPP 固定精度数値演算関数の特殊な事例

関数の基本名	データ・タイプ	特殊な事例	結果値	ステータス・コード
ippsInv	32f	Inv (x =+0)	INF_32F	ippStsSingularity
		Inv (x =-0)	-INF_32F	ippStsSingularity
	64f	Inv (x =+0)	INF_64F	ippStsSingularity
		Inv (x =-0)	-INF_64F	ippStsSingularity
ippsDiv	32f	Div (x>0, y=+0)	INF_32F	ippStsSingularity
		Div (x>0, y=-0)	-INF_32F	ippStsSingularity
		DIv (x<0, y=+0)	-INF_32F	ippStsSingularity
		Div (x<0, y=-0)	INF_32F	ippStsSingularity
		Div (x=0, y=0)	NAN_32F	ippStsSingularity
	64f	Div (x>0, y=+0)	INF_64F	ippStsSingularity
		Div (x>0, y=-0)	-INF_64F	ippStsSingularity
		DIv (x<0, y=+0)	-INF_64F	ippStsSingularity
		Div (x<0, y=-0)	INF_64F	ippStsSingularity
		Div (x=0, y=0)	NAN_64F	ippStsSingularity
ippsSqrt	32f	Sqrt (x<0)	NAN_32F	ippStsDomain
		Sqrt (x=-INF)	NAN_32F	ippStsDomain
	64f	Sqrt (x<0)	NAN_64F	ippStsDomain
		Sqrt (x=-INF)	NAN_64F	ippStsDomain
ippsInvSqrt	32f	InvSqrt (x<0)	NAN_32F	ippStsDomain
		InvSqrt (x=+0)	INF_32F	ippStsSingularity
		InvSqrt (x=-0)	-INF_32F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_32F	ippStsDomain
	64f	InvSqrt (x<0)	NAN_64F	ippStsDomain
		InvSqrt (x=+0)	INF_64F	ippStsSingularity
		InvSqrt (x=-0)	-INF_64F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_64F	ippStsDomain
ippsInvCbrt	32f	InvCbrt (x=+0)	INF_32F	ippStsSingularity
		InvCbrt (x=-0)	-INF_32F	ippStsSingularity
	64f	InvCbrt (x=+0)	INF_64F	ippStsSingularity
		InvCbrt (x=-0)	-INF_64F	ippStsSingularity
ippsPow	32f	Pow (x=+0, y=-ODD_INT)	INF_32F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_32F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_32F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_32F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_32F	ippStsDomain
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_64F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_64F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_64F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_64F	ippStsDomain
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity

表 A-2　インテル® IPP 固定精度数値演算関数の特殊な事例　（続き）

関数の基本名	データ・タイプ	特殊な事例	結果値	ステータス・コード
ippsPowx	32f	Pow (x=+0, y=-ODD_INT)	INF_32F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_32F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_32F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_32F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_32F	ippStsDomain
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_64F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_64F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_64F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_64F	ippStsDomain
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity
ippsExp	32f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_32F	ippStsOverflow
	64f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_64F	ippStsOverflow
ippsLn	32f	Ln (x<0)	NAN_32F	ippStsDomain
		Ln (x=-INF)	NAN_32F	ippStsDomain
		Ln (x=0)	-INF_32F	ippStsSingularity
	64f	Ln (x<0)	NAN_64F	ippStsDomain
		Ln (x=-INF)	NAN_64F	ippStsDomain
		Ln (x=0)	-INF_64F	ippStsSingularity
ippsLog10	32f	Ln (x<0)	NAN_32F	ippStsDomain
		Ln (x=-INF)	NAN_32F	ippStsDomain
		Ln (x=0)	-INF_32F	ippStsSingularity
	64f	Ln (x<0)	NAN_64F	ippStsDomain
		Ln (x=-INF)	NAN_64F	ippStsDomain
		Ln (x=0)	-INF_64F	ippStsSingularity
ippsCos	32f	Cos (INF)	NAN_32F	ippStsDomain
	64f	Cos (INF)	NAN_64F	ippStsDomain
ippsSin	32f	Sin (INF)	NAN_32F	ippStsDomain
	64f	Sin (INF)	NAN_64F	ippStsDomain
ippsSinCos	32f	SinCos (INF)	NAN_32F, NAN_32F	ippStsDomain
	64f	SinCos (INF)	NAN_64F, NAN_64F	ippStsDomain
ippsTan	32f	Tan (INF)	NAN_32F	ippStsDomain
	64f	Tan (INF)	NAN_64F	ippStsDomain
ippsAcos	32f	Acos (x), |x|>1	NAN_32F	ippStsDomain
		Acos (INF)	NAN_32F	ippStsDomain
	64f	Acos (x), |x|>1	NAN_64F	ippStsDomain
		Acos (INF)	NAN_64F	ippStsDomain
ippsAsin	32f	Acos (x), |x|>1	NAN_32F	ippStsDomain
		Acos (INF)	NAN_32F	ippStsDomain
	64f	Acos (x), |x|>1	NAN_64F	ippStsDomain
		Acos (INF)	NAN_64F	ippStsDomain

表 A-2　インテル® IPP 固定精度数値演算関数の特殊な事例 （続き）

関数の基本名	データ・タイプ	特殊な事例	結果値	ステータス・コード
ippsCosh	32f	Cosh (x), |x|>overflow	INF_32F	ippStsOverflow
	64f	Cosh (x), |x|>overflow	INF_64F	ippStsOverflow
ippsSinh	32f	Sinh (x), |x|>overflow	INF_32F	ippStsOverflow
	64f	Sinh (x), |x|>overflow	INF_64F	ippStsOverflow
ippsAcosh	32f	Acosh (x<1)	NAN_32F	ippStsDomain
		Acosh (x=-INF)	NAN_32F	ippStsDomain
	64f	Acosh (x<1)	NAN_64F	ippStsDomain
		Acosh (x=-INF)	NAN_64F	ippStsDomain
ippsAtanh	32f	Atanh (x=1)	INF_32F	ippStsSingularity
		Atanh (x=-1)	-INF_32F	ippStsSingularity
		Atanh (x), |x|>1	NAN_32F	ippStsDomain
		Atanh (INF)	NAN_32F	ippStsDomain
	64f	Atanh (x=1)	INF_64F	ippStsSingularity
		Atanh (x=-1)	-INF_64F	ippStsSingularity
		Atanh (x), |x|>1	NAN_64F	ippStsDomain
		Atanh (INF)	NAN_64F	ippStsDomain

intel®

参考文献

ここでは、アプリケーション・プログラマにとって参考になる文献とその他の情報源を紹介する。文献の一覧は、完璧なものではなく、すべてを網羅しているわけではない。一つの出発点と考えていただきたい。すべての参考文献の中で、信号処理の基礎知識がある人にとっては [Mit93] が一番有用である。ここでは、この分野の専門家 27 名の著作を挙げており、広範かつ高度な知識を提供する。

[Opp75]、[Opp89]、[Jac89]、[Zie83] は、信号処理を理解するための教科書である。[Opp89] は、古典である [Opp75] を大幅に改訂したものである。[Jac89] は、他のものに比べてコンパクトになっている。[Zie83] では、連続時間系に関する説明もされている。

[Ash94] M.R. Asharif、F. Amano 著『Acoustic Echo Canceler Using the FBAF Algorithm』(IEEE Trans.Comm., Vol. 42、No. 12、1994 年 12 月、pp.3090 ～ 3094)

[Bri94] C.Brislawn 著『Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks』(Los Alamos Report LA-UR-94-1747、1994 年)

[Cap78] V.Cappellini、A.G.Constantinides、P.Emilani 著『Digital Filters and Their Applications』(ロンドンー Academic Press 刊、1978 年)

[Cap94] Cappé O. 著『Elimination of the musical noise phenomenon with the Ephraim and Malah noise suppressor』(IEEE Trans. Speech and Audio Processing、vol. 2(2)、1994 年)

[CCITT] CCITT の勧告 G.711『Pulse Code Modulation of Frequencies』(1984 年)

[Coh02] I.Cohen、B.Berdugo 著『Noise Estimation by Minima Controlled Recursive Averaging for Robust Speech Enhancement』(IEEE Signal Proc.Letters、Vol.9、No.1、2002 年 1 月、pp.12 ～ 15)

[Cro83] R.E.Crochiere、L.R.Rabiner 著『Multirate Digital Signal Processing』(ニュージャージー州イーグルウッド・クリフープレンティス・ホール刊、1983 年)

[Dau92] I.Daubechies 著『Ten Lectures on Wavelets』(ペンシルベニア州ースプリンガー出版刊、1992 年)

[Eph84] Y.Ephraim、D.Malah 著 『Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator』（IEEE Trans. ASSP、Vol. 32、No. 6、1984 年 12 月、pp. 1109 ～ 1121)

[ES201] ETSI ES 201 108 V1.1.2.ETSI Standard.『Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Front-end feature extraction algorithm; Compression algorithms』

[ES202] ETSI ES 202 050 V1.1.1.ETSI Standard.『Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Advanced front-end feature extraction algorithm; Compression algorithms』

[Fei92] E.Feig、S.Winograd 著『Fast algorithms for DCT』（IEEE Transactions on Signal Processing、vol.40、No.9、1992 年）

[Ham83] R.W.Hamming 著『Digital Filters』（ニュージャージー州 －プレンティス・ホール刊、1983 年)

[Har78] F.Harris 著 『On the Use of Windows』（IEEE 会報、vol.66、No.1、IEEE、1978 年）

[Hay91] S.Haykin 著『Adaptive Filter Theory』（ニュージャージー州イーグルウッド・クリフープレンティス・ホール刊、1991 年）

[ISO11172] ISO/IEC 11172-3 - Information technology『Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s.Part 3: Audio』（1993 年）

[ISO13818] ISO/IEC 13818-3 - Information technology『Generic coding of moving pictures and associated audio information.Part 3: Audio』（1998 年）

[ISO14496] ISO/IEC 14496-3 - Information technology 『Coding of audio-visual objects.Part 3: Audio』（2001 年）

[ITU729] ITU-T の勧告 G.729 『Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)』（1996 年 3 月）

[ITU729A] ITU-T の勧告 G.729 付録 A『Reduced Complexity 8kbit/s CS-ACELP speech codec』（1996 年 11 月）

[ITU729B] ITU-T の勧告 G.729 付録 B『A silence compression scheme for G.729 optimized for terminals conforming to Recommendation V.70』(1996 年 10 月）

[ITU729B1] ITU-T の勧告 G.729 付録 B の正誤表 1（1998 年 2 月）

[ITU723] ITU-T の勧告 G.723.1 『Dual Rate speech coder for Multimedia Communications transmitting at 5.3 and 6.3 Kbit/s』（1996 年 3 月）

[ITU723A] ITU-T の勧告 G.723.1 付録 A 『Silence compression scheme』(1996 年 11 月)

[ITUV34] ITU-T の勧告 V.34 『A modem operating at data signalling rates of up to 33 600 bit/s for use on the general switched telephone network and on leased point-to-point 2-wire telephone-type circuit』(1998 年 2 月)

[ITU722] ITU-T の勧告 G.722.1『Coding at 24 and 32 8 kbit/s for hand-free operation in systems with low frame loss』(1999 年 9 月)

[Jac89] Leland B.Jackson 著 『Digital Filters and Signal Processing』(Kluwer Academic Publishers 刊、第 2 版、1989 年)

[Kab86] P.Kabal、P.Ramachandran 著『The Computation of line Spectral Frequencies Using Chebyshev Polynomials』(IEEE transaction on acoustic、speech and signal processing、vol.ASSP-34、No.6、1986 年)

[Lyn89] Paul A.Lynn 著『Introductory Digital Signal Processing with Computer Applications』(ニューヨーク－ John Wiley&Sons, Inc. 刊、1993 年)

[Mar01] R.Martin 著 『Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics』(IEEE Trans.Speech and Audio、Vol.9、No.5、2001 年 7 月、pp.504 ～ 512)

[Med91] Y.Medan、E.Yair、D.Chazan 著『Super Resolution Pitch Determination of Speech Signals』(IEEE Transactions on Signal Processing、vol 39、No.1、1991 年)

[Mit93] Sanjit K.Mitra、James F.Kaiser 他著 『Handbook for Digital Signal Processing』(ニューヨーク－ John Wiley&Sons, Inc. 刊、1993 年)

[Mit98] S.K.Mitra 著『Digital Signal Processing』(マグローヒル刊、1998 年)

[NIC91] Nam Ik Cho、Sang Uk Lee 著 『Fast algorithm and implementation of 2D DCT』(IEEE Transactions on Circuits and Systems、vol. 31、No.3、1991 年)

[Opp75] Alan V.Oppenheim、Ronald W.Schafer 著『Digital Signal Processing』(ニュージャージー州イーグルウッド・クリフープレンティス・ホール刊、1975 年)

[Opp89] Alan V.Oppenheim、Ronald W.Schafer 著 『Discrete-Time Signal Processing』(ニュージャージー州イーグルウッド・クリフープレンティス・ホール刊、1989 年)

[Rab78] L.R.Rabiner、R.W.Schafer 著 『Digital Processing of Speech Signals』(ニュージャージー州イーグルウッド・クリフープレンティス・ホール刊、1978 年)

[Rao90] K.R.Rao、P.Yip 著 『Discrete Cosine Transform.Algorithms, Advantages and Applications』（サンディエゴー Academic Press 刊、1990 年）

[Seg78] R.Sedgewick 著『Implementing quicksort programs』（ACM 会報、vol. 21、No. 10、pp. 847 ～ 857、1978 年 10 月)

[Str96] G.Strang、T.Nguyen 著 『Wavelet and Filter Banks』（Wellesley-Cambridge Press 刊、1996 年）

[Tuc92] R.Tucker 著 『Voice Activity Detection Using a Periodicity Measure』(IEE Proceedings-I、Vol. 139、No. 4、1992 年 8 月、pp.377 ～ 380)

[Vai93] P.P.Vaidyanathan 著 『Multirate Systems and Filter Banks』（ニュージャージー州イーグルウッド・クリフープレンティス・ホール刊）

[Wid85] B.Widrow、S.D.Stearns 著 『Adaptive Signal Processing』（ニュージャージー州イーグルウッド・クリフープレンティス・ホール刊、1985 年）

[Zie83] Rodger E.Ziemer、William H.Tranter、D.Ronald Fannin 著 『Signals and Systems:Continuous and Discrete』（ニューヨーク － Macmillan Publishing Co. 刊、1983 年）

用語集

BQ	モードの一つであり、該当する IIR 初期化関数がバイクワッド列を初期化するのを意味する。
CCS	「複素数の共役対称」を参照。
DCT	Discrete Cosine Transform の略語。
FIR	Finite Impulse Response フィルタの略称。FIR フィルタは、時間が経過しても、フィルタ係数（タップ）が変化しない。
FIR LMS	Least Mean Squares Finite Impulse Response フィルタの略称。
IIR	Infinite Impulse Response フィルタの略称。
LMS	Least Mean Square の略称。2 つの信号の差を計測する際によく使われるアルゴリズム。また、LMS のアルゴリズムを使って適応処理を実行するような適応 FIR フィルタの略記形としても使用される。
LTI	Linear Time-Invariant システムの略称。LTI システムでは、入力が一連の信号の合計である場合、その出力は各信号に対して該当システムが別々に処理した結果（応答）合計になる [Lyn89]。

MMX® テクノロジ（MMX® technology）

マルチメディアや通信アプリケーションでより高い性能を発揮するのを狙って開発された、インテル® アーキテクチャの拡張機能。MMX テクノロジでは、4 つの追加のデータ・タイプ、8 つの 64 ビット MMX テクノロジ・レジスタ、SIMD（single instruction, multiple data）を実装した 57 個の追加命令を使用する。

MR	「モード」の 1 つであり、該当する関数がマルチレート版であることを示す。
CCS	複素数の共役対称シーケンスの表現方法の 1 つ。Pack 形式や Perm 形式よりも簡単に使用できる。

Pack

複素数の共役対称シーケンスのコンパクトな表現方法。この形式の欠点は、実数型の FFT アルゴリズムで使用される自然な形式ではないことである（基数 2 の複素数型 FFT では、ビットの順序が逆になっている方が自然）。

Perm

FFT アルゴリズム用の一連の値を格納するための形式。RCPerm 形式では、FFT アルゴリズムによって使用する順番に従って、一連の値が格納される。つまり、サンプルの実数部と虚数部が隣接する必要はない。

Streaming SIMD Extensions（SSE）

インテル・アーキテクチャ命令セットの主要な拡張機能。SSE には、キャッシュ制御命令やステート管理命令とともに、パックド・データを操作する汎用浮動小数点命令や追加のパックド整数命令が組み込まれている。これらの命令を使用すると、浮動小数点や整数データを集中的に処理するアプリケーションの性能が大幅に向上する。

アップサンプリング（up-sampling）

アップサンプリングは、信号の隣接サンプル間に値ゼロのサンプルを挿入すると、信号のサンプリング・レートを概念的に上げる。

圧伸関数（companding functions）

対数のエンコーダとデコーダを使ってデータ圧縮処理を実行する関数。圧伸を使用すれば、量子化レベルの間隔を対数的に配分して、パーセンテージ・エラーを一定に保つことができる。

インプレース（in-place）

この方式で演算を実行する関数は、配列から入力を受け取り、その出力を同じ配列に返す。「ノット・インプレース」を参照。

共役（conjugate）

複素数 $a + bj$ の共役は、$a - bj$ である。

共役対称 （conjugate-symmetric）

「complex conjugate-symmetric」を参照。

固定小数点データ形式（fixed-point data format）

1 ビットを符合に割り当て、他のビットはすべて小数部に割り当てるような形式のこと。この形式は、符合付きの純粋小数ベクトルを使った最適化変換演算で使用される。例えば、S.31 の形式の場合は、符合ビットが 1 つで、小数部が 31 ビットになる。また、S15.16 の形式の場合は、符合ビットが 1 つ、整数部が 15 ビット、そして小数部が 16 ビットになる。

サイン・カーブ（sinusoid）

「トーン」を参照。

算術演算（arithmetic operation）

イメージ・ピクセルの値を加算、減算、乗算、または 2 乗する演算。

ダウンサンプリング（down-sampling）

ダウンサンプリングは、概念的に信号の間引きを実行して信号のサンプリング・レートを下げる。「デシメーション」を参照。

適応フィルタ （adaptive filter）

適応フィルタは、時間の経過に伴い、フィルタ係数（タップ）が変化する。フィルタの係数が変化するのは通常、プロトタイプの「目標」信号に出力をできるだけ近づけるためである。適応フィルタ以外のフィルタでは、時間が経過しても、フィルタ係数が変化しない。

デシメーション（decimation）

ダウンサンプリングの前に実行する信号のフィルタリング。このフィルタリングを実行すれば、以降のダウンサンプリングで歪のエイリアシングを防止できる。「ダウンサンプリング」を参照。

トーン（tone）

特定の周波数、位相、大きさを持つサイン・カーブ。テスト信号や、より複雑な信号の基本要素として使用される。

ノット・インプレース（not-in-place）

この方式で演算を実行する関数は、ソース配列から入力を受け取り、その出力を次のデスティネーション配列に格納する。

複素数の共役対称（complex conjugate-symmetric）

実数信号のフーリエ変換で発生する一種の対称性。複素数の共役対称性を持つ信号には、$x(-n) = x(n)^*$ の特性がある（「*」は共役を示す）。

飽和（saturation）

飽和計算を使用すると、データ・タイプで許されたデータ範囲を数値が超えたとき、その数値がデータ範囲の上限に飽和される。例えば、7FFFh を超える符号付きワードは、7FFFh に飽和する。数値がデータ範囲の下限より小さくなると、その数値はデータ範囲の下限に飽和される。例えば、8000h より小さい符号付きワードは、8000h に飽和する。

補間 （interpolation）

フィルタリングの前に実行する信号のアップサンプリング。このフィルタリングでは、元の信号内の前後のサンプル値に近い値を持つ一連のサンプルが挿入される。「アップサンプリング」を参照。

ポリフェーズ（polyphase）

マルチレート・フィルタリングの計算を効率よく実行するような方法。例えば、補間やデシメーションはその例である。

マルチレート（multi-rate）

信号のサンプリング・レートが複数個あるような演算または信号処理システムのこと。マルチレート演算の例としては、デシメーションや補間がある。

窓（Window）

数学的な関数であり、信号の乗算を行って以降の分析の特性を改善する。FFT ベースのスペクトル分析でよく使用される。

要素単位（element-wise）

要素単位の演算は、ベクトルの各要素に同一の演算を実行したり、複数のベクトルの中の同一の位置にある要素を演算の入力として使ったりする。例えば、$\{x_0, x_1, x_2\}$ および $\{y_0, y_1, y_2\}$ のベクトルを要素単位で加算すると、$\{x_0, x_1, x_2\}+\{y_0, y_1, y_2\}=\{x_0 + y_0, x_1 + y_1, x_2 + y_2\}$ のようになる。

索引

数字

10Log10, 5-44

A

Abs, 5-33
AccCovarianceMatrix, 8-67
ACELPFixedCodebookSearch_G723, 9-93
Acos, 12-30
Acosh, 12-43
AdaptiveCodebookContribution_G729, 9-56
AdaptiveCodebookGain_G729, 9-57
AdaptiveCodebookSearch_G723, 9-94
AdaptiveCodebookSearch_G729, 9-41
Add, 5-16
AddAllRowSum, 8-7
AddC, 5-14
AddMulColumn, 8-160
AddMulRow, 8-161
AddNRows, 8-85
AddProduct, 5-18
ALawToLin, 10-52
ALawToMuLaw, 10-55
AlignPtr, 3-11
AnalysisPQMF_MP3, 10-62
And, 5-5
AndC, 5-4
ApplySF_I, 10-11
Arctan, 5-46
Asin, 12-32
Asinh, 12-45
Atan, 12-34
Atan2, 12-36
Atanh, 12-46
AutoCorr, 6-4
AutoCorr_G723, 9-83
AutoCorr_G729, 9-23
AutoCorrLagMax, 9-13
AutoCorr_NormE, 9-14
AutoCorr_NormE_G723, 9-85
AutoScale, 9-9

B

BhatDist, 8-150
BlindEqualization_Aurora, 8-202
BlockDMatrixFree, 8-14
BlockDMatrixInitAlloc, 8-13
BuildHDT, 10-29
BuildHET, 10-33
BuildHET_VLC, 10-37
BuildSignTable, 8-121
BuildSymblTableDV4D, 5-86

C

CalcSF, 10-10
CalcStatesDV, 5-85
CartToPolar, complex, 5-77
Cbrt, 12-10
CCS 形式 , 7-4
CdbkFree, 8-178, 10-42
CdbkGetSize, 8-174
CdbkInit, 8-175
CdbkInitAlloc, 8-177, 10-41
CepstrumToLP, 8-28
Chebyshev 多項式 , 9-27
CoefUpdateAECNLMS, 8-233
Compare, 11-7
CompensateOffset, 8-19
Concat, 11-16
ConcatC, 11-17
Conj, 5-54

intel®

ConjCcs, 7-9
ConjFlip, 5-55
ConjPack, 7-7
ConjPerm, 7-5
ControllerGetSizeAEC, 8-236
ControllerInitAEC, 8-237
ControllerUpdateAEC, 8-238
Conv, 6-2
ConvCyclic, 6-3
Convert, 5-51
ConvPartial, 9-5
Copy, 4-2
CopyColumn, 8-69
CopyColumn_Indirect, 8-12
CopyWithPadding, 8-41
CoreGetCpuClocks, 3-8
CoreGetCpuType, 3-7
CoreGetStatusString, 3-6
CoreSetDenormAreZeros, 3-10
CoreSetFlushToZero, 3-9
Cos, 12-23
Cosh, 12-38
CountBits, 10-38
CplxToReal, 5-64
CrossCorr, 6-6, 9-15
CrossCorrCoeff, 8-82
CrossCorrCoeffDecim, 8-81
CrossCorrCoeffInterpolation, 8-83
Cubrt, 5-39

D

DcsClustLAccumulate, 8-157
DcsClustLCompute, 8-159
DCT 関数 , 7-42 – 7-47
  DCTFwd, 7-46
  DCTFwdFree, 7-44
  DCTFwdGetBufSize, 7-45
  DCTFwdInitAlloc, 7-43
  DCTInv, 7-46
  DCTInvFree, 7-44
  DCTInvGetBufSize, 7-45
  DCTInvInitAlloc, 7-43
DCTLifter, 8-59
DCTLifterFree, 8-58
DCTLifterGetSize_MulC0, 8-55
DCTLifterInitAlloc, 8-57
DCTLifterInit_MulC0, 8-56
DecodeAdaptiveVector_G723, 9-108
DecodeAdaptiveVector_G729, 9-44
DecodeChanPairElt_AAC, 10-117
DecodeDatStrElt_AAC, 10-122
DecodeExtensionHeader_AAC, 10-141
DecodeFillElt_AAC, 10-123
DecodeGain_G729, 9-50
DecodeIsStereo_AAC, 10-129
DecodeMainHeader_AAC, 10-140
DecodeMsStereo_AAC, 10-127
DecodePNS_AAC, 10-142
DecodePrgCfgElt_AAC, 10-116
DecodeTNS_AAC, 10-133
DecodeVLC, 10-30
Deinterleave, 10-6
DeinterleaveSpectrum_AAC, 10-131
Delta, 8-72
DeltaDelta, 8-76
DFT 関数 , 7-25 – 7-35
  DFTFree_C, 7-27
  DFTFree_R, 7-27
  DFTFwd_CToC, 7-29
  DFTFwd_RToCCS, 7-32
  DFTFwd_RToPack, 7-32
  DFTFwd_RToPerm, 7-32
  DFTGetBufSize_C, 7-28
  DFTGetBufSize_R, 7-28
  DFTInitAlloc_C, 7-26
  DFTInitAlloc_R, 7-26
  DFTInv_CCSToR, 7-32
  DFTInv_CToC, 7-29
  DFTInv_PackToR, 7-32
  DFTInv_PermToR, 7-32
  DFTOutOrdFree_C, 7-36
  DFTOutOrdFwd_CToC, 7-37
  DFTOutOrdGetBufSize_C, 7-37
  DFTOutOrdInitAlloc_C, 7-35
  DFTOutOrdInv_CToC, 7-37
DFTOutOrdFree_C, 7-36
DFTOutOrdFwd_CToC, 7-37
DFTOutOrdGetBufSize_C, 7-37
DFTOutOrdInitAlloc_C, 7-35
Div, 5-31, 12-5
DivC, 5-28
DivCRev, 5-30

intel®

DotProd, 5-114
DotProdAutoScale, 9-10
DotProdColumn, 8-163
DotProd_G729, 9-20
DTW, 8-128
Durbin, 8-23

E

EMNS 関数
 FilterUpdateEMNS, 8-215
 FilterUpdateWiener, 8-217
 GetSizeMCRA, 8-218
 InitAllocMCRA, 8-220
 InitMCRA, 8-219
 UpdateNoisePSDMCRA, 8-221
EmptyFBankInitAlloc, 8-49
EncodeBlock, 10-39
EncodeTNS_AAC, 10-145
EncodeVLC, 10-35
Entropy, 8-147
Equal, 11-9
Erf, 12-48
Erfc, 12-50
EvalDelta, 8-70
EvalDelta_Aurora, 8-203
EvalFBank, 8-54
Exp, 5-40, 12-18
ExpNegSqr, 8-149

F

FBankFree, 8-50
FBankGetCenters, 8-50
FBankGetCoeffs, 8-52
FBankSetCenters, 8-51
FBankSetCoeffs, 8-53
FDPFree, 10-23
FDPFwd, 10-25
FDPInitAlloc, 10-22
FDPInv, 10-26
FFT 関数 , 7-15 － 7-24
 FFTFree_C, 7-17
 FFTFree_R, 7-17
 FFTFwd_CToC, 7-19
 FFTFwd_RToCCS, 7-22
 FFTFwd_RToPack, 7-22
 FFTFwd_RToPerm, 7-22
 FFTGetBufSize_C, 7-18
 FFTGetBufSize_R, 7-18
 FFTInitAlloc_C, 7-16
 FFTInitAlloc_R, 7-16
 FFTInv_CCSToR, 7-22
 FFTInv_CToC, 7-19
 FFTInv_PackToR, 7-22
 FFTInv_PermToR, 7-22
FillShortlist_Column, 8-125
FillShortlist_Row, 8-123
FilterAECNLMS, 8-232
FilterMedian, 6-100
FilterUpdateEMNS, 8-215
FilterUpdateWiener, 8-217
FindC, FindRevC, 11-3
FindCAny, FindRevCAny, 11-4
FindNearest, 5-83
FindNearestOne, 5-82
FindPeaks, 8-240
Find、FindRev, 11-2
FIR, 6-32
FIR LMS フィルタ関数 , 6-53 － 6-58, 6-63 － 6-74
FIR フィルタ関数 , 6-12 － 6-30
 FIR, 6-32
 FIR_Direct, 6-40
 FIRFree, 6-17, 6-30
 FIRGenBandpass, 6-51
 FIRGenBandstop, 6-52
 FIRGenHighpass, 6-50
 FIRGenLowpass, 6-48
 FIRGetDlyLine, 6-28
 FIRGetStateSize, 6-23
 FIRGetTaps, 6-25
 FIRInit, 6-18
 FIRInitAlloc, 6-13
 FIRMR_Direct, 6-44
 FIRMRGetStateSize, 6-23
 FIRMRInit, 6-18
 FIRMRInitAlloc, 6-13
 FIROne, 6-30
 FIROne_Direct, 6-37
 FIRSetDlyLine, 6-28
 FIRSetTaps, 6-25
FIRBlockFree, 10-19
FIRBlockInitAlloc, 10-18
FIRBlockOne, 10-19

FIR_Direct, 6-40
FIRFree, 6-17, 6-30
FIRGenBandpass, 6-51
FIRGenBandstop, 6-52
FIRGenHighpass, 6-50
FIRGenLowpass, 6-48
FIRGetDlyLine, 6-28
FIRGetStateSize, 6-23
FIRGetTaps, 6-25
FIRInit, 6-18
FIRInitAlloc, 6-13
FIRLMS, 6-58
FIRLMSFree, 6-55
FIRLMSGetDlyLine, 6-57
FIRLMSGetTaps, 6-56
FIRLMSInitAlloc, 6-54
FIRLMSMRFree, 6-65
FIRLMSMRGetDlyVal, 6-71
FIRLMSMRGetTapsPointer, 6-69
FIRLMSMRInitAlloc, 6-64
FIRLMSMROne, 6-72
FIRLMSMROneVal, 6-73
FIRLMSMRPutVal, 6-72
FIRLMSMRSetMu, 6-66
FIRLMSMRUpdateTaps, 6-67
FIRLMSOne_Direct, 6-61
FIRLMSSetDlyLine, 6-57
FIRMR_Direct, 6-44
FIRMRGetStateSize, 6-23
FIRMRInit, 6-18
FIRMRInitAlloc, 6-13
FIROne, 6-30
FIROne_Direct, 6-37
FIRSetDlyLine, 6-28
FIRSetTaps, 6-25
FixedCodebookSearch_G729, 9-45
flag 引数と hint 引数 , 7-3
Flip, 5-81
FormVector, 8-171
FormVectorVQ, 8-185

G

G.729 コーデック関数 , 9-19
GainControl_G723, 9-100
GainControl_G729, 9-52
GainQuant_G723, 9-99
GainQuant_G729, 9-54
GaussianDist, 8-144
GaussianMerge, 8-146
GaussianSplit, 8-145
GetCdbkSize, 8-179
GetCodebook, 8-180
GetCpuFreqMhz, 3-8
GetLibVersion, 3-2
GetSizeHDT, 10-28
GetSizeHET, 10-32
GetSizeHET_VLC, 10-36
GetSizeMCRA, 8-218
GetVarPointDV, 5-84
Goertz, 7-39
GoertzTwo, 7-41
GSM-AMR 音声コーデック
  10 次 LS 係数の計算 , 9-118
  10 次 LP 係数を LSP 係数に変換 , 9-119
  10 次 LSP 係数を LP 係数に変換 , 9-121
  LP 分析と量子化プリミティブ , 9-115
  LSP 係数ベクトルの量子化 , 9-123
  SID フレームのパラメータの抽出 , 9-156
  VAD 1 機能の実装 , 9-152
  VAD 2 機能の実装 , 9-154
  アダプティブ・コードブック・パラメータのデコード , 9-143
  アダプティブ・コードブック検索 , 9-139
  アダプティブ・コードブック・プリミティブ , 9-127
  インパルス応答およびターゲット信号の計算 , 9-138
  オープン・ループのピッチ・ラグ推定の抽出 , 9-132
  オープン・ループのピッチ・ラグ推定の抽出 (VAD 2), 9-135
  オープン・ループのピッチ・ラグおよび最適なピッチ・ゲインの計算 , 9-130
  オープン・ループ・ピッチの検索 , 9-127
  現在のフレームの SID の確認 , 9-159
  合成音声のフィルタ , 9-161
  固定コードブック・プリミティブ , 9-147
  固定コードブック・ベクトルのデコード , 9-150
  自己相関シーケンス , 9-116
  代数コードブック検索の検索 , 9-147
  断片的な送信 (DTX) プリミティブ , 9-151

量子化された LSP のデコード , 9-125
音声コーデック
GSM-AMR, 9-112
GSM-AMR コーデック
LSP または LSF 係数のバッファ , 9-160
GSM-AMR 音声コーデック , 9-112
データ構造体 , 9-3

H

HarmonicFilter, 9-70
HarmonicNoiseSubtract_G723, 9-107
HarmonicSearch_G723, 9-106
Hash, 11-15
HighBandCoding_Aurora, 8-201
HighPassFilter_G723, 9-102
HighPassFilter_G729, 9-73
HighPassFilterInit_G729, 9-72
HighPassFilterSize_G729, 9-72
Hilbert, 7-50
HilbertFree, 7-49
HilbertInitAlloc, 7-49
HuffmanCountBits, 10-34
HuffmanDecode_MP3, 10-98
HuffmanEncode_MP3, 10-84

I

IIR, 6-92
IIR フィルタ関数 , 6-74 – 6-90
IIR, 6-92
IIR_BiQuadDirect, 6-98
IIR_Direct, 6-98
IIRFree, 6-80
IIRGetDlyLine, 6-88
IIRGetStateSize, IIRGetStateSize_BiQuad, 6-84
IIRInit, IIRInit_BiQuad, 6-81
IIRInitAlloc, 6-76
IIRInitAlloc_BiQuad, 6-76
IIROne, 6-90
IIROne_BiQuadDirect, 6-97
IIROne_Direct, 6-97
IIRSetDlyLine, 6-88
IIRSetTaps, 6-86
IIR16s_G723, 9-102
IIR16s_G729, 9-74
IIR_BiQuadDirect, 6-98
IIR_Direct, 6-98
IIRFree, 6-80
IIRGetDlyLine, 6-88
IIRGetStateSize, IIRGetStateSize_BiQuad, 6-84
IIRInit, IIRInit_BiQuad, 6-81
IIRInitAlloc, 6-76
IIRInitAlloc_BiQuad, 6-76
IIROne, 6-90
IIROne_BiQuadDirect, 6-97
IIROne_Direct, 6-97
IIRSetDlyLine, 6-88
IIRSetTaps, 6-86
Imag, 5-62
IndexSelect_VQ, 10-46
InitAllocMCRA, 8-220
InitBitReservoir_MP3, 10-90
InitMCRA, 8-219
Insert, 11-5
Interleave, 10-5
Interpolate_G729, 9-21
Inv, 12-3
InvCbrt, 12-12
InvSqrt, 9-11, 12-8
IPP ソフトウェアの概要 , 1-1
ippAlignPtr(), AlignPtr を参照
ippCoreGetCpuClocks(), CoreGetCpuClocks を参照
ippCoreGetCpuType(), CoreGetCpuType を参照
ippCoreGetStatusString(), CoreGetStatusString を参照
ippCoreSetDenormAreZeros(), CoreSetDenormAreZeros を参照
ippCoreSetFlushToZero(), CoreSetFlushToZero を参照
ippFree, 3-12
ippGetCpuFreqMhz(), GetCpuFreqMhz を参照
ippMalloc, 3-11
ipps10Log10(), 10Log10 を参照
ippsAbs(), Abs を参照
ippsAccCovarianceMatrix(), AccCovarianceMatrix を参照
ippsACELPFixedCodebookSearch_G723(), ACELPFixedCodebookSearch_G723 を参照
ippsAcos, 12-30
ippsAcosh, 12-43
ippsAdaptiveCodebookContribution_G729(), AdaptiveCodebookContribution_G729 を参照

ippsAdaptiveCodebookDecode_GSMAMR_16s, 9-143
ippsAdaptiveCodebookGain_G729(), AdaptiveCodebookGain_G729 を参照
ippsAdaptiveCodebookSearch_G723(), AdaptiveCodebookSearch_G723 を参照
ippsAdaptiveCodebookSearch_G729(), AdaptiveCodebookSearch_G729 を参照
ippsAdd(), Add を参照
ippsAddAllRowSum(), AddAllRowSum を参照
ippsAddC(), AddC を参照
ippsAddMulColumn(), AddMulColumn を参照
ippsAddMulRow(), AddMulRow を参照
ippsAddNRows(), AddNRows を参照
ippsAddProduct(), AddProduct を参照
ippsALawToLin(), ALawToLin を参照
ippsALawToMuLaw(), ALawToMuLaw
ippsAlgebraicCodebookSearch_GSMAMR_16s, 9-147
ippsAnalysisPQMF_MP3(), AnalysisPQMF_MP3 を参照
ippsAnalysisPQMF_MP3_16s32s, 10-62
ippsAnd(), And を参照
ippsAndC(), AndC を参照
ippsApplySF_I(), ApplySF_I を参照
ippsArctan(), Arctan を参照
ippsAsin, 12-32
ippsAsinh, 12-45
ippsAtan, 12-34
ippsAtan2, 12-36
ippsAtanh, 12-46
ippsAutoCorr(), AutoCorr を参照
ippsAutoCorr_G723(), AutoCorr_G723 を参照
ippsAutoCorr_G729(), AutoCorr_G729 を参照
ippsAutoCorr_GSMAMR, 9-116
ippsAutoCorrLagMax(), AutoCorrLagMax を参照
ippsAutoCorr_NormE(), AutoCorr_NormE を参照
ippsAutoCorr_NormE_G723(), AutoCorr_NormE_G723 を参照
ippsAutoScale(), AutoScale を参照
ippsBhatDist(), BhatDist を参照
ippsBlindEqualization_Aurora(), BlindEqualization_Aurora を参照
ippsBlockDMatrixFree(), BlockDMatrixFree を参照
ippsBlockDMatrixInitAlloc(), BlockDMatrixInitAlloc を参照
ippsBuildHDT(), BuildHDT を参照
ippsBuildHET(), BuildHET を参照
ippsBuildHET_VLC(), BuildHET_VLC
ippsBuildSignTable(), BuildSignTable を参照
ippsBuildSymblTableDV4D(), BuildSymblTableDV4D を参照
ippsCalcSF(), CalcSF を参照
ippsCalcStatesDV(), CalcStatesDV を参照
ippsCartToPolar(), CartToPolar、complex を参照
ippsCbrt, 12-10
ippsCdbkFree(), CdbkFree を参照
ippsCdbkGetSize(), CdbkGetSize を参照
ippsCdbkInit(), CdbkInit を参照
ippsCdbkInitAlloc(), CdbkInitAlloc を参照
ippsCepstrumToLP(), CepstrumToLP を参照
ippsCoefUpdateAECNLMS(), CoefUpdateAECNLMS を参照
ippsCompare(), Compare を参照
ippsCompareIgnoreCase(), CompareIgnoreCase を参照
ippsCompensateOffset(), CompensateOffset を参照
ippsConcat(), Concat を参照
ippsConcatC(), ConcatC を参照
ippsConj(), Conj を参照
ippsConjCcs(), ConjCcs を参照
ippsConjFlip(), ConjFlip を参照
ippsConjPack(), ConjPack を参照
ippsControllerGetSizeAEC(), ControllerGetSizeAEC を参照
ippsControllerInitAEC(), ControllerInitAEC を参照
ippsControllerUpdateAEC(), ControllerUpdateAEC を参照
ippsConv(), Conv を参照
ippsConvCyclic(), ConvCyclic を参照
ippsConvert(), Convert を参照
ippsConvert(), SortAscend を参照
ippsConvPartial(), ConvPartial を参照
ippsCopy(), Copy を参照
ippsCopyColumn(), CopyColumn を参照
ippsCopyColumn_Indirect(), CopyColumn_Indirect を参照
ippsCopyWithPadding(), CopyWithPadding を参照
ippsCos, 12-23
ippsCosh, 12-38

ippsCountBits(), CountBits を参照
ippsCplxToReal(), CplxToReal を参照
ippsCrossCorr(), CrossCorr を参照
ippsCrossCorrCoeff(), CrossCorrCoeff を参照
ippsCrossCorrCoeffDecim(), CrossCorrCoeffDecim を参照
ippsCrossCorrCoeffInterpolation(), CrossCorrCoeffInterpolation を参照。
ippsCubrt(), Cubrt を参照
ippsDcsClustLAccumulate(), DcsClustLAccumulate を参照
ippsDcsClustLCompute(), DcsClustLCompute を参照
ippsDCTFwd(), DCTFwd を参照
ippsDCTFwdFree(), DCTFwdFree を参照
ippsDCTFwdGetBufSize(), DCTFwdGetBufSize を参照
ippsDCTFwdInitAlloc(), DCTFwdInitAlloc を参照
ippsDCTInv(), DCTInv を参照
ippsDCTInvFree(), DCTInvFree を参照
ippsDCTInvGetBufSize(), DCTInvGetBufSize を参照
ippsDCTInvInitAlloc(), DCTInvInitAlloc を参照
ippsDCTLifter(), DCTLifter を参照
ippsDCTLifterFree(), DCTLifterFree を参照
ippsDCTLifterGetSize_MulC0(), DCTLifterGetSize_MulC0 を参照
ippsDCTLifterInitAlloc(), DCTLifterInitAlloc を参照
ippsDCTLifterInit_MulC0(), DCTLifterInit_MulC0 を参照
ippsDecodeAdaptiveVector_G723(), DecodeAdaptiveVector_G723 を参照
ippsDecodeAdaptiveVector_G729(), DecodeAdaptiveVector_G729 を参照
ippsDecodeChanPairElt_AAC, 10-117
ippsDecodeChanPairElt_AAC(), DecodeChanPairElt_AAC を参照
ippsDecodeDatStrElt_AAC, 10-122
ippsDecodeDatStrElt_AAC(), DecodeDatStrElt_AAC を参照
ippsDecodeExtensionHeader_AAC, 10-141
ippsDecodeExtensionHeader_AAC(), DecodeExtensionHeader_AAC を参照
ippsDecodeFillElt_AAC, 10-123
ippsDecodeFillElt_AAC(), DecodeFillElt_AAC を参照
ippsDecodeGain_G729(), DecodeGain_G729 を参照
ippsDecodeIsStereo_AAC(), DecodeIsStereo_AAC を参照
ippsDecodeIsStereo_AAC_32s, 10-129
ippsDecodeMainHeader_AAC, 10-140
ippsDecodeMainHeader_AAC(), DecodeMainHeader_AAC を参照
ippsDecodeMsStereo_AAC(), DecodeMsStereo_AAC を参照
ippsDecodeMsStereo_AAC_32s_I, 10-127
ippsDecodePNS_AAC(), DecodePNS_AAC を参照
ippsDecodePNS_AAC_32s, 10-142
ippsDecodePrgCfgElt_AAC, 10-116
ippsDecodePrgCfgElt_AAC(), DecodePrgCfgElt_AAC を参照
ippsDecodeTNS_AAC(), DecodeTNS_AAC を参照
ippsDecodeTNS_AAC_32s_I, 10-133
ippsDecodeVLC(), DecodeVLC を参照
ippsDeinterleave(), Deinterleave を参照
ippsDeinterleaveSpectrum_AAC(), DeinterleaveSpectrum_AAC を参照
ippsDeinterleaveSpectrum_AAC_32s, 10-131
ippsDelta(), Delta を参照
ippsDeltaDelta(), DeltaDelta を参照
ippsDFTFree_C(), DFTFree_C を参照
ippsDFTFree_R(), DFTFree_R を参照
ippsDFTFwd_CToC(), DFTFwd_CToC を参照
ippsDFTFwd_RToCCS(), DFTFwd_RToCCS を参照
ippsDFTFwd_RToPack(), DFTFwd_RToPack を参照
ippsDFTFwd_RToPerm(), DFTFwd_RToPerm を参照
ippsDFTGetBufSize_R(), DFTGetBufSize_R を参照
ippsDFTInitAlloc_C(), DFTInitAlloc_C を参照
ippsDFTInitAlloc_R(), DFTInitAlloc_R を参照
ippsDFTInv_CCSToR(), DFTInv_CCSToR を参照
ippsDFTInv_CToC(), DFTInv_CToC を参照
ippsDFTInv_PackToR(), DFTInv_PackToR を参照
ippsDFTInv_PermToR(), DFTInv_PermToR を参照
ippsDFTOutOrdFree_C(), DFTOutOrdFree_C を参照
ippsDFTOutOrdFwd_CToC(), DFTOutOrdFwd_CToC を参照

ippsDFTOutOrdGetBufSize_C(), DFTOutOrdGetBufSize_C を参照
ippsDFTOutOrdInitAlloc_C(), DFTOutOrdInitAlloc_C を参照
ippsDFTOutOrdInv_CToC(), DFTOutOrdInv_CToC を参照
ippsDiv, 12-5
ippsDiv(), Div を参照
ippsDivC(), DivC を参照
ippsDivCRev(), DivCRev を参照
ippsDotProd (), DotProd を参照
ippsDotProdAutoScale(), DotProdAutoScale を参照
ippsDotProdColumn(), DotProdColumn を参照
ippsDotProd_G729(), DotProd_G729 を参照
ippsDTW(), DTW を参照
ippsDurbin(), Durbin を参照
ippsEmptyFBankInitAlloc(), EmptyFBankInitAlloc を参照
ippsEncDTXBuffer_GSMAMR_16s, 9-160
ippsEncDTXHandler_GSMAMR_16s, 9-159
ippsEncDTXSID_GSMAMR_16s, 9-156
ippsEncodeBlock(), EncodeBlock を参照
ippsEncodeTNS_AAC(), EncodeTNS_AAC を参照
ippsEncodeTNS_AAC_32s_I, 10-145
ippsEncodeVLC(), EncodeVLC を参照
ippsEntropy(), Entropy を参照
ippsEqual(), Equal を参照
ippsErf, 12-48
ippsErfc, 12-50
ippsEvalDelta(), EvalDelta を参照
ippsEvalDelta_Aurora(), EvalDelta_Aurora を参照
ippsEvalFBank(), EvalFBank を参照
ippsExp, 12-18
ippsExp(), Exp を参照
ippsExpNegSqr(), ExpNegSqr を参照
ippsFBankFree(), FBankFree を参照
ippsFBankGetCenters(), FBankGetCenters を参照
ippsFBankGetCoeffs(), FBankGetCoeffs を参照
ippsFBankSetCenters(), FBankSetCenters を参照
ippsFBankSetCoeffs(), FBankSetCoeffs を参照
ippsFDPFree(), FDPFree を参照
ippsFDPFwd(), FDPFwd を参照
ippsFDPInitAlloc(), FDPInitAlloc を参照
ippsFDPInv(), FDPInv を参照
ippsFFTFree_C(), FFTFree_C を参照
ippsFFTFree_R(), FFTFree_R を参照
ippsFFTFwd_CToC(), FFTFwd_CToC を参照
ippsFFTFwd_RToCCS(), FFTFwd_RToCCS を参照
ippsFFTFwd_RToPack(), FFTFwd_RToPack を参照
ippsFFTFwd_RToPerm(), FFTFwd_RToPerm を参照
ippsFFTGetBufSize_C(), FFTGetBufSize_C を参照
ippsFFTGetBufSize_R(), FFTGetBufSize_R を参照
ippsFFTInitAlloc_C(), FFTInitAlloc_C を参照
ippsFFTInitAlloc_R(), FFTInitAlloc_R を参照
ippsFFTInv_CCSToR(), FFTInv_CCSToR を参照
ippsFFTInv_CToC(), FFTInv_CToC を参照
ippsFFTInv_PackToR(), FFTInv_PackToR を参照
ippsFFTInv_PermToR(), FFTInv_PermToR を参照
ippsFillShortlist_Column(), FillShortlist_Column を参照
ippsFillShortlist_Row(), FillShortlist_Row を参照
ippsFilterAECNLMS(), FilterAECNLMS を参照
ippsFilterMedian, FilterMedian を参照
ippsFilterUpdateEMNS(), FilterUpdateEMNS を参照
ippsFilterUpdateWiener(), FilterUpdateWiener を参照
ippsFind(), Find、FindRev を参照
ippsFindC(), FindC、FindRevC を参照
ippsFindCAny(), FindCAny、FindRevCAny を参照
ippsFindNearest(), FindNearest を参照
ippsFindNearestOne(), FindNearestOne を参照
ippsFindPeaks(), FindPeaks を参照
ippsFIR(), FIR を参照
ippsFIRBlockFree(), FIRBlockFree を参照
ippsFIRBlockInitAlloc(), FIRBlockInitAlloc を参照
ippsFIRBlockOne(), FIRBlockOne を参照
ippsFIR_Direct(), FIR_Direct を参照
ippsFIRFree(), FIRFree を参照
ippsFIRGenBandpass(), FIRGenBandpass を参照
ippsFIRGenBandstop(), FIRGenBandstop を参照
ippsFIRGenHighpass(), FIRGenHighpass を参照
ippsFIRGenLowpass(), FIRGenLowpass を参照
ippsFIRGetDlyLine(), FIRGetDlyLine を参照
ippsFIRGetStateSize(), FIRGetStateSize を参照
ippsFIRGetTaps(), FIRGetTaps を参照
ippsFIRInit(), FIRInit を参照
ippsFIRInitAlloc(), FIRInitAlloc を参照
ippsFIRLMS(), FIRLMS を参照

ippsFIRLMSFree(), FIRLMSFree を参照
ippsFIRLMSGetDlyLine(), FIRLMSGetDlyLine を参照
ippsFIRLMSGetTaps(), FIRLMSGetTaps を参照
ippsFIRLMSInitAlloc(), FIRLMSInitAlloc を参照
ippsFIRLMSMRFree(), FIRLMSMRFree を参照
ippsFIRLMSMRGetDlyLine(), FIRLMSMRGetDlyLine を参照
ippsFIRLMSMRGetDlyVal(), FIRLMSMRGetDlyVal を参照
ippsFIRLMSMRGetTaps(), FIRLMSMRGetTaps を参照
ippsFIRLMSMRGetTapsPointer(), FIRLMSMRGetTapsPointer を参照
ippsFIRLMSMRInitAlloc(), FIRLMSMRInitAlloc を参照
ippsFIRLMSMROne(), FIRLMSMROne を参照
ippsFIRLMSMROneVal(), FIRLMSMROneVal を参照
ippsFIRLMSMRPutVal(), FIRLMSMRPutVal を参照
ippsFIRLMSMRSetDlyLine(), FIRLMSMRSetDlyLine を参照
ippsFIRLMSMRSetMu(), FIRLMSMRSetMu を参照
ippsFIRLMSMRSetTaps(), FIRLMSMRSetTaps を参照
ippsFIRLMSMRUpdateTaps(), FIRLMSMRUpdateTaps を参照
ippsFIRLMSOne_Direct(), FIRLMSOne_Direct を参照
ippsFIRLMSSetDlyLine(), FIRLMSSetDlyLine を参照
ippsFIRMR_Direct(), FIRMR_Direct を参照
ippsFIRMRGetStateSize(), FIRMRGetStateSize を参照
ippsFIRMRInit(), FIRMRInit を参照
ippsFIRMRInitAlloc(), FIRMRInitAlloc を参照
ippsFIROne(), FIROne を参照
ippsFIROne_Direct(), FIROne_Direct を参照
ippsFIRSetDlyLine(), FIRSetDlyLine を参照
ippsFIRSetTaps(), FIRSetTaps を参照
ippsFixedCodebookDecode_GSMAMR_16s, 9-150
ippsFixedCodebookSearch_G729(), FixedCodebookSearch_G729 を参照
ippsFlip(), Flip を参照
ippsFormVector(), FormVector を参照
ippsFormVectorVQ(), FormVectorVQ を参照
ippsFree, 3-5
ippsGainControl_G723(), GainControl_G723 を参照
ippsGainControl_G729(), GainControl_G729 を参照
ippsGainQuant_G723(), GainQuant_G723 を参照
ippsGainQuant_G729(), GainQuant_G729 を参照
ippsGaussianDist(), GaussianDist を参照
ippsGaussianMerge(), GaussianMerge を参照
ippsGaussianSplit(), GaussianSplit を参照
ippsGetBufSize_C(), DFTGetBufSize_C を参照
ippsGetCdbkSize(), GetCdbkSize を参照
ippsGetCodebook(), GetCodebook を参照
ippsGetLibVersion(), GetLibVersion を参照
ippsGetSizeHDT(), GetSizeHDT を参照
ippsGetSizeHET(), GetSizeHET を参照
ippsGetSizeHET_VLC(), GetSizeHET_VLC
ippsGetSizeMCRA(), GetSizeMCRA を参照
ippsGetVarPointDV(), GetVarPointDV を参照
ippsGoertz(), Goertz を参照
ippsGoertzTwo(), GoertzTwo を参照
ippsHarmonicFilter(), HarmonicFilter を参照
ippsHarmonicNoiseSubtract_G723(), HarmonicNoiseSubtract_G723 を参照
ippsHarmonicSearch_G723(), HarmonicSearch_G723 を参照
ippsHash(), Hash を参照
ippsHighBandCoding_Aurora(), HighBandCoding_Aurora を参照
ippsHighPassFilter_G723(), HighPassFilter_G723 を参照
ippsHighPassFilter_G729(), HighPassFilter_G729 を参照
ippsHighPassFilterInit_G729(), HighPassFilterInit_G729 を参照
ippsHighPassFilterSize_G729(), HighPassFilterSize_G729 を参照
ippsHilbert(), Hilbert を参照
ippsHilbertFree(), HilbertFree を参照
ippsHilbertInitAlloc(), HilbertInitAlloc を参照
ippsHuffmanCountBits(), HuffmanCountBits
ippsHuffmanDecode_MP3(), HuffmanDecode_MP3 を参照
ippsHuffmanEncode_MP3(), HuffmanEncode_MP3

を参照
ippsHuffmanEncode_MP3_32s1u, 10-84
ippsIIR(), IIR を参照
ippsIIR16s_G723(), IIR16s_G723 を参照
ippsIIR16s_G729(), IIR16s_G729 を参照
ippsIIR_BiQuadDirect(), IIR_BiQuadDirect を参照
ippsIIR_Direct(), IIR_Direct を参照
ippsIIRFree(), IIRFree を参照
ippsIIRGetDlyLine(), IIRGetDlyLine を参照
ippsIIRInitAlloc(), IIRInitAlloc を参照
ippsIIRInitAlloc_BiQuad(), IIRInitAlloc_BiQuad を参照
ippsIIROne(), IIROne を参照
ippsIIROne_BiQuadDirect(), IIROne_BiQuadDirect を参照
ippsIIROne_Direct(), IIROne_Direct を参照
ippsIIRSetDlyLine(), IIRSetDlyLine を参照
ippsImag(), Imag を参照
ippsImpulseResponseTarget_GSMAMR_16s, 9-138
ippsIndexSelect_VQ(), IndexSelect_VQ を参照
ippsInitAllocMCRA(), InitAllocMCRA を参照
ippsInitBitReservoir_MP3, 10-90
ippsInitBitReservoir_MP3(), InitBitReservoir_MP3 を参照
ippsInitMCRA(), InitMCRA を参照
ippsInsert (), Insert を参照
ippsInterleave(), Interleave を参照
ippsInterpolate_G729(), Interpolate_G729 を参照
ippsInv, 12-3
ippsInvCbrt, 12-12
ippsInvSqrt, 12-8
ippsInvSqrt(), InvSqrt を参照
ippsJoin(), Join を参照
ippsJointStereoEncode_MP3(), JointStereoEncode_MP3 を参照
ippsJointStereoEncode_MP3_32s_I, 10-71
ippsLagWindow_G729(), LagWindow_G729 を参照
ippsLevinsonDurbin_G723(), LevinsonDurbin_G723 を参照
ippsLevinsonDurbin_G729(), LevinsonDurbin_G729 を参照
ippsLevinsonDurbin_GSMAMR, 9-118
ippsLinearPrediction(), LinearPrediction を参照
ippsLinearToMel(), LinearToMel を参照
ippsLinToALaw(), LinToALaw を参照
ippsLinToMuLaw(), LinToMuLaw を参照
ippsLn, 12-20
ippsLn(), Ln を参照
ippsLog10, 12-22
ippsLogAdd(), LogAdd を参照
ippsLogGauss(), LogGauss を参照
ippsLogGaussAdd(), LogGaussAdd を参照
ippsLogGaussAddMultiMix(), LogGaussAddMultiMix を参照
ippsLogGaussMax(), LogGaussMax を参照
ippsLogGaussMaxMultiMix(), LogGaussMaxMultiMix を参照
ippsLogGaussMixture(), LogGaussMixture を参照
ippsLogGaussMixtureSelect(), LogGaussMixtureSelect を参照
ippsLogGaussMultiMix(), LogGaussMultiMix を参照
ippsLogGaussSingle(), LogGaussSingle を参照
ippsLogSub(), LogSub を参照
ippsLogSum(), LogSum を参照
ippsLongTermPostFilter_G729(), LongTermPostFilter_G729 を参照
ippsLongTermPredict_AAC(), LongTermPredict_AAC を参照
ippsLongTermPredict_AAC_32s, 10-147
ippsLongTermReconstruct_AAC(), LongTermReconstruct_AAC を参照
ippsLongTermReconstruct_AAC_32s, 10-143
ippsLowercase(), Lowercase を参照
ippsLowHighFilter_Aurora(), LowHighFilter_Aurora を参照
ippsLPCToLSF_G723(), LPCToLSF_G723 を参照
ippsLPCToLSP_G729(), LPCToLSP_G729 を参照
ippsLPCToLSP_GSMAMR_16s, 9-119
ippsLPToCepstrum(), LPToCepstrum を参照
ippsLPToLSP(), LPToLSP を参照
ippsLPToReflection(), LPToReflection を参照
ippsLPToSpectrum(), LPToSpectrum を参照
ippsLSFDecodeErased_G729(), LSFDecodeErased_G729 を参照
ippsLSFDecode_G723(), LSFDecode_G723 を参照
ippsLSFDecode_G729(), LSFDecode_G729 を参照
ippsLSFQuant_G723(), LSFQuant_G723 を参照
ippsLSFQuant_G729(), LSFQuant_G729 を参照
ippsLSFToLPC_G723(), LSFToLPC_G723 を参照

intel®

ippsLSFToLSP_G729(), LSFToLSP_G729 を参照
ippsLShiftC(), LShiftC を参照
ippsLSPQuant_G729(), LSPQuant_G729 を参照
ippsLSPQuant_GSMAMR_16s, 9-123
ippsLSPToLPC_G729(), LSPToLPC_G729 を参照
ippsLSPToLPC_GSMAMR_16s, 9-121
ippsLSPToLSF_G729(), LSPToLSF_G729 を参照
ippsLtpUpdate_AAC(), LtpUpdate_AAC を参照
ippsLtpUpdate_AAC_32s, 10-150
ippsMagnitude(), Magnitude を参照
ippsMagSquared(), MagSquared を参照
ippsMahDist(), MahDist を参照
ippsMahDistMultiMix(), MahDistMultiMix を参照
ippsMahDistSingle(), MahDistSingle を参照
ippsMainSelect_VQ(), MainSelect_VQ を参照
ippsMakeFloat(), MakeFloat を参照
ippsMalloc, 3-4
ippsMatVecMul(), MatVecMul を参照
ippsMax(), Max を参照
ippsMaxEvery(), MaxEvery を参照
ippsMaxIndx(), MaxIndx を参照
ippsMaxOrder(), MaxOrder を参照
ippsMDCTFwd(), MDCTFwd、MDCTInv を参照
ippsMDCTFwd_AAC(), MDCTFwd_AAC を参照
ippsMDCTFwd_AAC_32s, 10-144
ippsMDCTFwdFree(), MDCTFwdFree、MDCTInvFree を参照
ippsMDCTFwdGetBufSize(), MDCTFwdGetBufSize、MDCTInvGetBufSize を参照
ippsMDCTFwdInitAlloc(), MDCTFwdInitAlloc、MDCTInvInitAlloc を参照
ippsMDCTFwd_MP3(), MDCTFwd_MP3 を参照
ippsMDCTFwd_MP3_32s, 10-64
ippsMDCTInv(), MDCTFwd、MDCTInv を参照
ippsMDCTInv_AAC(), MDCTInv_AAC を参照
ippsMDCTInv_AAC_32s16s, 10-137
ippsMDCTInvFree(), MDCTFwdFree、MDCTInvFree を参照
ippsMDCTInvGetBufSize(), MDCTFwdGetBufSize、MDCTInvGetBufSize を参照
ippsMDCTInvInitAlloc(), MDCTFwdInitAlloc、MDCTInvInitAlloc を参照
ippsMDCTInv_MP3(), MDCTInv_MP3 を参照
ippsMean(), Mean を参照
ippsMeanColumn(), MeanColumn を参照
ippsMeanVarAcc(), MeanVarAcc を参照
ippsMeanVarColumn(), MeanVarColumn を参照
ippsMelFBankGetSize(), MelFBankGetSize を参照
ippsMelFBankInit(), MelFBankInit を参照
ippsMelFBankInitAlloc(), MelFBankInitAlloc を参照
ippsMelFBankInitAlloc_Aurora(), MelFBankInitAlloc_Aurora を参照
ippsMelLinFBankInitAlloc(), MelLinFBankInitAlloc を参照
ippsMelToLinear(), MelToLinear を参照
ippsMin(), Min を参照
ippsMinEvery(), MinEvery を参照
ippsMinIndxt(), MinIndx を参照
ippsMinMax(), MinMax を参照
ippsMinMaxIndx(), MinMaxIndx を参照
ippsMove(), Move を参照
ippsMPMLQFixedCodebookSearch_G723(), MPMLQFixedCodebookSearch_G723 を参照
ippsMul(), Mul を参照
ippsMuLawToALaw(), MuLawToALaw を参照
ippsMuLawToLin(), MuLawToLin を参照
ippsMulC_NR(), MulC_NR を参照
ippsMulColumn(), MulColumn を参照
ippsMul_NR(), Mul_NR を参照
ippsMulPack(), MulPack を参照
ippsMulPackConj(), MulPackConj を参照
ippsMulPerm(), MulPerm を参照
ippsMulPowerC_NR(), MulPowerC_NR を参照
ippsNewVar(), NewVar を参照
ippsNoiseLessDecode_AAC, 10-148
ippsNoiseLessDecode_AAC(), NoiseLessDecode_AAC を参照
ippsNoiselessDecoder_LC_AAC, 10-119
ippsNoiselessDecoder_LC_AAC(), NoiselessDecoder_LC_AAC を参照
ippsNoiseSpectrumUpdate_Aurora(), NoiseSpectrumUpdate_Aurora を参照
ippsNorm(), Norm を参照
ippsNormalize(), Normalize を参照
ippsNormalizeColumn(), NormalizeColumn を参照
ippsNormalizeInRange(), NormalizeInRange を参照
ippsNormDiff(), NormDiff を参照

ippsNormEnergy(), NormEnergy を参照
ippsNot(), Not を参照
ippsNthMaxElement(), NthMaxElement を参照
ippsOpenLoopPitchSearchDTXVAD1_GSMAMR_16s, 9-132
ippsOpenLoopPitchSearchDTXVAD2_GSMAMR, 9-135
ippsOpenLoopPitchSearch_G723(), OpenLoopPitchSearch_G723 を参照
ippsOpenLoopPitchSearch_G729(), OpenLoopPitchSearch_G729 を参照
ippsOpenLoopPitchSearchNonDTX_GSMAMR_16s, 9-130
ippsOr(), Or を参照
ippsOrC(), OrC を参照
ippsOutProbPreCalc(), OutProbPreCalc を参照
ippsPackFrameHeader_MP3, 10-87
ippsPackFrameHeader_MP3(), PackFrameHeader_MP3 を参照
ippsPackScalefactors_MP3(), PackScalefactors_MP3 を参照
ippsPackScalefactors_MP3_8s1u, 10-81
ippsPackSideInfo_MP3, 10-88
ippsPackSideInfo_MP3(), PackSideInfo_MP3 を参照
ippsPeriodicity(), Periodicity を参照
ippsPeriodicityLSPE(), PeriodicityLSPE を参照
ippsPhase(), Phase、complex を参照
ippsPitchmarkToF0(), PitchmarkToF0 を参照
ippsPitchPostFilter_G723(), PitchPostFilter_G723 を参照
ippsPolarToCart(), PolarToCart、complex を参照
ippsPostFilter_GSMAMR_16s, 9-161
ippsPow, 12-14
ippsPow34(), Pow34 を参照
ippsPow43(), Pow43
ippsPowerSpectr(), PowerSpectr、complex を参照
ippsPowx, 12-16
ippsPreemphasize(), Preemphasize を参照
ippsPreemphasize_G729(), Preemphasize_G729 を参照
ippsPreemphasize_GSMAMR(), Preemphasize_GSMAMR を参照
ippsPreSelect_VQ(), PreSelect_VQ を参照
ippsPsych_MP3_16s, 10-66
ippsPsychoacousticModelTwo_MP3(), PsychoacousticModelTwo_MP3 を参照
ippsQRTransColumn(), QRTransColumn を参照
ippsQuantInv_AAC(), QuantInv_AAC を参照
ippsQuantInv_AAC_32s_I, 10-125
ippsQuantize_MP3(), Quantize_MP3 を参照
ippsQuantize_MP3_32s_I, 10-75
ippsQuantLSPDecode_GSMAMR_16s, 9-125
ippsRandGauss(), RandGauss を参照
ippsRandGauss_Direct(), RandGauss_Direct を参照
ippsRandGaussFree(), RandGaussFree を参照
ippsRandGaussGetSize(), RandGaussGetSize を参照
ippsRandGaussInit(), RandGaussInit を参照
ippsRandGaussInitAlloc(), RandGaussInitAlloc を参照
ippsRandomNoiseExcitation_G729B(), RandomNoiseExcitation_G729B を参照
ippsRandUniform_Direct(), RandUniform_Direct を参照
ippsRandUniformFree(), RandUniformFree を参照
ippsRandUniformGetSize(), RandUniformGetSize を参照
ippsRandUniformInit(), RandUniformInit を参照
ippsRandUniformInitAlloc(), RandUniformInitAlloc を参照
ippsRandUnifrom(), RandUnifrom を参照
ippsReal(), Real を参照
ippsRealToCplx(), RealToCplx を参照
ippsRecSqrt(), RecSqrt を参照
ippsReflectionToAR(), ReflectionToAR を参照
ippsReflectionToLP(), ReflectionToLP を参照
ippsReflectionToTilt(), ReflectionToTilt を参照
ippsRemove(), Remove を参照
ippsReplaceC(), ReplaceC を参照
ippsReQuantize_MP3(), ReQuantize_MP3 を参照
ippsResamplePolyphase(), ResamplePolyphase を参照
ippsResamplePolyphaseFree(), ResamplePolyphaseFree を参照
ippsResamplePolyphaseInitAlloc(), ResamplePolyphaseInitAlloc を参照
ippsResetFDP(), ResetFDP を参照
ippsResetFDPGroup(), ResetFDPGroup を参照
ippsResetFDP_SFB(), ResetFDP_SFB を参照
ippsResidualFilter_Aurora(), ResidualFilter_Aurora を参照

intel®

ippsResidualFilter_G729(), ResidualFilter_G729 を参照
ippsRShiftC(), RShiftC を参照
ippsSampleDown(), SampleDown を参照
ippsSampleUp(), SampleUp を参照
ippsScaleLM(), ScaleLM を参照
ippsSchur(), Schur を参照
ippsSet(), Set を参照
ippsShortTermPostFilter_G729(), ShortTermPostFilter_G729 を参照
ippsSignChangeRate(), SignChangeRate を参照
ippsSin, 12-25
ippsSinC(), SinC を参照
ippsSinCos, 12-27
ippsSinh, 12-40
ippsSmoothedPowerSpectrum_Aurora(), SmoothedPowerSpectrum_Aurora を参照
ippsSortAscend(), SortAscend を参照
ippsSortDescend(), SortDescend を参照
ippsSplitC(), SplitC を参照
ippsSplitVQ(), SplitVQ を参照
ippsSqr(), Sqr を参照
ippsSqrt, 12-7
ippsSqrt(), Sqrt を参照
ippsStdDev(), StdDev を参照
ippsStepSizeUpdateAECNLMS(), StepSizeUpdateAECNLMS を参照
ippsSub(), Sub を参照
ippsSubC(), SubC を参照
ippsSubCRev(), SubCRev を参照
ippsSubRow(), SubRow を参照
ippsSum(), Sum を参照
ippsSumColumn(), SumColumn を参照
ippsSumColumnAbs(), SumColumnAbs を参照
ippsSumColumnSqr(), SumColumnSqr を参照
ippsSumLn(), SumLn を参照
ippsSumMeanVar(), SumMeanVar を参照
ippsSumRow(), SumRow を参照
ippsSumRowAbs(), SumRowAbs を参照
ippsSumRowSqr(), SumRowSqr を参照
ippsSVD(), SVD を参照
ippsSwapBytes(), SwapBytes を参照
ippsSynthesisFilter_G723(), SynthesisFilter_G723 を参照
ippsSynthesisFilter_G729(), SynthesisFilter_G729 を参照
ippsSynthPQMF_MP3(), SynthPQMF_MP3 を参照
ippsTabsCalculation_Aurora(), TabsCalculation_Aurora を参照
ippsTan, 12-29
ippsTanh, 12-41
ippStaticFree(), StaticFree を参照
ippStaticInit(), StaticInit を参照
ippStaticInitBest(), StaticInitBest を参照
ippStaticInitCpu(), StaticInitCpu を参照
ippsThreshold(), Threshold を参照
ippsThreshold_GT(), Threshold_GT を参照
ippsThreshold_GTVal(), Threshold_GTVal を参照
ippsThreshold_LT(), Threshold_LT を参照
ippsThreshold_LTInv(), Threshold_LTInv を参照
ippsThreshold_LTVal(), Threshold_LTVal を参照
ippsThreshold_LTValGTVal(), Threshold_LTValGTVal を参照
ippsTiltCompensation_G723(), TiltCompensation_G723 を参照
ippsTiltCompensation_G729(), TiltCompensation_G729 を参照
ippsToeplizMatrix_G723(), ToeplizMatrix_G723 を参照
ippsToeplizMatrix_G729(), ToeplizMatrix_G729 を参照
ippsTone_Direct(), Tone_Direct を参照
ippsToneFree(), ToneFree を参照
ippsToneGetStateSizeQ15(), ToneGetStateSizeQ15 を参照
ippsToneInitAllocQ15(), ToneInitAllocQ15 を参照
ippsToneInitQ15(), ToneInitQ15 を参照
ippsToneQ15(), ToneQ15 を参照
ippsTriangle_Direct(), Triangle_Direct を参照
ippsTriangleFree(), TriangleFree を参照
ippsTriangleGetStateSizeQ15(), TriangleGetStateSizeQ15 を参照
ippsTriangleInitAllocQ15(), TriangleInitAllocQ15 を参照
ippsTriangleInitQ15(), TriangleInitQ15 を参照
ippsTriangleQ15(), TriangleQ15 を参照
ippsTrimC(), TrimC を参照
ippsTrimCAny(), TrimCAny を参照
ippsUnitCurve(), UnitCurve を参照
ippsUnpackADIFHeader_AAC, 10-114

ippsUnpackADIFHeader_AAC(), UnpackADIFHeader_AAC を参照
ippsUnpackADTSFrameHeader_AAC, 10-115
ippsUnpackADTSFrameHeader_AAC(), UnpackADTSFrameHeader_AAC を参照
ippsUnpackFrameHeader_MP3(), UnpackFrameHeader_MP3 を参照
ippsUnpackScaleFactors_MP3(), UnpackScaleFactors_MP3 を参照
ippsUnpackSideInfo_MP3(), UnpackSideInfo_MP3 を参照
ippsUpdateGConst(), UpdateGConst を参照
ippsUpdateLinear(), UpdateLinear を参照
ippsUpdateMean(), UpdateMean を参照
ippsUpdateNoisePSDMCRA(), UpdateNoisePSDMCRA を参照
ippsUpdatePathMetricsDV(), UpdatePathMetricsDV を参照
ippsUpdatePower(), UpdatePower を参照
ippsUpdateVar(), UpdateVar を参照
ippsUpdateWeight(), UpdateWeight を参照
ippsUppercase(), Uppercase を参照
ippsVAD1_GSMAMR_16s, 9-152
ippsVAD2_GSMAMR_16s, 9-154
ippsVADDecision_Aurora(), VADDecision_Aurora を参照
ippsVADFlush_Aurora(), VADFlush_Aurora を参照
ippsVADGetBufSize_Aurora(), VADGetBufSize_Aurora を参照
ippsVADInit_Aurora(), VADInit_Aurora を参照
ippsVarColumn(), VarColumn を参照
ippsVecMatMul(), VecMatMul を参照
ippsVectorJaehne(), VectorJaehne を参照
ippsVectorRamp(), VectorRamp を参照
ippsVectorReconstruction_VQ(), VectorReconstruction_VQ を参照
ippsVQ(), VQ を参照
ippsVQSingle_Sort(), VQSingle_Sort を参照
ippsVQSingle_Thresh(), VQSingle_Thresh を参照
ippsWaveProcessing_Aurora(), WaveProcessing_Aurora を参照
ippsWeightedMeanColumn(), WeightedMeanColumn を参照
ippsWeightedMeanVarColumn(), WeightedMeanVarColumn を参照
ippsWeightedSum(), WeightedSum を参照
ippsWeightedVarColumn(), WeightedVarColumn を参照
ippsWienerFilterDesign_Aurora(), WienerFilterDesign_Aurora を参照
ippsWinBartlett(), WinBartlett を参照
ippsWinBlackman(), WinBlackman を参照
ippsWinHamming(), WinHamming を参照
ippsWinHann(), WinHann を参照
ippsWinKaiser(), WinKaiser を参照
ippsWTFwd(), WTFwd を参照
ippsWTFwdFree(), WTFwdFree を参照
ippsWTFwdGetDlyLine(), WTFwdGetDlyLine を参照
ippsWTFwdInitAlloc(), WTFwdInitAlloc を参照
ippsWTFwdSetDlyLine(), WTFwdSetDlyLine を参照
ippsWTHaarFwd(), WTHaarFwd を参照
ippsWTHaarInv(), WTHaarInv を参照
ippsWTInv(), WTInv を参照
ippsWTInvFree(), WTInvFree を参照
ippsWTInvGetDlyLine(), WTInvGetDlyLine を参照
ippsWTInvInitAlloc(), WTInvInitAlloc を参照
ippsWTInvSetDlyLine(), WTInvSetDlyLine を参照
ippsXor(), Xor を参照
ippsXorC(), XorC を参照
ippsZero(), Zero を参照
ippsZeroMean(), ZeroMean を参照

J

Join, 5-53
JointStereoEncode_MP3, 10-71

L

LagWindow_G729, 9-37
LevinsonDurbin_G723, 9-86
LevinsonDurbin_G729, 9-24
LinearPrediction, 8-22
LinearToMel, 8-40
LinToALaw, 10-53
LinToMuLaw, 10-51
Ln, 5-42, 12-20
Log10, 12-22
LogAdd, 8-87
LogGauss, 8-98

intel®

LogGaussAdd, 8-109
LogGaussAddMultiMix, 8-113
LogGaussMax, 8-103
LogGaussMaxMultiMix, 8-107
LogGaussMixture, 8-115
LogGaussMixtureSelect, 8-118
LogGaussMultiMix, 8-101
LogGaussSingle, 8-94
LogSub, 8-88
LogSum, 8-89
LongTermPostFilter_G729, 9-63
LongTermPredict_AAC, 10-147
LongTermReconstruct_AAC, 10-143
Lowercase, 11-14
LowHighFilter_Aurora, 8-200
LP 係数 , 9-32
LPCToLSF_G723, 9-87
LPCToLSP_G729, 9-26
LPToCepstrum, 8-27
LPToLSP, 8-36
LPToReflection, 8-29
LPToSpectrum, 8-26
LSFDecodeErased_G729, 9-30
LSFDecode_G723, 9-89
LSFDecode_G729, 9-29
LSFQuant_G723, 9-90
LSFQuant_G729, 9-28
LSFToLPC_G723, 9-88
LSFToLSP_G729, 9-27
LShiftC, 5-11
LSP 係数 , 9-36
LSPQuant_G729, 9-32
LSPToLPC_G729, 9-31
LSPToLSF_G729, 9-36
LtpUpdate_AAC, 10-150

M

MA 予測子 , 9-29
Magnitude, 5-56
MagSquared, 5-58
MahDist, 8-91
MahDistMultiMix, 8-93
MahDistSingle, 8-90
MainSelect_VQ, 10-44
MakeFloat, 10-12
MatVecMul, 8-17
Max, 5-101
MaxEvery, 5-117
MaxIndx, 5-102
MaxOrder, 5-80
MDCTFwd, MDCTInv, 10-16
MDCTFwd_AAC, 10-144
MDCTFwdFree, MDCTInvFree, 10-14
MDCTFwdGetBufSize, MDCTInvGetBufSize, 10-15
MDCTFwdInitAlloc, MDCTInvInitAlloc, 10-13
MDCTFwd_MP3, 10-64
MDCTInv_AAC, 10-137
MDCTInv_MP3, 10-102
Mean, 5-107
MeanColumn, 8-130
MeanVarAcc, 8-143
MeanVarColumn, 8-133
MelFBankGetSize, 8-42
MelFBankInit, 8-43
MelFBankInitAlloc, 8-44
MelFBankInitAlloc_Aurora, 8-196
MelLinFBankInitAlloc, 8-47
MelToLinear, 8-39
Min, 5-103
MinEvery, 5-117
MinIndx, 5-104
MinMax, 5-105
MinMaxIndx, 5-106
MMX テクノロジ , 1-1
Move, 4-4
MP3 オーディオ・エンコーダ
    心理音響モデル , 10-59
    心理音響モデル構造体 , 10-59
    ビット貯蓄構造体 , 10-60
MP3 オーディオ・エンコーダ API
    列挙型 , 10-61
オーディオ・エンコーダ
    MP3 オーディオ・エンコーダを参照
MP3 オーディオ・デコーダ , 10-91
MP3 オーディオ・デコーダ関数 , 10-91 – 10-106
    MDCTInv_MP3, 10-102

HuffmanDecode_MP3, 10-98
ReQuantize_MP3, 10-100
SynthPQMF_MP3, 10-104
UnpackFrameHeader_MP3, 10-93
UnpackScaleFactors_MP3, 10-95
UnpackSideInfo_MP3, 10-94
MP3、データ構造体 , 10-57
MP3、マクロおよび定数の定義 , 10-57
MPEG-2
ADIF ヘッダ構造体 , 10-108
ADTS フレーム・ヘッダ構造体 , 10-109
グローバル・マクロ , 10-108
単一チャネル・サイド情報構造体 , 10-110
チャネル情報構造体 , 10-112
チャネルのペア要素構造体 , 10-112
MPEG-4
AAC LTP 構造体 , 10-112
AAC TNS 構造体 , 10-111
AAC スケーラブル拡張要素ヘッダ構造体 , 10-111
AAC スケーラブル・メイン要素構造体 , 10-110
MPMLQFixedCodebookSearch_G723, 9-96
Mul, 5-21
MuLawToALaw, 10-54
MuLawToLin, 10-50
MulC, 5-19
MulC_NR, 9-7
MulColumn, 8-164
Mul_NR, 9-6
MulPack, 7-12
MulPackConj, 7-14
MulPerm, 7-12
MulPowerC_NR, 9-8

N

NewVar, 8-65
NoiseLessDecode_AAC, 10-148
NoiselessDecoder_LC_AAC, 10-119
NoiseSpectrumUpdate_Aurora, 8-193
Norm, 5-110
Normalize, 5-47
NormalizeColumn, 8-139
NormalizeInRange, 8-141
NormDiff, 5-112
NormEnergy, 8-62
Not, 5-10
NthMaxElement, 8-15

O

OpenLoopPitchSearch_G723, 9-91
OpenLoopPitchSearch_G729, 9-38
Or, 5-7
OrC, 5-6
OutProbPreCalc, 8-156

P

Pack 形式 , 7-4
PackFrameHeader_MP3, 10-87
PackScalefactors_MP3, 10-81
PackSideInfo_MP3, 10-88
Periodicity, 8-242
PeriodicityLSPE, 8-241
Perm 形式 , 7-4
Phase、complex, 5-59
PitchmarkToF0, 8-34
PitchPostFilter_G723, 9-110
PolarToCart、complex, 5-79
Pow, 12-14
Pow34, 10-7
Pow43, 10-9
PowerSpectr、complex, 5-60
Powx, 12-16
Preemphasize, 5-80
Preemphasize_G729, 9-78
Preemphasize_GSMAMR, 9-151
PreSelect_VQ, 10-42
PsychoacousticModelTwo_MP3, 10-66

Q

QRTransColumn, 8-162
QuantInv_AAC, 10-125
Quantize_MP3, 10-75

R

RandGauss, 4-31
RandGauss_Direct, 4-32
RandGaussFree, 4-29

RandGaussGetSize, 4-29
RandGaussInit, 4-30
RandGaussInitAlloc, 4-28
RandomNoiseExcitation_G729B, 9-81
RandUniform_Direct, 4-27
RandUniformFree, 4-24
RandUniformGetSize, 4-25
RandUniformInit, 4-24
RandUniformInitAlloc, 4-23
RandUnifrom, 4-26
Real, 5-61
RealToCplx, 5-63
RecSqrt, 8-66
ReflectionToAR, 8-31
ReflectionToLP, 8-30
ReflectionToTilt, 8-32
Remove, 11-6
ReplaceC, 11-12
ReQuantize_MP3, 10-100
ResamplePolyphase, 8-189
ResamplePolyphaseFree, 8-189
ResamplePolyphaseInitAlloc, 8-187
ResetFDP, 10-23
ResetFDPGroup, 10-25
ResetFDP_SFB, 10-24
ResidualFilter_Aurora, 8-197
ResidualFilter_G729, 9-60
RShiftC, 5-12

S

SampleDown, 5-120
SampleUp, 5-118
ScaleLM, 8-86
Schur, 8-25
Set, 4-5
Shift 関数 , 5-11
ShortTermPostFilter_G729, 9-66
SignChangeRate, 8-21
SIMD 命令 , 1-1
Sin, 12-25
SinC, 8-148
SinCos, 12-27
Sinh, 12-40
SmoothedPowerSpectrum_Aurora, 8-192
SortAscend, 5-49
SortDescend, 5-49
SplitC, 11-18
SplitVQ, 8-183
Sqr, 5-35
Sqrt, 5-37, 12-7
StaticInit, 3-13
StaticInitBest, 3-14
StaticInitCpu, 3-15
StdDev, 5-109
StepSizeUpdateAECNLMS, 8-235
Sub, 5-26
SubC, 5-23
SubCRev, 5-25
SubRow, 8-11
Sum, 5-100
SumColumn, 8-8
SumColumnAbs, 8-165
SumColumnSqr, 8-166
SumLn, 5-45
SumMeanVar, 8-63
SumRow, 8-9
SumRowAbs, 8-167
SumRowSqr, 8-167
SVD, 8-168
SwapBytes, 5-50
SynthesisFilter_G723, 9-17, 9-104
SynthesisFilter_G729, 9-61
SynthPQMF_MP3, 10-104

T

TabsCalculation_Aurora, 8-196
Tan, 12-29
Tanh, 12-41
Threshold, 5-64
Threshold_GT, 5-68
Threshold_GTVal, 5-71
Threshold_LT, 5-68
Threshold_LTInv, 5-75
Threshold_LTVal, 5-71
Threshold_LTValGTVal, 5-71
TiltCompensation_G723, 9-105

TiltCompensation_G729, 9-68
ToeplizMatrix_G723, 9-97
ToeplizMatrix_G729, 9-49
Tone, 4-11
Tone_Direct, 4-13
ToneFree, 4-9
ToneGetStateSizeQ15, 4-9
ToneInitAllocQ15, 4-8
ToneInitQ15, 4-10
ToneQ15, 4-11
Triangle, 4-20
Triangle_Direct, 4-21
TriangleFree, 4-17
TriangleGetStateSizeQ15, 4-17
TriangleInitAllocQ15, 4-15
TriangleInitQ15, 4-18
TriangleQ15, 4-19
TrimC, 11-10
TrimCAny, 11-11

U

UnitCurve, 8-35
UnpackADIFHeader_AAC, 10-114
UnpackADTSFrameHeader_AAC, 10-115
UnpackFrameHeader_MP3, 10-93
UnpackScaleFactors_MP3, 10-95
UnpackSideInfo_MP3, 10-94
UpdateGConst, 8-155
UpdateLinear, 6-8
UpdateMean, 8-152
UpdateNoisePSDMCRA, 8-221
UpdatePathMetricsDV, 5-86
UpdatePower, 6-9
UpdateVar, 8-153
UpdateWeight, 8-154
Uppercase, 11-13

V

VADDecision_Aurora, 8-209
VADFlush_Aurora, 8-209
VADGetBufSize_Aurora, 8-208
VADInit_Aurora, 8-208
VarColumn, 8-131
VecMatMul, 8-16
VectorJaehne, 4-33
VectorRamp, 4-34
VectorReconstruction_VQ, 10-48
VQSingle_Sort, 8-182
VQSingle_Thresh, 8-182

W

WaveProcessing_Aurora, 8-198
WeightedMeanColumn, 8-134
WeightedMeanVarColumn, 8-137
WeightedSum, 8-170
WeightedVarColumn, 8-135
WienerFilterDesign_Aurora, 8-194
WinBartlett, 5-88
WinBlackman, 5-90
WinHamming, 5-94
WTFwd, 7-63
WTFwdFree, 7-62
WTFwdGetDlyLine, 7-67
WTFwdInitAlloc, 7-60
WTFwdSetDlyLine, 7-67
WTHaarFwd, 7-55
WTHaarInv, 7-55
WTInv, 7-69
WTInvFree, 7-62
WTInvGetDlyLine, 7-72
WTInvInitAlloc, 7-60
WTInvSetDlyLine, 7-72

X

Xor, 5-9
XorC, 5-8

Z

Zero, 4-6
ZeroMean, 8-18

あ

アダプティブ予測機構 , 10-17
アダプティブ・コードブック , 9-58
アダプティブ・フィルタ , 10-17

intel®

圧伸関数 , 10-49 – 10-56
ALawToLin, 10-52
bALawToMuLaw, 10-55
LinToALaw, 10-53
LinToMuLaw, 10-51
MuLawToALaw, 10-54
MuLawToLin, 10-50

い

一様分布関数
RandUniformGetSize, 4-25
RandUniformInit, 4-24
RandUniformInitAlloc, 4-23
RandUnifrom, 4-26
RandUnifrom_Direct, 4-27
RandUniformFree, 4-24
インターリーブ形式から複数行形式への変換関数 , 10-5
インテル ® パフォーマンス・プリミティブ・ソフトウェア , 1-1
インテル ® パフォーマンス・ライブラリ・スイート , 2-1

う

ウェーブレット変換関数 , 7-52 – 7-74
WTFwd, 7-63
WTFwdFree, 7-62
WTFwdGetDlyLine, 7-67
WTFwdInitAlloc, 7-60
WTFwdSetDlyLine, 7-67
WTHaarFwd, 7-55
WTHaarInv, 7-55
WTInv, 7-69
WTInvFree, 7-62
WTInvGetDlyLine, 7-72
WTInvInitAlloc, 7-60
WTInvSetDlyLine, 7-72

え

エネルギー、平均 , 9-55
エラー・レポート , 2-11

お

オーディオ符号化関数
ApplySF_I, 10-11
オーディオ符号化 , 10-1
オーディオ符号化関数
BuildHDT, 10-29
BuildHET, 10-33
BuildHET_VLC, 10-37
CalcSF, 10-10
CdbkFree, 10-42
CdbkInitAlloc, 10-41
CountBits, 10-38
DecodeVLC, 10-30
Deinterleave, 10-6
EncodeBlock, 10-39
EncodeVLC, 10-35
FDPFree, 10-23
FDPFwd, 10-25
FDPInitAlloc, 10-22
FDPInv, 10-26
FIRBlockFree, 10-19
FIRBlockInitAlloc, 10-18
FIRBlockOne, 10-19
GetSizeHDT, 10-28
GetSizeHET, 10-32
GetSizeHET_VLC, 10-36
HuffmanCountBits, 10-34
IndexSelect_VQ, 10-46
Interleave, 10-5
MainSelect_VQ, 10-44
MakeFloat, 10-12
MDCTFwd, MDCTInv, 10-16
MDCTFwdFree, MDCTInvFree, 10-14
MDCTFwdGetBufSize, MDCTInvGetBufSize, 10-15
MDCTFwdInitAlloc, MDCTInvInitAlloc, 10-13
Pow34, 10-7
Pow43, 10-9
PreSelect_VQ, 10-42
ResetFDP, 10-23
ResetFDPGroup, 10-25
ResetFDP_SFB, 10-24
VectorReconstruction_VQ, 10-48
オーロラ関数
BlindEqualization_Aurora, 8-202
EvalDelta_Aurora, 8-203
HighBandCoding_Aurora, 8-201
LowHighFilter_Aurora, 8-200
MelFBankInitAlloc_Aurora, 8-196
NoiseSpectrumUpdate_Aurora, 8-193
ResidualFilter_Aurora, 8-197
SmoothedPowerSpectrum_Aurora, 8-192
TabsCalculation_Aurora, 8-196
VADDecision_Aurora, 8-209
VADFlush_Aurora, 8-209
VADGetBufSize_Aurora, 8-208

VADInit_Aurora, 8-208
WaveProcessing_Aurora, 8-198
WienerFilterDesign_Aurora, 8-194
帯域幅の拡張 , 9-87
重み付け行列 , 9-90
音響エコー・キャンセラ関数
CoefUpdateAECNLMS, 8-233
ControllerGetSizeAEC, 8-236
ControllerInitAEC, 8-237
ControllerUpdateAEC, 8-238
FilterAECNLMS, 8-232
StepSizeUpdateAECNLMS, 8-235
音声アクティビティ検出（VAD）関数
FindPeaks, 8-240
Periodicity, 8-242
PeriodicityLSPE, 8-241
音声コーデック
GSM-AMR, 9-112
音声コーデック関数
G729 基本関数
DotProd_G729, 9-20
Interpolate_G729, 9-21
LP 分析関数
AutoCorr_G723, 9-83
AutoCorr_G729, 9-23
AutoCorrLagMax, 9-13
AutoCorr_NormE, 9-14
AutoCorr_NormE_G723, 9-85
CrossCorr, 9-15
GainControl_G723, 9-100
HarmonicNoiseSubtract_G723, 9-107
HighPassFilter_G723, 9-102
IIR16s_G723, 9-102
IIR16s_G729, 9-74
LevinsonDurbin_G723, 9-86
LevinsonDurbin_G729, 9-24
LPCToLSF_G723, 9-87
LPCToLSP_G729, 9-26
LSFDecodeErased_G729, 9-30
LSFDecode_G723, 9-89
LSFDecode_G729, 9-29
LSFQuant_G729, 9-28
LSFToLPC_G723, 9-88
LSFToLSP_G729, 9-27
LSPQuant_G729, 9-32
LSPToLPC_G729, 9-31
LSPToLSF_G729, 9-36
共通の関数
AutoScale, 9-9
ConvPartial, 9-5
DotProdAutoScale, 9-10
InvSqrt, 9-11
MulC_NR, 9-7
Mul_NR, 9-6
MulPowerC_NR, 9-8
ゲイン量子化関数
DecodeGain_G729, 9-50
GainControl_G729, 9-52
GainQuant_G723, 9-99
GainQuant_G729, 9-54
コードブック検索関数
AdaptiveCodebookContribution_G729, 9-56
AdaptiveCodebookGain_G729, 9-57
DecodeAdaptiveVector_G729, 9-44
FixedCodebookSearch_G729, 9-45
LagWindow_G729, 9-37
OpenLoopPitchSearch_G723, 9-91
OpenLoopPitchSearch_G729, 9-38
ToeplizMatrix_G729, 9-49
AdaptiveCodebookSearch_G729, 9-41
フィルタ関数
ACELPFixedCodebookSearch_G723, 9-93
AdaptiveCodebookSearch_G723, 9-94
DecodeAdaptiveVector_G723, 9-108
HarmonicFilter, 9-70
HarmonicSearch_G723, 9-106
HighPassFilter_G729, 9-73
HighPassFilterInit_G729, 9-72
HighPassFilterSize_G729, 9-72
LongTermPostFilter_G729, 9-63
LSFQuant_G723, 9-90
MPMLQFixedCodebookSearch_G723, 9-96
PitchPostFilter_G723, 9-110
Preemphasize_G729, 9-78, 9-151
RandomNoiseExcitation_G729B, 9-81
ResidualFilter_G729, 9-60
ShortTermPostFilter_G729, 9-66
SynthesisFilter_G723, 9-17, 9-104
SynthesisFilter_G729, 9-61
TiltCompensation_G723, 9-105
TiltCompensation_G729, 9-68
ToeplizMatrix_G723, 9-97
音声認識の基本算術関数 , 8-6 – 8-15
AddAllRowSum, 8-7
BlockDMatrixFree, 8-14
BlockDMatrixInitAlloc, 8-13
CopyColumn_Indirect, 8-12
MatVecMul, 8-17
SubRow, 8-11
SumColumn, 8-8
SumRow, 8-9

VecMatMul, 8-16
音声、合成 , 9-61
オンライン版 , 1-6

か

概念
IPP の , 2-1
IPP の構造体 , 2-7
ガウス分布関数
RandGauss, 4-31
RandGauss_Direct, 4-32
RandGaussFree, 4-29
RandGaussGetSize, 4-29
RandGaussInit, 4-30
RandGaussInitAlloc, 4-28
仮数変換およびスケーリング関数 , 10-11
関数の説明 , 1-6
関数の命名 , 2-2
関連資料 , 1-6

き

規則
字体 , 1-7
信号名 , 1-7
命名 , 1-7
共通関数
AlignPtr, 3-11
CoreGetCpuClocks, 3-8
GetCpuFreqMhz, 3-8
CoreGetCpuType, 3-7
CoreGetStatusString, 3-6
CoreSetFlushToZero, 3-9
ippFree, 3-12
ippMalloc, 3-11
CoreSetDenormAreZeros, 3-10

く

クロスアーキテクチャの統一 , 1-2
クロスプラットフォーム・アプリケーション , 2-1

け

ゲイン
推定 , 9-99
アダプティブ・コードブック・ベクトルの , 9-58
最適な , 9-55

こ

コードブック , 9-29
インデックス , 9-32
検索 , 9-56
固定 , 9-51
固定精度算術関数
三角関数
Acos, 12-30
Asin, 12-32
Atan, 12-34
Atan2, 12-36
Cos, 12-23
Sin, 12-25
SinCos, 12-27
Tan, 12-29
指数関数と対数関数
Exp, 12-18
Ln, 12-20
Log10, 12-22
双曲線
Acosh, 12-43
Asinh, 12-45
Atanh, 12-46
Cosh, 12-38
Sinh, 12-40
Tanh, 12-41
特殊
Erf, 12-48
Erfc, 12-50
累乗と根
Cbrt, 12-10
Div, 12-5
Inv, 12-3
InvCbrt, 12-12
InvSqrt, 12-8
Pow, 12-14
Powx, 12-16
Sqrt, 12-7
固定フィルタ・バンクのウェーブレット変換 , 7-55 – 7-59

さ

サポートされるプラットフォーム , 1-2
算術関数 , 5-14 – 5-48
10Log10, 5-44
Abs, 5-33
Add, 5-16

AddC, 5-14
AddProduct, 5-18
Arctan, 5-46
Cubrt, 5-39
Div, 5-31
DivC, 5-28
DivCRev, 5-30
Exp, 5-40
Ln, 5-42
Mul, 5-21
MulC, 5-19
Sqr, 5-35
Sqrt, 5-37
Sub, 5-26
SubC, 5-23
SubCRev, 5-25
SumLn, 5-45
サンプリング関数 , 5-118 – 5-122
SampleDown, 5-120
SampleUp, 5-118
サンプル・コード , 2-2
残留信号 , 9-60

し

自己相関、音声コーデック内の , 9-23
字体の規則 , 1-7
周波数領域予測 , 10-21
順方向変形拡散コサイン変換 (MDCT), 10-13
シングルレート FIR LMS フィルタ関数
FIRLMS, 6-58
FIRLMSFree, 6-55
FIRLMSGetDlyLine, 6-57
FIRLMSGetTaps, 6-56
FIRLMSInitAlloc, 6-54
FIRLMSOne_Direct, 6-61
FIRLMSSetDlyLine, 6-57
信号名の規則 , 1-7
振幅比 , 9-101

す

スケール係数計算 , 10-10
スケール係数バンド , 10-11
ストリーミング SIMD 拡張命令（SSE）, 1-1
スペクトル値の復元 , 10-11
スペクトル・データ・プレ量子化 , 10-7

せ

精度、算術関数の , 12-1
線形予測分析 , 9-23
線スペクトル周波数 , 9-28

た

多項式係数 , 9-87
たたみ込み関数と相関関数 , 6-1 – 6-8
Conv, 6-2
ConvCyclic, 6-3
CrossCorr, 6-6
UpdateLinear, 6-8
UpdatePower, 6-9

ち

超越数値演算関数 , 12-1

て

ディスパッチャ関数
StaticInit, 3-13
StaticInitBest, 3-14
StaticInitCpu, 3-15
ティルト , 9-69

と

導関数 , 8-68 – 8-80
AccCovarianceMatrix, 8-67
CopyColumn, 8-69
Delta, 8-72
DeltaDelta, 8-76
EvalDelta, 8-70
統計関数 , 5-100 – 5-116
DotProd, 5-114
Max, 5-101
MaxEvery, 5-117
MaxIndx, 5-102
Mean, 5-107
Min, 5-103
MinEvery, 5-117
MinIndx, 5-104
MinMax, 5-105
MinMaxIndx, 5-106
Norm, 5-110
Normalize, 5-47
NormDiff, 5-112
Phase, complex, 5-59

PowerSpectr, complex, 5-60
StdDev, 5-109
Sum, 5-100

トーン生成関数 , 4-11
Tone_Direct, 4-13
ToneFree, 4-9
ToneGetStateSizeQ15, 4-9
ToneInitAllocQ15, 4-8
ToneInitQ15, 4-10
ToneQ15, 4-11

特殊ベクトル関数
VectorJaehne, 4-33
VectorRamp, 4-34

特徴処理関数 , 8-18 – 8-84
CepstrumToLP, 8-28
CompensateOffset, 8-19
CopyWithPadding, 8-41
DCTLifter, 8-59
DCTLifterFree, 8-58
DCTLifterGetSize_MulC0, 8-55
DCTLifterInitAlloc, 8-57
DCTLifterInit_MulC0, 8-56
Durbin, 8-23
EmptyFBankInitAlloc, 8-49
EvalFBank, 8-54
FBankFree, 8-50
FBankGetCenters, 8-50
FBankGetCoeffs, 8-52
FBankSetCenters, 8-51
FBankSetCoeffs, 8-53
LinearPrediction, 8-22
LinearToMel, 8-40
LPToCepstrum, 8-27
LPToLSP, 8-36
LPToReflection, 8-29
LPToSpectrum, 8-26
MelFBankGetSize, 8-42
MelFBankInit, 8-43
MelFBankInitAlloc, 8-44
MelLinFBankInitAlloc, 8-47
MelToLinear, 8-39
PitchmarkToF0, 8-34
ReflectionToAR, 8-31
ReflectionToLP, 8-30
ReflectionToTilt, 8-32
Schur, 8-25
SignChangeRate, 8-21
UnitCurve, 8-35
ZeroMean, 8-18

特定周波数用 DFT（Goertzel）関数 , 7-39 – 7-42
Goertz, 7-39
GoertzTwo, 7-41

トライアングル生成関数 , 4-20
Triangle_Direct, 4-21
TriangleFree, 4-17
TriangleGetStateSizeQ15, 4-17
TriangleInitAllocQ15, 4-15
TriangleInitQ15, 4-18
TriangleQ15, 4-19

に

二項窓 , 9-84

は

バージョン情報関数 , 3-2

ハードウェアとソフトウェアの要件 , 1-2

パックド・データのアンパック , 7-5 – 7-11
ConjCcs, 7-9
ConjPack, 7-7
ConjPerm, 7-5

パックド・データの乗算 , 7-11 – 7-13
MulPack, 7-12
MulPackConj, 7-14
MulPerm, 7-12

ハフマン・アルゴリズム関数 , 10-27

ハミング窓 , 9-84

ひ

非整数遅延 , 9-42

ビタビ・デコーダ関数
BuildSymblTableDV4D, 5-86
CalcStatesDV, 5-85
GetVarPointDV, 5-84
UpdatePathMetricsDV, 5-86

ピッチ , 9-38
オープン・ループ , 9-92
クローズ・ループ , 9-95

ピッチ超解像度関数
CrossCorrCoeff, 8-82
CrossCorrCoeffDecim, 8-81
CrossCorrCoeffInterpolation, 8-83

ビットストリーム , 10-10

表記の規則 , 1-7

ヒルベルト変換関数
Hilbert, 7-50
HilbertFree, 7-49

HilbertInitAlloc, 7-49

ふ

フィルタ
逆 , 9-67
短期 , 9-67
長期 , 9-64
フィルタリング関数 , 6-10 – 6-102
プリフィックス、関数名の , 1-8
プレ量子化 , 10-7
ブロック・フィルタリング , 10-17

へ

並列処理 , 1-1
ベクトル初期化関数
Copy, 4-2
Move, 4-4
Set, 4-5
Zero, 4-6
ベクトル量子化関数 , 8-171 – 8-186
CdbkFree, 8-178
CdbkGetSize, 8-174
CdbkInit, 8-175
CdbkInitAlloc, 8-177
FormVector, 8-171
FormVectorVQ, 8-185
GetCdbkSize, 8-179
GetCodebook, 8-180
SplitVQ, 8-183
VQ, 8-181
ベクトル相関関数
AutoCorr, 6-4
ベクトル量子化 , 9-33
変換関数 , 5-48 – 5-81
CartToPolar、complex, 5-77
Conj, 5-54
ConjFlip, 5-55
Convert, 5-51
CplxToReal, 5-64
Flip, 5-81
Join, 5-53
Magnitude, 5-56
MagSquared, 5-58
MaxOrder, 5-80
PolarToCart, complex, 5-79
Preemphasize, 5-80
Real, 5-61
RealToCplx, 5-63
SortAscend, 5-49
SortDescend, 5-49
SwapBytes, 5-50
Threshold, 5-64
Threshold_GT, 5-68
Threshold_GTVal, 5-71
Threshold_LT, 5-68
Threshold_LTInv, 5-75
Threshold_LTVal, 5-71
Threshold_LTValGTVal, 5-71
Imag, 5-62
変換サポート関数 , 7-3 – 7-14

ほ

ポスト・フィルタ , 9-110
ポリフェーズ関数
ResamplePolyphase, 8-189
ResamplePolyphaseFree, 8-189
ResamplePolyphaseInitAlloc, 8-187
本書について , 1-4
本書の構成 , 1-5
本書の対象読者 , 1-6

ま

窓（Window）関数 , 5-87 – 5-99
WinBartlett, 5-88
WinBlackman, 5-90
WinHamming, 5-94
WinHann, 5-96
WinKaiser, 5-97
窓（Window）関数の概要 , 5-87 – 5-88
マルチレート FIR LMS フィルタ関数
FIRLMSMRFree, 6-65
FIRLMSMRGetDlyLine, 6-70
FIRLMSMRGetDlyVal, 6-71
FIRLMSMRGetTaps, 6-68
FIRLMSMRGetTapsPointer, 6-69
FIRLMSMRInitAlloc, 6-64
FIRLMSMROne, 6-72
FIRLMSMROneVal, 6-73
FIRLMSMRPutVal, 6-72
FIRLMSMRSetDlyLine, 6-70
FIRLMSMRSetMu, 6-66
FIRLMSMRSetTaps, 6-68
FIRLMSMRUpdateTaps, 6-67
丸めモード、音声コーデック内の , 9-1

intel®

め

メディアン・フィルタ関数 , 6-100 – 6-102
  FilterMedian, 6-100
メモリ割り当て関数 , 3-3 – 3-6
  ippsFree, 3-5
  ippsMalloc, 3-4

も

文字列関数
  Compare, 11-7
  CompareIgnoreCase, 11-8
  Concat, 11-16
  ConcatC, 11-17
  Equal, 11-9
  Find, FindRev, 11-2
  FindC, FindRevC, 11-3
  FindCAny, FindRevCAny, 11-4
  Hash, 11-15
  Insert, 11-5
  Lowercase, 11-14
  Remove, 11-6
  ReplaceC, 11-12
  SplitC, 11-18
  TrimC, 11-10
  TrimCAny, 11-11
  Uppercase, 11-13
モデル推定関数 , 8-130 – 8-160
  WeightedVarColumn, 8-135
  BhatDist, 8-150
  DcsClustLAccumulate, 8-157
  DcsClustLCompute, 8-159
  Entropy, 8-147
  ExpNegSqr, 8-149
  GaussianDist, 8-144
  GaussianMerge, 8-146
  GaussianSplit, 8-145
  MeanColumn, 8-130
  MeanVarAcc, 8-143
  MeanVarColumn, 8-133
  NormalizeColumn, 8-139
  NormalizeInRange, 8-141
  OutProbPreCalc, 8-156
  SinC, 8-148
  UpdateGConst, 8-155
  UpdateMean, 8-152
  UpdateVar, 8-153
  UpdateWeight, 8-154
  VarColumn, 8-131
  WeightedMeanColumn, 8-134
  WeightedMeanVarColumn, 8-137
モデル適応関数 , 8-160 – 8-171
  AddMulColumn, 8-160
  AddMulRow, 8-161
  DotProdColumn, 8-163
  MulColumn, 8-164
  QRTransColumn, 8-162
  SumColumnAbs, 8-165
  SumColumnSqr, 8-166
  SumRowAbs, 8-167
  SumRowSqr, 8-167
  SVD, 8-168
  WeightedSum, 8-170
モデル評価関数 , 8-84 – 8-130
  AddNRows, 8-85
  BuildSignTable, 8-121
  DTW, 8-128
  FillShortlist_Column, 8-125
  FillShortlist_Row, 8-123
  LogAdd, 8-87
  LogGauss, 8-98
  LogGaussAdd, 8-109
  LogGaussAddMultiMix, 8-113
  LogGaussMax, 8-103
  LogGaussMaxMultiMix, 8-107
  LogGaussMixture, 8-115
  LogGaussMixtureSelect, 8-118
  LogGaussMultiMix, 8-101
  LogGaussSingle, 8-94
  LogSub, 8-88
  LogSum, 8-89
  MahDist, 8-91
  MahDistMultiMix, 8-93
  MahDistSingle, 8-90
  ScaleLM, 8-86

ゆ

ユーザ・フィルタバンクのウェーブレット変換 , 7-60 – 7-74

よ

予測エラー , 9-55
予測、周波数領域内の , 10-21

り

量子化
  音声コーデック内の , 9-28
  逆 , 9-89
  処理 , 9-33

る

累乗 , 9-8

れ

励振 , 9-43
ランダム , 9-81

ろ

論理関数とシフト関数 , 5-4 – 5-13
And, 5-5
AndC, 5-4
LShiftC, 5-11
Not, 5-10
Or, 5-7
OrC, 5-6
RShiftC, 5-12
Xor, 5-9
XorC, 5-8

intel®