たのしいYorick (改訂 α 版 (宇佐子の雛祭))

東芝インフォメーションシステムズ株式会社
横田博史

平成 23 年 2 月 20 日 (日)

歌川国芳: 相馬の古内裏

まえがき

ここで紹介する Yorick は使って楽しいアプリケーションで, 名前から想像されるような「貧弱」(poor) なソフトではありません.
私が初めて Yorick を知ったのは 1994 年頃のことでした. それは Mac 用のアーカイブを収録した CD-ROM 集 (InfoMac) を探索していたとき, 数学アプリケーションのフォルダの中に のような髑髏のアイコンがあることに気付きました. そこで附属の文書を読むと数値計算ができるとあったので $1+1$ のような簡単な数値演算を幾つかを試して最後に変数を未定義のままに x^2-2x+1 のような数式を入れたところで, いきなり MacOS の爆弾マークが出て,「髑髏 と爆弾 のアンサンブル」に見舞われたという強烈な思い出があります.
それから EWS に自力で初めてインストールに成功したのも, MacOS 上の UNIX 環境である MachTen 上で動かそうと苦労したのも, MkLinux でソースファイルを修正して最初にコンパイルしたのも Yorick だったと何かと思い出の多いアプリケーションです.
この Yorick は「軽い・速い・高機能」を売りにしたアプリケーションですが, 不思議なことにあまり人気がありません. その一因として配列処理の独自さが挙げられるでしょうが, この Yorick を使って頂ければ Yorick の持つ能力に驚嘆されるでしょう.
ここで Yorick に関連する絵として, Wikipedia から Sarah Bernhardt のポートレイトと歌川国芳の「相馬の古内裏」を引用しています. 双方共に Yorick 本人や骸骨に関係があります. また西は息子 (Hamlet) に東は娘 (滝夜叉姫) という違いもあって面白いでしょう.
この優れたアプリケーションを楽しまれる人が増えることを願って, このささやかな本を書きました. なお, この本は様々な方々の協力の御陰でできています. まず, KNOPPIX/Math については KNOPPIX/Math Project の方々, 本書に附属の KNOPPIX/Math 2010 特別版 DVD は濱田龍義先生に作成して頂きました. そして, KNOPPIX/Math に収録されたアプリケーションや文献を作成・保守されている方々に深く感謝します.
なお, この文書は「数値計算&画像処理ソフト Yorick」の加筆修正版, より正確には

Maxima 本と同様に (多分無さそうな) 改訂版に向けた α 版でもあります. 従って, この文書に関する質問は工学社ではなく私宛にお願いします. 平成 23 年 2 月 20 日 (日)

横田博史

目 次

第1章　Yorickの概要

Hamlet	ハムレット
Let me see. Alas, poor Yorick! I knew him, Horatio; A fellow of infinite jest, of most excellent fancy. He hath bore me on his back a thousand times.	見せてくれ. ああ, 可哀想なヨリック! 奴を憶えているよ, ホレーショ; 冗談ばかり言う, 頗る陽気な奴だった 幾千もお馬になってくれたっけ.
	Hamlet: 第五幕 第一場

1.1　Yorick とは

ここでは Yorick の特徴を幾つかのキーワードで纏めて紹介しましょう.

アイコンは髑髏:　Yorick のアイコンは髑髏 "" で, Shakespea の有名な悲劇「Hamlet」の「第五幕, 第一幕」で登場 (?) する道化師 Yorick に由来するようです. この場面は前のページに掲げた大女優 Sarah Bernhardt のポートレートからも判るように, Yorick の髑髏に Hamlet が話かける場面です[1]. ちなみに, この場面に出たいがために死後, 自分の頭蓋骨を劇団に寄付する人もいる程です[2].

数値配列処理ツール:　Yorick は「**数値配列**」に対して「**対話的な処理**」が行える「**ソフトウェア**」です. 似た処理が行えるソフトウエアに The MathWorks Inc. の「**MATLAB**」[16] があります. この MATLAB は LINPACK 等の数値計算ライブラリを学生が容易に扱えるようにするために開発され, 現在は Toolbox と呼ばれる膨大なライブラリ群を従えた数値行列処理を目的としたソフトウェアの標準的存在になっています[3]. MATLAB の影響を特に強く受けた「**OSS** (Open Source Software)」の代表的として「**GNU Octave**」[18] と INRIA の「**Scilab**」[19] を挙げておきます. Octave は GNU の MATLAB と言える程の高い互換性を持つ 非常に安定したシステムです. Scilab は Octave 程の互換性はないものの MATLAB と同様の処理が行え, さらに MATLAB の GUI 環境の「**Simulink**」[4]に似た「**SCICOS**」を擁する下手な売物を凌駕するシステムです.
Yorick は数値配列の算術演算を成分単位で行い, 配列の添字を活用して行列やベクトルを処理する仕様となっているので, MATLAB よりも一見すると複雑な処理になりますが, 高次元配列でも同様の表現で処理が可能で, 数値テンソルの処理に適した側面も持っています. このように MATLAB と Yorick は類似点もある一方で根底の趣向が異っています. それに加えて Yorick は軽いシステムです. 実際, Yorick のソースファイルの大きさは gzip で圧縮したもので 2.1 MB 程と Octave の 13.6 MB, Scilab の 34 MB と比較すると Yorick のコンパクトさが目立ちます.
Yorick は OSS なので自由に内部の閲覧や改造ができますが, 軽量なことが幸いしてソースファイルも Octave と比べて少なく, Yorick の働きを把握し易くなっています.

---

[1] このポートレートは Belle epoque のものなのでいささか大時代的です.
[2] `http://en.wikipedia.org/wiki/Yorick` を参照のこと. ここから自分の頭蓋骨をこの場面のために劇団に寄付したポーランド人ピアニスト Andre Tchaikowsky に辿れますが他にも色々と居るようです.
[3] 構成は `http://www.mathworks.co.jp/products/pfo` を参照.
[4] 本来は制御系のブロック線図の解析を目的にしていましたが, 現在では MATLAB の GUI 環境としても用いられています.

Yorick の処理言語: Yorick は対話処理が可能な C と思っても良いほど類似しています．だから C を少しでも知っていれば習得は容易です．ここで C との違いは, 変数の型の宣言を行う必要がない点と配列の添字が 0 からではなく 1 から開始する点が挙げられるでしょう．
ここで変数の型の宣言を行う必要がないことは型の概念がないことを意味するものではありません．変数はあくまでも対象を入れる容器としての役割を果すものであって, 対象そのものではないからです．そして配列の添字や一部の函数の引数では型が厳密に定められています．
そして, Yorick は対話処理が行える言語としては比較的高速な言語になります．これは Octave や Scilab と比較して言えることで, 数値計算ライブラリを利用する個所は同程度ですが, 言語のオーバヘッドが他と比較して軽いために全体的な処理が高速となっています．
一般的に対話処理が行える言語はコンパイラ言語と比較して低速です．ただし, 近年の計算機の能力向上に伴い, 対話処理言語でも十分な処理が行えるようになっているのが現状です．一例を挙げると 1980 年前半ではレイトレーシングを一晩かけて「パソコン」で計算していましたが, §11.3.6 で紹介している「**surfer**」のようなアプリケーションでは一枚一枚, 代数曲面のレンダリングを行うことでアニメーションを実現するという力技を披露している程です．

描画機能: Yorick が「貧弱」でない理由の 1 つに軽いシステムの割に描画能力が優れている点も挙げられますが, この描画機能には癖があるので慣れが必要です．この描画機能にはマウスを用いたグラフの拡大・縮小や移動, グラフ上の点の座標の読取やアニメーションもできます．

このように Yorick は C よりも気楽に使え, MATLAB よりも型を意識した使い方ができる言語を持った優れたシステムなのです．そこで今度は Yorick を使ってどのようなものであるか俯瞰してみることにしましょう．

1.2　簡単な入門

1.2.1　Yorick の動作環境

Yorick は UNIX, MacOS や MS-Windows といった主要な計算機環境で動作します. ただし,「仮想端末」(MS-Windows 環境なら「コマンドプロンプト **(command)**」) 上で利用することが前提で, 他はエディタの **GNU Emacs** を流用する程度です. MS-Windows 版のみに附属のフロントエンドも図 1.1 に示す MS-Windows 3.1 風の古風な代物です.

この本では「**KNOPPIX/Math 2009**」以降を標準的な環境とします. ここで「**KNOPPIX/Math**」については §13 で解説しておきますが, おなじ章で解説している「仮想計算機環境」と併用することで強力な数学アプリケーション環境として利用できます. さらに KNOPPIX/Math に収録された他の数学アプリケーションとの連動といった楽しみ方もできます.

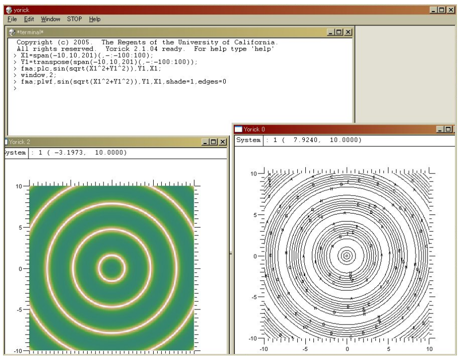


図 1.1: MS-Windows 版の Yorick の様子

また「仮想計算機環境」としては「**xVM VirtualBox**」[25][26] や「**VMware Player**」[27] が個人的な利用であれば無課金で利用でき, VirtualBox の OSE 版 (=<u>O</u>pen<u>S</u>ource<u>E</u>dition) なら自由に使えます. この KNOPPIX/Math や仮想計算機環境の利用については §13 を参照して下さい.

1.2.2　入れておくと便利なアプリケーション

KNOPPIX/Math 以外の環境で Yorick を楽しむために次のアプリケーションもインストールしておくことを薦めます:

入れておくと便利なアプリケーション

アプリケーション	入手先
Yorick	http://sourceforge.net/projects/yorick/
OpenOffice	http://ja.openoffice.org/download/
Maxima	http://sourceforge.net/projects/maxima/
ImageMagick	http://www.imagemagick.org/www/binary-releases.html
gs	http://sourceforge.net/projects/ghostscript/files/

Yorick については言うまでもないでしょう．**OpenOffice** は MS-Office を使っていない人向けです．OpenOffice の Calc や MS-Excel を使ってちょっと面白い実験をすることになるでしょう．**Maxima** は数式処理と呼ばれるソフトウエアで Yorick や MATLAB とは別の使い方ができます．**ImageMagick** は画像変換ツールのパッケージです．**gs** は ImageMagic で POSTSCRIPT ファイルの変換で使われます．ここで ImageMagick と gs のインストールを奨める理由は, Yorick で生成したグラフは容易に PostScript ファイルに落せることに加え, Yorick を使ってちょっとした画像処理をしてみたいからです．なお, KNOPPIX/Math であれば全て揃っています．

1.2.3 Yorick の起動

Yorick を動作して何ができるか確認しましょう．KNOPPIX/Math 2009 以前であれば下のメニューバーの左端から二番目の から Yorick を選択します．KNOPPIX/Math 2010 はデスクトップ環境を KDE から LXDE に変更したこともあってやや操作が異なります．最も簡単な方法は §13.7 で解説している KnxmLauncher を使って起動する方法で, 画面左下のアイコン を押せば昔のワープロ風のアイコンの並んだメニュが現われるので, このメニューの下側にある Yorick を押せばよいのです．つぎの方法は画面下側の左端にある「**LXDEM メニュー**」 を使う方法です．この を押して次に Math から Yorick を選択すればよいのです．なお, Yorick のメニューはあとで追加されたため, Yorick メニューがない KNOPPIX/Math 2010 をお使いの場合は画面下のツールバー左側にあるアイコン を押して仮想端末の LXTerminal を起動して `yorick` か `rlwrap yorick` と入力して下さい．ここで **rlwrap** は GNU Emacs 風の行編集と履歴が扱えるようにするアプリケーションです[5]．なお, ここでは LXTerminal から Yorick を起動させましたが,

[5]**rlwrap** = readline libery **wrap**per

仮想端末であれば xterm でも konsole でも何でも構いません．また MS-Windows 環境であれば「**スタート**」から「**すべてのプログラム (P)**」を選び, そこから「**Yorick 2.1.04**」[6]の中の 

を選択すると Yorick が立ち上がります．なお, 「**コマンドプロンプト (DOS 窓, command 命令)**」で `yorick` と直接打ち込んでも構いませんが, この場合は予め yorick の在処が環境変数「**Path**」に登録されていなければなりません．
ここで Yorick が起動すると次の表示が端末に現われている筈です:

```
Copyright (c) 2005.  The Regents of the University of  California.
All  rights  reserved.   Yorick 2.1.05  ready.   For help type 'help'
>
```

ここで最後の行の記号 “`>`” が Yorick のプロンプトで, このプロンプトに続けて文を入力します．Yorick のプロンプトには「**虫取りモード**」に入っていることを示すプロンプト “`debug>`” や「**入力の継続**」を示すプロンプト “`cont>`” があります．
なお Yorick のライセンスは「**BSD License**」というライセンスです．このライセンスの全文を読みたければ `legal` と入力すると条項が表示されます．どうでしたか？もしも貴方が xterm 等の仮想端末で操作していれば, この条項全文がバッファに入り切らないために頭から読むことができない筈です．このような事態を避けるためには, GNU Emacs のような外部アプリケーションをフロントエンドに使うしか有効な手立てがない弱点があります．

1.2.4　Yorick のデモ

Yorick には標準で幾つかのデモファイルが付属しています:

1. demo1.i: 2 次元グラフ表示の例
2. demo2.i: 太鼓の振動のアニメーション
3. demo3.i: Chaotic pendrum の例 (非力な計算機では要注意)
4. demo4.i: 2 次元の流体計算の例
5. demo5.i: 3 次元可視化の例

[6]Yorick のうしろの番号 2.1.04 は執筆時の Yorick の版です．

ここで demo2 を見たければ Yorick のプロンプト “>” に続けて `include,"demo2.i"` と入力することで “demo2.i” ファイルの内容を読込み, それから今度は `demo2` と入力すれば demo2 が起動します. すると図 1.2 に示すような膜がウニョウニョ動いているアニメーションが表示される筈です.

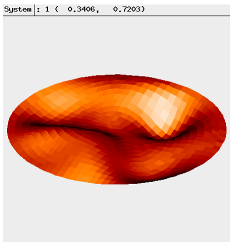


図 1.2: demo2 の様子

今度は `Y_SITE` と入力してみて下さい. UNIX 環境なら `"/usr/lib/yorick/"`, MS-Windows 環境なら`"C:/Program Files/yorick-2.1.04/"`といった文字列が表示されますが, この文字列で指示された “i” ディレクトリ (フォルダ) にデモファイルが収録されています. これらは Yorick 言語で記述されたライブラリ (ファイルの修飾子が “.i”) です. Yorick は Octave や Scilab と比較して今風に “user-friendry” ではありません. むしろ「**必要なものは自分で作る**」という流儀があり, デモファイルやライブラリは参考になります.

1.2.5 電卓代りに使ってみよう！

対話処理

ここでは「太鼓の振動」のデモのようなことではなく, 日常的な計算の話に限定して Yorick を使ってみましょう. 今度は `1+1` と入力して下さい:

```
> 1+1
2
```

すると 1＋1 の意味の 2 が即座に返却されていますね. Yorick はこのような対話処理が行えるシステムなのです. ここで「**対話処理**」とはキーボード等の入力端末を介して「**計算機と対話**」を行うように「**処理**」が進められることです.

整数

さて Yorick ではどのような数が扱えるのでしょうか？ まず「**整数**」が扱えます. 扱える整数の範囲は比較的広く, OS が 32 bit 環境であれば -2147483648 から 2147483647 の範囲, OS が 64 bit 環境で Yorick も 64 bit 版であれば -9223372036854775808 から 9223372036854775807 の範囲の整数が扱えます. そこで扱える範囲を逸脱するとどうなるでしょうか？ Yorick らしく髑髏 “” でも出すのでしょうか？ そこで `9223372036854775808` と入力してみましょう. どうなりましたか？ 恐らく 0 や負の数になって入力した値と異っている筈です. この理由は整数の内部表現によるもので, 詳細は §2.2.1 を参照して下さい. このような大きな整数の計算で正しい結果が得られないことがあり, この問題を避けるために整数を「**浮動小数点数**」で表現するといった工夫が必要になりますが, 浮動小数点数でも, たとえば倍精度であれば, 絶対値が 2^{53} を超過すると近似計算になって 64 bit 環境の整数演算よりも精度が落ちてしまうので, 可能であれば 64 bit 環境で使うことを推奨します[7].
では整数の範囲に注意して整数計算を幾つかさせてみましょう. そのために Yorick の代表的な二項算術演算子を纏めておきます:

Yorick の二項算術演算子

演算子	演算子の概要	可換性	表記	対応する数式
+	二項間の和を計算	可換演算子	`1+2`	$1+2$
-	二項間の差を計算	非可換演算子	`3-2`	$3-2$
*	二項間の積を計算	可換演算子	`4*5`	3×5
/	二項間の商を計算	非可換演算子	`6/2`	$6/2$
^	二項間の冪を計算	非可換演算子	`2^3`	2^3
%	二項間の剰余を計算	非可換演算子	`19%4`	$19 \bmod 4$

ここで演算子の優先度は通常の数式と同じですが, 用心のために優先度を示しておきましょう:

演算子の優先度

“+”	<	“-”	<	“%”	<	“*”	=	“/”	<	“^”

Yorick では数式の表記の左側から, 演算子の優先度に従って解釈されます. もし優先度に反する演算を行う場合や優先度を明示したい場合は記号 “( )” を併用します. この記号 “( )” は数式で用いられる括弧 ( ) と同じ意味です. たとえば数式 $1-2\times 3^4/2$ と

[7] 残念ながら KNOPPIX/Math 2009 は 32 bit 環境であり, MS-Windows 版の Yorick も 32 bit 版のみです. 64 bit 版を利用したければ自力で構築する必要があります.

$1-(2\times(3^4))/2$ は同値な式ですが, Yorick における表記 '1-2*3^4/2' と '1-(2*(3^4))/2' も同様に同じ「**意味**」, すなわち同じ「**値**」を持ちます.

型の問題

ここで問題ですが数式 $1-3^4/2\times 2$ の Yorick での「文字通りの表記 」'1-3^4*2/2' と数式 $1-((3^4)/2)\times 2$ の「文字通りの表記」 '1-((3^4)/2*2' の「意味」は先程の数式 $1-2\times 3^4/2$ の「文字通りの表記」 '1-2*3^4/2' の「意味」と一致するでしょうか？

```
> 1−2∗3ˆ4/2
−80
> 1−(2∗(3ˆ4))/2
−80
> 1−3ˆ4/2∗2
−79
> 1−((3ˆ4)/2)∗2
−79
```

数式であれば一致する筈ですが, Yorick の処理では異なりますね. 同値な数式の筈なのに Yorick の表記 '1-2*3^4/2' と '1-3^4/2*2' の意味が違うのは何故でしょう？ 前者を記号 "( )" を使って同値な表記に書き直せば '1-(2*(3^4))/2' になります. それに対し後者は '1-((3^4)/2)*2' です. 括弧 "( )" を使うことで表記の構造の違いが明瞭になりますが, この違いの意義を観察してみましょう. まず前者は '2' と '3^4' の積 "*" を実行してから '2' の商 "/" が実行されます. そして後者では '3^4' と '2' の商 "/" を実行してから '2' との積 "*" を実行します. ここで商 "/" に着目して下さい. 前者は「偶数 '2*3^4' を '2' で割る」処理になっていますが, 後者は「奇数 '3^4' を '2' で割る」処理になっています. ここで**商 "/" は整数同士の演算では整数を返却**します. だから '(3^4)/2' の結果は偶数 '(3^4-1)' を '2' で割った値に等しく, その結果を 2 倍すれば両者の差異の '1' が現れます.

このように **Yorick の算術演算子は被演算子が同じ型であれば結果も同じ型**になります. だから Yorick で整数 1 と整数 2 を使った表記 '1/2' の意味は整数 0 であって, 有理数 1/2 でも実数 0.5 でもありません. このように Yorick で数式を表現する場合, Yorick での数式の表現の「**意義**」[8], つまり演算子の性質や演算子の組合せといったものが, その表記の元となる数式以上に Yorick での「**意味**」, すなわち「**値**」に関連します. これが MATLAB 系の言語と異なる点です. たとえば Octave で同じ計算をさせてみましょう:

[8]「意義」と「意味」については「算術の基本法則」[6] を参照

```
octave:1>  1−2*3^4/2
ans = −80
octave:2> 1−(2*(3^4))/2
ans = −80
octave:3> 1−3^4/2*2
ans = −80
octave:4> 1−((3^4)/2)*2
ans = −80
```

このように Octave では全ての結果が一致します．一般的に MATLAB 系の言語は与件の型をあまり意識しなくても処理が行えるように設計されていますが, **Yorick では常に与件型のことを念頭に置いて処理を進めなければなりません.**

浮動小数点数

では小数点を含む 0.5 といった数, すなわち「**実数**」は Yorick ではどのように表現すればよいのでしょうか？ これは単純に '123.' や '12.34' のように小数点記号 "." を数に 1 つだけ挿入すればよいのです．このような計算機で実数を表現する数のことを「**浮動小数点数**」と呼びます．この浮動小数点数は '12345.' のような小数点を使った表記の他に '1.2345e+4' といった冪を使った表記も使えます．この数は「**近似値**」としての性格を持ちます．つまり「**丸め**」や「**切捨**」といった操作を経由した数です．したがって π のような「**超越数**」や 1/5 のように 2 進数展開したときに「**巡回小数**」となる有理数は近似値で表現されます．詳細は §2.2.3 を参照して下さい.
さて Yorick の算術演算では整数よりも浮動小数点数が優先されます． つまり式中に 1 つでも浮動小数点数が含まれていれば結果は浮動小数点数になります．だから $1 - 3\hat{\ }4/2 * 2$ を正確に計算するためには商の演算子 "/" の左右のどちらかの項が浮動小数点であればよいのです:

```
> 1−((3^4)/2)*2
−79
> 1−((3.^4)/2)*2
−80
> 1−((3^4.)/2)*2
−80
> 1−((3^4)/2.)*2
−80
> 1−((3^4)/2)*2.
−79
```

Yorick は小数点以下が 0 だけであれば, MATLAB や Octave と同様に小数点以下を

省略して整数のように表示しますが, その与件型が浮動小数点数であることに注意しなければなりません:

```
> 123.
123
> 1.23e+2
123
> 1.23e+2/300
0.41
```

Yorick では有理数はそのままでは扱えません. ‘2/5’ という表記は演算子 “/” を使って二つの整数 2 と 5 を処理するという意味で有理数 2/5 を意味しません. では有理数を使いたければどうすればよいのでしょうか？ その場合は自分で定義すればよいのです. 具体的な手法は §11.2 を参照して下さい.

複素数

Yorick では「**複素数**」も扱えます. このとき虚部は通常の数値表現に続けて記号 “i” を付加した表現になります:

```
> 1i
0+1i
> 2.4i
0+2.4i
> 1.23e+4i
0+12300i
> 12+5e−1i
12+0.5i
```

ここで純虚数は ‘i’ 単体ではなく ‘1i’ であることに注意して下さい.
これで Yorick を複素数も扱える電卓代りに使えるようになったでしょうか？もちろん, Yorick にできることはこれだけではありません.

1.2.6 変数

自由変数と束縛変数

今度は「**変数**」について解説しましょう. ここで「変数って何？」という人はいませんか？ **Yorick の「変数」は式中で値を入れるために用いる箱のようなものです**. そこで Yorick に `eewyg` と入力して下さい.

```
> eewyg
[]
```

ここで Yorick は空の配列 '[]' を返しました．この対象 '[]' は nil と呼ばれる特殊な対象で, 変数 eewyg には '[]' 以外の何者も束縛されていないことが判ります．ここでは '[]' を値に持つ変数のことを (具体的な値が) 「束縛されていない変数」, すなわち「**自由変数**」と呼びます．したがって変数 eewyg は自由変数になります．ちなみに "eewyg" の他の意味は日本語キーボードの平仮名をご覧になると良いでしょう[9]．では「**自由ではない変数**」とはどのような変数でしょうか？変数を Yorick の対象を入れられる箱だと思えば '[]' 以外の対象が入っている箱に相当します．この '[]' 以外の対象を値に持つ変数のことを「**束縛変数**」と呼びます．そして演算子 "=" が「**箱**」, すなわち「**束縛変数**」を創り出す演算子になります．さて `eewyg="yes"` と入力し, それから再び `eewyg` と入力して下さい:

```
> eewyg="yes"
> eewyg
"yes"
```

ここで最初の '`eewyg="yes"`' で束縛変数 eewyg を生成しますが, 演算子 "=" を用いたときに Yorick からの返答がないことに注意して下さい．ここで変数に束縛されて値を見たければ単純に変数名を入力します．この例では `eewyg` と入力することで値 `"yes"`を返却しています．

割当と代入

ここで「**割当**」と「**代入**」という言葉を導入しておきましょう．まず変数への「**割当**」は値を入れるための変数を Yorick 内部で生成し, その変数に値を束縛させる処理のことです．一方の「**代入**」は既存の変数に新たに値を束縛させることを意味します．Yorick では函数によって割当と代入の双方の処理を行う函数と代入のみを行う函数が存在するので注意が必要です．その理由は**「代入」を行うためには入れ物となる「変数」が存在しなければならない**からです．この事態は「**ポインタ**」を用いた処理で顕在化します．
変数の型の宣言は Yorick では基本的に不要ですが函数によっては指定した変数に配列をポインタを用いて代入するために予め配列の宣言が必要な場合もあります．しかし通常の演算子 "=" を用いて代入処理を行うだけであれば与件型の宣言は不要です．

[9] eewyg = いいてんき (良い天気 or 良い転機)

そのために A という変数に整数 1 を束縛させ, それから浮動小数点数 1.2 を割当てて最期に文字列`"end"`を割当てるといったことができるのです. このように **Yorick の変数は「Yorick の任意の対象が入れられる箱」であって, C のような「あらかじめ指定した型の対象だけが入れられる箱」ではありません**. また演算子 “=” は MATLAB や Octave でも同様ですが ‘a=b=c=d=1’ のような割当・代入の表記ができます:

```
> a=b=c=d=1
> print,a,b,c,d
1 1 1 1
```

この例で示すように一番右の対象が演算子 “=” の右側の変数に割当てられます.

さて変数を解説したので, これで $x = 1$ のときの多項式 $x^2 + 3x + 4$ の計算が行えるようになりましたね. 今度は初等函数の話を簡単にしておきましょう.

1.2.7 初等函数

Yorick には三角函数 cos, sin 等, 対数函数 log, 指数函数 exp 等の「**初等函数**」が用意されています. これらの函数では ‘cos(128)’ のように値を函数に引渡したり, 数値が束縛されている変数 x を使って ‘sin(x)’ のような書式で入力すれば Yorick は即座に計算して値を返却します. さらに値が束縛された変数や数値函数, 四則演算子を使いることで通常の数式に近い形で入力が行えます. たとえば $x = 1.234$ のときの $\sin^2 x - x\log x - 1/x$ の値を求めたければ次のように入力すれば良いのです:

```
> x=1.234
> sin(x)^2+x*log(x)-1/x
0.339882
```

さて変数 x を区間 $[1, 2]$ 上の幾つかの点とするときに数式 $\sin^2 x - x\log x - 1/x$ をどのように計算すれば良いでしょうか？1 つの方法は C や FORTRAN でお馴染の方法の一点づつ計算した結果を配列に収める方法です. ところが Yorick のような言語では引数に「**配列**」という対象を使うという手法もあり, そちらの方が効率良く処理が行えるように工夫されています.

1.2.8 配列

配列という対象

Yorick には「**配列**」という対象があります. 配列は n 個の整数の組 $(i_1, \ldots, i_n)$ に対して Yorick の 1 つの対象 $a(i_1, \ldots, i_n)$ が定められる対象で, 配列の成分を指定する整

数の組 $i_{k,\ k=1,\dots,n}$ を「添字」, $(i_1,\dots,i_n)$ の左から k 番目の添字 i_k のことを「k 次の添字」と呼びます. ここで Yorick の添字は C と異なり通常 1 から開始します. そして添字 $i_1,\dots,i_n$ が取り得る上限 $N_1,\dots,N_n$ を並べた列のことを「配列の大きさ」と呼び, ここでは $N_1\times\cdots\times N_m$ と記述します. そして配列の添字の個数のことを「配列の次元」と呼びます. この配列の次元は配列の大きさを構成する整数列の長さに対応します. 実際, 配列の大きさが $N_1\times\cdots\times N_m$ のときに次元は m になります.
この Yorick の配列は数学の「テンソル」に対応しており, この点を踏まえておけば Yorick の配列処理の特徴が理解し易くなります. 一方で MATLAB や Octave は「テンソル」ではなく「行列」に対応し, この違いが Yorick と MATLAB との処理手法の違いに大きく影響しています.
ところで Yorick の殆どの対象が配列としての性質を持ちます. たとえば数値や文字列は 0 次元配列になります. この詳細は §3 で解説します. そして 1 次元以上の配列は鉤括弧 "[ ]" で括られた数値列や文字列として表現され, この「鉤括弧の深さ」が「配列の次元」, 「鉤括弧による区分」が「配列の大きさ」を定めます. そして「配列の成分の総数」のことを「配列の長さ」と呼びます.

1 次元配列と 2 次元配列

比較的理解し易い 1 次元と 2 次元の配列について解説しておきましょう.
Yorick の 1 次元配列は Yorick の対象の列を記号 "[ ]" で括った対象です. たとえば '[1,2,3,4,5]' や '["yes","no"]' は 1 次元配列になります. この 1 次元配列のことを特に「ベクトル」, あるいは「リスト」と呼びます. ただし Yorick は「**LISP** 風のリスト」も持っているので, 1 次元配列のことは「リスト」ではなく「ベクトル」を主に用います. また「ベクトルとしての配列の大きさ」のことを単に「長さ」とも呼びます.
そして 2 次元配列はベクトルやベクトルの列を記号 "[ ]" で括った対象になります. たとえば配列 '[[1],[2],[3],[4],[5]]' と配列 '[[1,2,3],[4,5,6]]' は 2 次元配列です. そして配列の大きさは配列 '[[1],[2],[3],[4],[5]]' が 1×5, 配列 '[[1,2,3],[4,5,6]]' で 3×2 になります.

MATLAB 系言語との 2 次元配列表記の違い

Yorick の 2 次元配列が行列に対応しますが, MATLAB 系の言語とはどのような違いがあるでしょうか？ 簡単な入力例で比較してみましょう:

$$\text{Yorick の入力：}\quad \texttt{[[1,2,3],[4,5,6]]} \quad\Leftrightarrow\quad \begin{pmatrix}1 & 4\\ 2 & 5\\ 3 & 6\end{pmatrix}$$

$$\text{Octave の入力：}\quad \texttt{[1,2,3;4,5,6]} \quad\Leftrightarrow\quad \begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix}$$

このように Yorick の行列 (2 次元配列) の表記と MATLAB 系の言語の行列の入力時の記述は単純に転置の関係にあります[10]

配列を使った計算

Yorick の配列処理には MATLAB や Octave でも容易に行える処理と, Yorick 独自の処理に分類できます. ここでは最初に MATLAB でも可能な処理について述べます. まず Yorick では同じ大きさの数値配列に対して数値と同様の四則演算が行えます[11]:

```
> a=[1,2,3]
> b=[9,2,4]
> a+b
 [10,4,7]
> a^2+2*b^3-b%3
[1459,18,136]
```

そのために §1.2.7 の問題:「区間 $[1,2]$ の点 x に対して $\sin^2 x - x\log x - 1/x$ を計算」は変数 x が数値配列でありさえすれば容易に行なえます. ここでは変数 x に span 函数を用いて, 始点を 1, 終点を 2 とする等間隔の 11 個の点から構成される 1 次元配列 x を生成します:

```
> x=span(1,2,11)
> x
   [1,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2]
> y=sin(x)^2+x*log(x)-1/x
> y
[-0.291927,-0.00999915,0.254149,0.500287,0.727887,0.936527,1.12615,
1.29723,1.45084,1.58869,1.71312]
```

[10] これは入力の表記に関することで, Yorick の配列 a の (i,j) 成分を行列 A の (i,j) 成分に対応させていれば転置とは無関係です.

[11] より正確には Yorick では配列の成分単位で四則演算が行われますが, MATLAB 系の言語では行列演算が主体となるために成分単位の演算とは限りません. また利用者定義の構造体に対する演算では勝手が異なることに注意が必要です.

では配列の成分はどのように取出すのでしょうか？ それは単純に何番目の成分であるかを単純に整数や整数の組で指定します．具体的には n 次元の配列 `a` の $(i_1,\ldots,i_n)$ 成分は ‘a$(i_1,\ldots,i_n)$’ で取出せます．

配列の照合

Yorick に限らず MATLAB や Octave のように配列や行列を扱うことに長じた言語では「配列の照合」が容易に行えます．たとえば配列 ‘[5,4,3,2,1]’ に対して 3 以上の成分が何処にあるかを次の方法で簡単に調べられます:

```
>[5,4,3,2,1]>=3
 [1,1,1,0,0]
> where([5,4,3,2,1]>=3)
 [1,2,3]
```

この例を簡単に解説しておきましょう．まず ‘[5,4,3,2,1]>=3’ により 3 以上の成分を 1, それ以外を 0 で置換えた配列 ‘[1,1,1,0,0]’ が返されます．この配列に対して where 函数 [12]が 0 と異なる配列の成分位置を 1 次元配列で返した結果が配列’[1,2,3]’ なのです．この方法は高次元の配列でも行えます．実際に 2 次元配列で試してみましょう．まず 3×3 の乱数配列 rnd を random 函数を使って `rnd=random(3,3)` で生成します．さて配列 rnd に対して 0.3 よりも大きな数が何処にあり, それらがどのような数で幾つあるのでしょうか？ Yorick では次の処理で求めることができます:

```
> rnd=random(3,3)
> rnd
 [[0.0891647,0.0719207,0.739632],[0.191635,0.350893,0.597056],[0.420973,
0.608159,0.607138]]
>  rnd>0.3
   [[0,0,1],[0,1,1],[1,1,1]]
> where(rnd>0.3)
 [3,5,6,7,8,9]
> where2(rnd>0.3)
      [[3,1],[2,2],[3,2],[1,3],[2,3],[3,3]]
> sum(rnd>0.3)
6
```

まず ‘rnd >0.3’ により 0.3 よりも大きな配列 rnd の成分が 1, 0.3 以下の成分が 0 で置換されます．この配列に対して where 函数や where2 函数で配列の 0 と異なる成分の位置を検出することができます．まず where 函数は配列を単純な 1 次元配列と見做して 0 以外の成分の添字の配列を返し, where2 函数 n が配列の次元に則した添字の

[12]MATLAB では find 函数が対応します．

配列を返却します. ここで 'rnd>0.3' が 0 と 1 の配列になりましたが, この配列の総和を取れば 0.3 よりも大となる成分の総数になります.
このように Yorick の「真理値」は偽が '0', 真が '1' で表現される[13]ので, 上手く使えば if 文や for 文を使わずに済ますことができます. たとえば配列 x の成分が 5 よりも大であれば 2 倍し, 5 以下であれば-1 で置換する処理は '(x>5)*x*2+(x<=5)*(-1)' と一行で表現できます:

```
> x =[1,2,3,4,5,6,7,8,9,10]
> (x>5)*x*2+(x<=5)*(−1)
[−1,−1,−1,−1,−1,12,14,16,18,20]
```

このような処理が容易に行えることが MATLAB 系のシステムの醍醐味です[14].

Yorick 独自の配列処理

Yorick 独自の配列処理としては配列の添字だけで色々な処理が行えるということが挙げられます. たとえば 1 次元配列 a に対して, その成分の平均値や総和を 'a(avg)' や 'a(sum)' でそれぞれ求めることができます. ここで具体例で解説しましょう. まず区間 $[0, 3]$ 内の等間隔の 11 個の点で構成された配列の sin 函数による像の配列 x を考えます. すると, その平均値と総和は次で求められます:

```
> x=sin(span(0,3,11))
> x
[0,0.29552,0.564642,0.783327,0.932039,0.997495,0.973848,0.863209,
0.675463,0.42738,0.14112]
> x(avg)
0.604913
> x(sum)
6.65404
> x(sum)==sum(x)
1
> x(avg)==sum(x)/11.
1
```

ここで 'sum(x)' で総和が, 'sun(x)/11.' で平均値が計算できますが, それらは 'x(sum)' や 'x(avg)' と一致します. もちろん, 函数を用いた方法が添字を用いる方法よりも処理速度で幾分勝ります. とはいえ一寸した利用で問題にならない程度であり, 高次の配列処理では特定の添字に対して処理を行うことができるので, この点は目的の機能を実現させるために費す手間との兼ね合いになるでしょう.

[13] より正確には**偽は '0', 真は '0' 以外の対象**です. しかし, Yorick の述語函数の多くが偽を int 型の整数 '0', 真を int 型の整数 '1' で出力しているので, ここではこのように記述しています.
[14] MATLAB でも '(x>5).*x*2+(x<=5)*(-1)' で同じ結果が得られます.

1.2.9　虫取りモード

Yorick の処理中にエラーが発生すると次のような表示が出てきます;

```
> sin(x,y)
ERROR (*main*) expecting exactly one argument
WARNING source code unavailable (try dbdis function)
now at pc= 1 (of 14),  failed  at  pc= 7
 To enter debug mode, type <RETURN> now (then dbexit to get out)
>
```

この例では自由変数 x, y に対する式 ‘sin(x,y)’ の評価で意図的に構文エラーを起させていますが, ここで現われた表示の最期の行を見てください. 超訳すれば「`<RETURN>`を押せば虫取りモードに入られるよ. (抜けたければ `dbexit` を使ってね)」とありますね. では `RETURN` キーか `Enter` キーを押してみましょう. すると次に示すようにプロンプトが “`debug>`” に切り切替わります:

```
> sin(x,y)
ERROR (*main*) expecting exactly one argument
WARNING source code unavailable (try dbdis function)
now at pc= 1 (of 14),  failed  at  pc= 7
 To enter debug mode, type <RETURN> now (then dbexit to get out)
>
dbug>
```

このプロンプトが出た状態が虫取りモードです. この状態で問題を起している函数の局所変数や大域変数の値を確認したり, どのような処理が実行されているかを調べることができます. この虫取りモードから抜けたければ `dbexit` を入力しましょう.

1.2.10 数学函数とそのグラフ

Yorick で通常の XY グラフは plg 函数を使って描けます. たとえば次の入力を行うと描画用の Window が出現し, 図 1.3 に示すグラフが描かれます:

```
x1 =[1,2,3,4,5,6,7,8,9,10];
y1=x1^2;
plg,y1,x1;
```

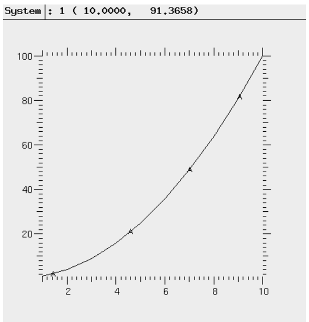


図 1.3: “plg,y1,x1” の結果

ここでグラフを描くために配列を手入力しても仕方がありませんね. もっと良い方法はないでしょうか？ Yorick には等間隔の 1 次元配列を生成する函数に indgen 函数と span 函数があります. ここで indgen 函数が生成する配列は整数型, span 函数が生成する配列は浮動小数点数型の 1 次元配列になります.

```
> indgen(1:10)
  [1,2,3,4,5,6,7,8,9,10]
> indgen(1:10:2)
 [1,3,5,7,9]
> span(0,10,11)
  [0,1,2,3,4,5,6,7,8,9,10]
> span(0,1,11)
    [0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]
```

これらの函数を使うことで, いろいろな函数のグラフを plg 函数を使って描くことができます. 勿論, Yorick の描画函数はこれだけではありません. その他の描画函数については §10 の画像処理で詳細を述べます.

第2章　Yorickの対象

Hamlet

To be, or not to be: that is a question:

Hamlet: 第三幕, 第一場

2.1　Yorick で扱える基本与件型

Yorick で扱える基本的な型を次にまとめておきます:

与件型一覧

対象	与件型
整数:	char, short, int, long
実数:	float, double
複素数:	complex
文字列:	string
自由変数:	void
構造体:	struct_instance , struct_definition
領域:	range
函数:	function, autoload, buildin, spawn-process
ポインタ:	pointer
ストリーム:	stream, text_stream, bookmark

ここで Yorick の大きな特徴は, 整数, 実数や複素数といった数を表現する対象や文字列といった対象は単体で配列としての構造を持っていること, すなわち 0 次元の配列であるということです. このことは C や FORTRAN と大きく異なる性質で, 配列を主体とする Yorick の性格を端的に現す性質です. ただし全ての対象が単体で配列としての構造を持つわけではありません. たとえば構造体, 函数, ポインタやストリームは単体で配列の構造を持ちません.
また配列の「**領域**」を表現する range 型は MATLAB でお馴染の '1:10' のような表記で, その意味も MATLAB と同様に 1 から 10 までの自然数の列に対応しますが, Yorick では函数の引数や配列の添字として用いられて単独では使えない対象です.

2.2　Yorick の整数と実数の与件型

Yorick の実数を表現する型には, 整数を表現する「**char 型**」,「**short 型**」,「**int 型**」と「**long 型**」の 4 種類の与件型, 実数を表現する浮動小数点数の「**単精度 (single)**」と「**倍精度 (double)**」の 2 種類の与件型, 複素数を表現する「**complex 型**」があります. これらの与件型を次に纏めておきましょう:

整数と実数の型

対象	与件型	概要
整数:	long	符号付き 64-bit 長 (64-bit 環境), あるいは 32-bit 長 (32-bit 環境) の整数表現. 64-bit 長の場合は -9223372036854775808 から 9223372036854775807 まで, 32-bit 長の場合は -2147483648 から 2147483647 までの整数が扱える
整数:	int	符号付き 32-bit 長の整数表現. -2147483648 から 2147483647 までの整数が扱える
整数:	short	符号付き 16-bit 長の整数表現. -32768 から 32767 までの整数が扱える
整数:	char	8-bit 長の整数. 0 から 255 までの正整数が扱える.
実数:	double	倍精度浮動小数点数.
実数:	float	単精度浮動小数点数.

これらの整数や浮動小数点数の違いは何でしょうか？ またより本質的なことですが, 計算機内部で整数や実数はどのように表現されているのでしょうか？ そのために整数の表現について解説することにしましょう.

2.2.1 整数の表現

Yorick で扱える整数の型

整数の与件型の違いは, 端的に言えば整数を内部で表現するために費される 2 進数の桁数です. Yorick では明示的に整数の型を指定しなければ long 型の整数として扱われます. また後述の各型に対応する変換函数を用いて希望する型に変換できます.
なお, Yorick の整数は各型の範囲内で処理されるために注意が必要になります. つまり long 型は 64-bit 環境なら 64-bit 長の整数, int 型は 32-bit 長の整数, short 型は 16-bit 長, char 型は 8-bit 長の 2 進数として整数値が表現されるために各型の上限を越えた整数は本来の意図した整数として表現されないことを意味します. ここで「**n-bit 長**」とは 0 か 1 のみが入れられる「**n 個の箱**」と言い換えられるでしょう. たとえば char 型は正整数のみを 8-bit 長の 2 進数として表現した対象, すなわち $\boxed{d_7}\boxed{d_6}\boxed{\dots}\boxed{d_0}$ のような 8 個の箱の各 d_i $(0 \leq i \leq 7)$ の中に '0' か '1' のどちらかが入れられた対象として表現し, 正整数 $\sum_{i=0}^{7} d_i \times 2^i$ を対応させせます. ここでもし $2^8 - 1$ を超過するとどうなるでしょうか？ すると 8 個の箱で表現し切れませんが, Yorick はこの状態をエ

ラーにせずに箱からの超過分を無視して 8 個の箱の部分だけで整数を表現します．つまり与えられた整数を 2^8 の「**剰余類**」[1]で考えているのです．

補数による負の数の表現

正整数のみを扱う **char** 型の処理はまだ簡単ですが, **long** 型, **int** 型や **short** 型のように負の整数が表現できる型はもっと複雑になります．まず, 限られた箱で数値を表現しなければならないので, 2 進数の桁数を n とするときに 2^n で割ったときの剰余を扱うことは char 型と同様です．では負の数をこれらの箱を使ってどのように表現すればよいのでしょうか？ ここで負の数は -128 のように数字の前に記号 “$-$” を付けたものなので計算機内部の表現でも何かのフラグを立ててしまえば良さそうですが, その前に「**負の数**」の意味をもう少し考えでみませんか？ 何よりも整数 a に対する負の数 $-a$ は等式 $a+(-a)=0$ を満す数, すなわち **「負の数」は「元の数に足すと結果が 0 になる数」**です．そこで $a+(-a)=0$ を計算機でどのように表現すべきか考えてみましょう．
まず正整数 a を char 型で考えたように ‘0’ と ‘1’ の羅列で表現します．では 0 を計算機でどう表現すればよいでしょうか？ これは全ての箱の値が ‘0’ になる列を 0 に対応させることが自然です．これで $a+(-a)=0$ に現われる対象 a と 0 の計算機における表現ができました．では残りの a の負の数 $-a$ はどう料理しましょうか？ ここでは箱の数を 64 個, a を 2^{62} という数にして具体的に考えてみましょう．この数 2^{62} は char 型の例から $\begin{array}{|c|c|c|c|c|}\hline 0_{63} & 1_{62} & 0_{61} & \dots & 0_0 \\ \hline\end{array}$ で表現できます．ここで天下り的ですが, この 2 進数表現の各 bit を単純に反転させた $\begin{array}{|c|c|c|c|c|}\hline 1_{63} & 0_{62} & 1_{60} & \dots & 1_0 \\ \hline\end{array}$ を考えます．この数は 2^{62} の「**1 の補数**」と呼ばれる数です．ここで元の数とその「1 の補数」を足し合わせると $\begin{array}{|c|c|c|c|c|}\hline 1_{63} & 1_{62} & 1_{61} & \dots & 1_0 \\ \hline\end{array}$ が得られ, この結果に 1 を加えると 2^{64}, すなわち (1_{64}) $\begin{array}{|c|c|c|c|c|}\hline 0_{63} & 0_{62} & 0_{61} & \dots & 0_0 \\ \hline\end{array}$ が得られます．そして箱の外に出た bit は char 型での処理のように無視すること, すなわち 2^{64} による剰余を扱うようにすれば安心して $\begin{array}{|c|c|c|c|c|}\hline 0_{63} & 0_{62} & 0_{61} & \dots & 0_0 \\ \hline\end{array}$, つまり 0 が得られます．このことから 2^{62} の逆数 -2^{62} はその「1 の補数」に 1 を加えた数, つまり 2^{62} の「**2 の補数**」の $\begin{array}{|c|c|c|c|c|}\hline 1_{63} & 1_{62} & 0_{60} & \dots & 0_0 \\ \hline\end{array}$ で与えられ, この「2 の補数」は構成方法から判るように一意に定まります．以上から正整数 a に対する負の数 $-a$ は a の「2 の補数」で与えればよいのです！
この「2 の補数」を用いた負の整数の表現では 64 番目の bit を正負の表現で利用することになってしまうので, 正整数としては残りの 63 bit しか使えません．そのために 64-bit 長の整数は 64 番目の bit が ‘0’ で他が全て ‘1’ の対象 $\begin{array}{|c|c|c|c|c|}\hline 0_{63} & 1_{62} & 1_{61} & \dots & 1_0 \\ \hline\end{array}$,

[1]「p の剰余類」とは p で割った余りの集合 $\{0,1,\dots,p-1\}$ のことです

すなわち $2^{63}-1=9223372036854775807$ が上限になります．しかし，その下限は 64 番目の bit のみが '1' で他が '0' となる対象 $\boxed{1_{63}}\boxed{0_{62}}\boxed{0_{60}}\boxed{\ldots}\boxed{0_0}$ になって，これが 2^{63} の「2 の補数」なので，結局，下限は $-2^{63}=-9223372036854775808$ です．この考え方を一般化すれば n-bit 長の整数は正整数が $n-1$ 個の bit を使って表現されるので，上限が $2^{n-1}-1$，下限は -2^{n} になります．このように自然数は流石に「**神様がお創りになられた数**」[2]だけあって計算機上でも「**自然**」に表現できますが，「**負の数というものは存在しない**」[3]ために補数を用いるという「**人工的な方法**」で表現することになるのです．

16 進数表示

ここで計算機内部では 2 進数で表現されていても，たとえば 64-bit 環境で long 型の整数は 64-bit 長，つまり '0' と '1' が 64 個並んだ対象になります．流石にそのまま入力や表示させることは効率が良いものではありません．だからもう少し人間に判り易いように内部表現の表示を 2 進数ではなく 16 進数で行います．実際，64-bit 長の整数で扱える最大の正整数 9223372036854775807 を 2 進数で表示すれば $\boxed{0_{63}}\boxed{1_{62}}\boxed{1_{61}}\boxed{\ldots}\boxed{1_0}$ と '1' が 63 個続きます．ところが 16 進数であれば '`7fffffffffffffff`' と 16 桁になって多少は実用的になります．これは後述の実数の表現でも同様です．

剰余類上の演算

ここでは n-bit 長の 2 進数で整数を表現するために 2^n による剰余で考えています．この 0 から 2^n-1 までの整数の集合を $\mathbb{Z}_{(2^n)}$ と記述します．また $\mathbb{Z}$ を整数の集合の表記とします．計算機での整数の和 "+" や積 "×" は全てこの世界 $\mathbb{Z}_{(2^n)}$ で考えますが，普通の整数の演算とは異なったことが生じます．たとえば '$n>1$' であれば '$a\times b=0$' となる 0 と異なる元 $a,\ b$，すなわち「**零因子**」と呼ばれる数が存在します．実際，$n>2$ のとき 2^{n-2} と 2^2 は 0 ではありませんが，これらの積は 2^n になって $\mathbb{Z}_{(2^n)}$ では 0 になります！また和についても正整数同士の和が負の整数になることがあります．たとえば $2^{n-1}-1$ と 2 の和は $(2^{n-1}-1)+2=2^{n-1}+1$ ですが，この $2^{n-1}+1$ は $2^{n-1}-1$ の「2 の補数」なので $-(2^{n-1}-1)$ になります！
つまり計算機上の整数演算は**計算機で表現可能な範囲内で処理が行われ，入力や結果が範囲外にあれば勝手に範囲内の整数に置換えられます**．もし入力や結果が領域を超

[2]自然数，全部創ったのは神様，その他全部は人様の創作 (Die ganzen Zahlen hat der liebe Gott gemacht, alles andrere ist Meschenwerk.) - Leopold Kronecker

[3]De Morgan は負の数を認めなかったそうですが，その気持も理解できませんか？

過する可能性があれば後述の「浮動小数点数」を用いるか, 自分で「多倍長整数」を定義する等の工夫が必要になります.

2.2.2 Yorick の整数の型

long 型の整数: ASCII 文字の “0” から “9” までの数字の羅列で表現される対象で, 通常の整数表現には long 型が対応します. なお, 型を明確に指定したければ通常の整数の表記末尾に “l” や “L” を ‘123L’ のように追加すれば long 型の整数として扱われます. また `long(⟨ 対象 ⟩)` でも long 型の対象が生成できます.
ここで long 型の整数は Yorick が動作する OS 環境に依存します. まず 64-bit 環境であれば, その上限は $2^{63}-1 = 9223372036854775807$, 下限は $-2^{63} = -9223372036854775808$ になります. ただし 32-bit 環境であれば後述の int 型と同じ領域になります.
整数の「8 進数表現」や「16 進数表現」も long 型の整数として扱います. Yorick の「8 進数表現」は整数の 8 進数表現の先頭に “0” を追加した表記, 同様に「16 進数表現」は整数の 16 進数表現の先頭に “0x” を付けた表記です. たとえば整数 75 の 8 進数表現は 113, 16 進数表現は $4B$ になりますが, Yorick では ‘0113’ と ‘0x4B’ と表記されて共に long 型の整数となります:

```
> 0113
75
> 0x4b
75
> print,typeof(0113),typeof(0x4b)
"long" "long"
```

ここで long 型以外の整数を利用するためには型を別途指定する必要があります.

int 型の整数: 上限が $2^{31}-1 = 2147483647$, 下限が $-2^{31} = 2147483648$ の整数に対して利用可能な符号付き整数の型です. 具体的には long 型表記の末尾に ‘1234n’ のように “n” や “N” を付けて指定します. また `int(⟨ 対象 ⟩)` でも int 型の対象が生成できます.
Yorick は int 型の整数を真理値として用います. すなわち真 (true) が int 型の ‘1’, 偽 (false) が int 型の ‘0’ で表現されます[4]:

```
> 1==1
1
> 3<2
0
```

[4] if 文や while 文では条件文の返却値が ‘0’ であれば偽, ‘0’ 以外の値を真として処理を行っています.

```
> typeof(1==1)
"int"
> typeof(3<2)
"int"
```

short 型の整数: 上限が $2^{15}-1=32767$, 下限が $-2^{15}=-32768$ の範囲の符号付き整数の型で, 数字の末尾に "s" や "S" を '123s' のように付けて指定します. この short 型の整数も `short(⟨ 対象 ⟩)` で生成できます.

char 型: char 型では 8-bit の正整数が表現できますが, その表記は他と違って印字可能な ASCII 文字の数字, あるいは 2 桁の 16 進数表記の先頭に "0x" を置いた対象の先頭に "\" を置き, 全体を単引用符 " ' " で括った対象です. この char 型の整数も `char(⟨ 対象 ⟩)` で変換できますが, 区間 $[0,255]$ に収まらない場合に 256 を法とした剰余類で返却されます:

```
> char(256)+0
0
> char(−257)+0
255
```

この例では char 函数で char 型に変換して long 型の 0 との和を取ることで 16 進表記ではなく理解し易い 10 進数に変換させています. 最初の '256' は 256 による剰余で考えるために '0' になります. 同様に '-257' は 256 を法とする剰余類の世界では $0 \equiv 256 \bmod 256$ と $-1 \equiv 0+(-1) \equiv 256-1 \equiv 255 \bmod 256$ から '255' を得ます. ここで char 型の整数は 16 進数に変換されてから表示されます:

```
> '\a'
0x07
> a='\a'
> typeof(a)
"char"
> 0x07
7
> typeof(0x07)
"long"
```

ここでの表示はあくまでも表示のみの問題で, 実際の型は long 型ではなく char 型であることに注意が必要です. ここで Yorick の特徴として "0" から "7" 迄の数字は "0x0" をその数字に付けた値が返されます:

```
>       ['\0','\1','\2','\3','\4','\5','\6','\7','\8']
[0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x38]
```

このように 0 から 7 迄の値は通常の ASCII コードとは異なります.
char 型は他の整数の型と同様に算術演算が可能で, 配列の添字としても利用できます. また画像ファイルを読込んだ結果の数値配列もこの char 型になり, pli 函数で配列を表示する場合は配列が char 型でなければなりません.

整数列の表現

整数の型は **char 型**, **short 型**, **int 型**と **long 型**で表現されますが, この整数を用いて構成された対象として **range 型**と呼ばれる対象があります.

range 型:　特殊な与件型としての range 型は $\langle$ 整数$_1\rangle$: $\langle$ 整数$_2\rangle$ や $\langle$ 整数$_1\rangle$: $\langle$ 整数$_2\rangle$: $\langle$ 整数$_3\rangle$ の書式を持つ Yorick の対象です. MATLAB や Octave で, このような表記は自動的に 1 次元配列に変換されますが, range 型の対象を Yorick に入力しても自動的に配列等に変換されません. この range 型の対象は単体では存在できない対象です. では何故必要なのでしょうか？ 実は函数の引数として価値があるからです:

```
> func a(x,m){if(is_range(m) && is_array(x)) return x(m);}
> a ([1,2,3,4,5],3:5)
 [3,4,5]
```

この例で定義した函数 a 第 1 引数を配列型, 第 2 引数を range 型とし, 単純に ‘a(x,m)’ で ‘x(m)’ を返却するだけの函数ですが, range 型が存在するお陰でこのような函数の定義が行えるのです.

2.2.3　実数の表現

浮動小数点数

計算機上で実数を表現するために「**浮動小数点数**」と呼ばれる表現が使われます. この浮動小数点数は一定の bit 長を持った対象で, 32-bit 長の「**単精度浮動小数点数**」, 64-bit 長の「**倍精度浮動小数点数**」等があります. ここでは浮動小数点数について簡単に解説することにしましょう.
まず, 浮動小数点数は「**符号部**」, 「**仮数 (小数) 部**」, 「**指数部**」の三部構成になっています. ここで符号部を構成する整数を s , 仮数部から構成される有理数を f, 指数部から構成される整数を ε , 基数を正整数 β とすると $(-1)^s \times f \times \beta^{\varepsilon}$ で対応する実数が復元できます. このときに符号部の s は ‘0’, あるいは ‘1’ が用いられて仮数部 f は正整数 $d_i,\ 0 \leq d_i < \beta,\ (i = 1, \ldots, n)$ から $d_1/\beta + \cdots + d_n/\beta^n$ により指数部 ε は

$\delta_1 + \delta_2 \times \beta + \cdots + \delta_m \times \beta^{m-1} - b$ で与えられます．ここで定数 b は「**下駄履き値**」と呼ばれます．浮動小数点数は「零」を中心に左右対称に分布することが構成方法からも判りますが, 浮動小数点数の濃度 (個数) は $2 \times \beta^{m+n} + 1$ で, $m,\ n$ の一方を ∞ としても高々 ${\aleph_0}^5$ です．しかし, 実数の濃度は $\aleph_0$ よりも大きな $\aleph$ なので浮動小数点数で実数を網羅することはできません．そのために浮動小数点数は近似値としての性格を持たなければならず, ここに精度の問題が生じる可能性があるのです．
この浮動小数点数には規格があります．代表的な規格として「**IBM 方式 (Excess 64)**」,「**IEEE 854**」や「**IEEE 754**」があります．最初の IBM 方式は IBM の汎用機で用いられ, IEEE 854 は基数を 10 とする規格, IEEE 754 が現在の殆どの計算機で用いられ, 策定当初の IEEE 754-1985 で基数を 2 とし, IEEE 754-2008 になると IEEE 854 の規格も包含しています．ここでは IEEE 754-1985(以降, 混乱の恐れがなければ IEEE 754 と略記) が策定する基数が 2 の浮動小数点数について述べます．なお詳細は「**Sun Microsystems ドキュメント**」[20] の浮動小数点数や IEEE に関連する項目, 概要は「**精度保証付き数値計算**」[1] や「**IEEE754 倍精度浮動小数点数のフォーマット**」[3] を参照して下さい．

IEEE 754 による基数が 2 の浮動小数点数

IEEE 754-1985 は 2 進数浮動小数点数のための規格であり,「**単精度**」,「**倍精度**」,「**拡張単精度**」と「**拡張倍精度**」という 4 種類の浮動小数点数, および浮動小数点数の「**四則演算**」:（“+”, “−”, “∗”, “/”),「**平方根**」:(“$\sqrt{\ }$”) と「**剰余**」や「**比較操作**」:(“>”, “≥”, “=”),「**整数と浮動小数点数間の変換**」や「**異なる浮動小数点書式間の変換**」, 浮動小数点数の基本書式と 10 進数変換と無限大等の「**浮動小数点例外**」と「**非数 (NaN)**」の扱いを定めているので, この規格に従っていれば異なった計算機環境でも同じ結果がある程度は期待できます．現実には三角函数や指数函数といった初等函数が IEEE 754 で規格化されていないために, これらの函数の処理でハードウェアやライブラリによって違いが生じる可能性が残っています．
ここで IEEE 754 で規定されている 2 を基数とする浮動小数点数の書式として,「**32-bit 長の単精度**」,「**64-bit 長の倍精度**」,「**拡張単精度**」と「**拡張倍精度**」の 4 種類があります．ここで計算機に実装すべき浮動小数点数は単精度のみで, 拡張単精度と拡張倍精度については単に満すべき最低限の bit 長と精度のみが定められています．たとえば「拡張単精度」が 43-bit 長以上,「拡張倍精度」が 79-bit 長以上です．ではこれらの精度をまとめておきましょう:

[5] 整数の濃度 (個数) です．

IEEE 754 浮動小数点数の言語毎の型

精度	**Yorick**	**C, C++**	**FORTRAN**
単精度	float	float	REAL, REAL*4
倍精度	double	double	DOUBLE PRECISION, REAL*8
拡張倍精度	なし	long double	REAL*16

ここで「浮動小数点数」は ‘0’ と ‘1’ の羅列, たとえば “01111110” のように固定された長さを持ち, この羅列を構成する ‘0’ と ‘1’ の総数を「**bit 長**」と呼び, この bit 長によって浮動小数点数が決定されます．たとえば bit 長が 32 であれば「**単精度**」, bit 長が 64 であれば「**倍精度**」の浮動小数点数です． そして, この列には番地が $\begin{array}{|c|c|c|c|c|c|c|}\hline a_{(m+n)} & a_{(m+n-1)} & \ldots & a_{(n)} & a_{(n-1)} & \ldots & a_{(0)} \\ \hline\end{array}$ のように割当てられ, ここで現われた m, n は, それぞれ指数部と仮数部を表現する ‘0’ と ‘1’ の列の総数, すなわち bit 長であり, 浮動小数点数の番地に対して $\begin{array}{|c|c|c|}\hline \text{符号部 } s_{[m+n:m+n]} & \text{指数部 } e_{[m+n-1:n]} & \text{仮数部 } f_{[n-1:0]} \\ \hline\end{array}$ の三部で構成されています．ここで表記 “$[a:b]$” は番地 a から b を利用することを意味します．したがって「**符号部**」は 1-bit 長, 「**指数部**」が m-bit 長, 「**仮数部**」が n-bit 長の 2 進数で, 浮動小数点数はこれらの 2 進数を並べた $m+n+1$-bit 長の 2 進数 a として表現されます．ここで仮数部の bit 長に 1 を加えた $n+1$ のことを「**浮動小数点数の精度**」と呼びますが, 1 を加える理由は仮数部の構造が関係します．ここでは符号部, 仮数部と指数部の順番で詳細を解説しましょう．

符号部:　符号部 s は最も大きな番地の bit の “$a_{(m+n)}$” に対応し, 表現する数値が正であれば ‘0’, 負であれば ‘1’ が指示されます．

指数部:　指数部は 2 進数 a の m 桁の部分 “$a_{(m+n-1)} \ldots a_{(n)}$” から構成されます．ここで $\delta_{(i)} = a_{(i+n)}$ $(0 \leq i \leq m-1)$ とするとき, 指数部によって表現される整数 e は $\delta_{(0)} \times 2^0 + \delta_{(1)} \times 2^1 + \cdots + \delta_{(m-1)} \times 2^{m-1}$ で与えられます．そして $\delta_{(0)} = 0$ と $\delta_{(i)} = 1$ $(1 \leq i \leq m-1)$ を満す指数部 e を「**最大許容指数**」と呼んで $e_{\max}$ と表記し, 1 から $e_{\max}$ の指数部 e の範囲を「正規化浮動小数点数」と呼ばれる通常の実数の表現で用います．また, 全てが 1, すなわち $\delta_{(i)} = 1$ $(0 \leq i \leq m-1)$ を満す指数部 $e_{\max}+1$ が「**無限大** ∞」や「**非数 NaN**」の表現で用いられ, 逆に全てが ‘0’, すなわち $\delta_{(i)} = 0$ $(0 \leq i \leq m-1)$ となる指数部 e は浮動小数点数の「零」や「非正規化浮動小数点数」と呼ばれる「零」周辺の実数を表現する浮動小数点数で用いられます．なお IEEE 754 では指数部が表現する整数 e は本来の指数 ε に定数 b を加えたもので表現され, 指数 ε は $e-b$ で与えられます．この「**下駄**」 b を履かせた指数部を「**下**

駄履き表示」と呼び, 定数 b を「**下駄履き値** (bias)」と呼びます．この下駄履き値 b は浮動小数点数の精度で異なります.

仮数部:　仮数部は 2 進数 a の一部となる n 桁の 2 進数 “$a_n \dots a_{(1)}$” で与えられますが実際の仮数部として用いられるのは, この 2 進数にさらに 1 桁を追加することで得られる 2 進数 $d = d_n d_{(n-2)} \dots d_{(1)} d_{(0)}$ です．ここで $d_{(i)} = a_{(i-1)}$ $(1 \le i \le n)$ とし, 残りの $d_{(0)}$ は指数部が表現する整数 e が ‘0’ でなければ $d_{(0)} = 1$, 整数 e が ‘0’ ならば $d_{(0)} = 0$ 定めます．また全てが ‘1’, すなわち $d_{(i)} = 1$ $(0 \le i \le n-1)$ を満す仮数部を $f_{\max}$ と表記します.
この仮数部が表現する有理数 f は $d_{(0)} + d_{(1)}/2 + \cdots + cd_{(n)}/2^n$ で与えられるので, $d_{(0)} = 1$ であれば “1.dd…”, $d_{(0)} = 0$ であれば “0.dd…” の書式の数になります．ここで $d_{(0)}$ のことを「**隠れ bit**」, あるいは「**暗黙の 1**」と呼びます．この理由は仮数部の実体が $n+1$-bit 長なのに表に出ているのが, この「隠れ bit」を除いた n-bit 長の部分であるためです．そして「隠れ bit」の浮動小数点数を「**規格化浮動小数点数**」と呼び, 隠れ bit が 0 の浮動小数点数を「**非規格化浮動小数点数**」と呼びます．この本では問題がなければ「規格化浮動小数点数」と「非非核化浮動小数点数」をそれぞれ「**規格化数**」,「**非規格化数**」と略記します．特に「非規格化数」は, 指数部を表現する整数 e が ‘0’ の場合に非規格化数に切換えられることから, 零の周囲を埋める数としての性格を持ちます, また規格化数から非規格化数の領域に移行することを「**段階的アンダーフロー**」, または「**漸近アンダーフロー**」と呼びます．ここで「隠れ bit」を加えた桁数 $n+1$ が規格化数の有効桁ですが, これを「**有効桁精度**」と呼びます．ところで非規格化数の場合は先頭が ‘0’ のために実質的に規格化数の有効精度よりも 1 つ低下した桁数 n になります．同様に指数部の下駄履き値も規格化数と非規格化数で異なります．まず b を正規化数の下駄履き値とすると, 非正規化数で用いられる下駄履き値 b_ν は $b-1$ で与えられます．なぜなら正規化数の絶対値での最小値と非正規化数の最大値を考えると正規化数の最小値が 2^{1-b}, 非正規数の最大値は $\left(\sum_{i=1}^{n} 1/2^i\right) \times 2^{-b_\nu}$ で与えられ, ここで $\sum_{i=1}^{n} 1/2^i = 1 - 1/2^n$ より, 結局, $(1 - 1/2^n) \times 2^{-b_\nu}$ になります．そして最小正規化数と最大非正規化数は隣接したものであるべきです．この要請から $b_\nu = b - 1$ でなければならないことが判ります.
この非規格化数は精度に関して規格化数に劣りますが, 規格化数よりも絶対値が小さいため, 非規格化数が扱えればそれだけ高精度の計算が行えます．この非規格化数は Yorick では扱えませんが, MATLAB や Octave では扱えます

零の表現:　IEEE 754 では指数部と仮数部を表現する整数が ‘0’ の場合, すなわち $(-1)^s \times 0$ で「零」を定義します．なお, 符号 s には ‘0’ と ‘1’ の場合があり $s = 0$ と

なる零を「**正の零**」$+0$, $s=1$ となる零を「**負の零**」-0 と表記します．そして，これらの零に対して $+0=-0$ であることが IEEE 754 で定められています．この符号付き零の規定は右極限や左極限を考慮する上で重要です．

表現可能な領域:　浮動小数点数で表現される領域はどうなるでしょうか？ まず整数の表現と異なり，零を挟んで左右対称に浮動小数点数が存在することが容易に判ります．それから正規化数の最大値は $x_{\max}=f_{\max}\times 2^{e_{\max}-b}$ で与えられ，零と異なる浮動小数点数の絶対値での最小値については，規格化数であれば $x_{\min}=2^{-b}$, 非規格化数であれば $\check{x}_{\min}=2^{-n-b_\nu}=2^{1-n-b}$ で与えられます．ここで b と b_ν をそれぞれ正規化数と非正規化数の下駄履き値とします．また規格化数の最小値 $x_{\min}=2^{-b}$ を越えて零に近づくと非正規化数に切替わって零の周囲の浮動小数点数が密になります．では実際に用いられる具体的な範囲を次の表に纏めておきましょう．なお小数点部は小数点下 5 桁で四捨五入して表示しています:

表現可能な領域

	単精度	倍精度
指数部の bit 長 (M)	8	11
仮数部の bit 長 (N)	23	52
正規化の下駄履き値 (b)	127	1023
非正規化の下駄履き値 (b_ν)	126	1022
最大正規化数	3.4028×10^{38}	1.7977×10^{308}
最小正規化数	1.1755×10^{-38}	2.2251×10^{-308}
最小非正規化数	1.4013×10^{-45}	4.9407×10^{-324}

浮動小数点数は有限個であるために表現し切れない実数が存在します．まず $|x|>x_{\max}$ となる数 x を「**桁溢れ (overflow)**」, $x_{\min}>|x|>0$ となる数 x を「**アンダーフロー (underflow)**」と呼びます．
なお仮数部を n-bit 長，指数部を m-bit 長とする浮動小数点数で表現可能な数の集合を $\mathcal{F}_{m,n}$, bit 長を予め固定した状態で問題としない場合には簡単に $\mathcal{F}$ と表記します．

丸めと切捨:　閉区間 $[-x_{\max},x_{\max}]$ に包含される実数でなければ浮動小数点数として表現できませんが，稠密で連続的な実数とは違い，浮動小数点数は離散的です．そのため近似によって実数を表現しなければなりません．
ここで ulp(Unit in the Last Place)[6] という函数を導入しましょう．この函数は実数

[6]「ウルプ」と呼びます

$x \in [-x_{\max}, x_{\max}]$ に対し，この実数 x を数直線上で挟む二つの浮動小数点数間の最小距離 ulp(x) を与える函数です．ulp(x) は x が大きな数値であれば大きくなり，逆に x が小さな数ならば小さな値を返します．たとえば整数 2^{52} から 2^{53} の間の整数を浮動小数点数で表現する場合，仮数部の最小の数が 2^{-52} となることから ulp は 1 になりますが，2^{53} を越える数に対して ulp は 1 よりも大になります．したがって絶対値が 2^{53} を越える整数は倍精度浮動小数点数でも一意に表現できません．このことを Octave[7]で確認しておきましょう:

```
octave:15> fprintf('%16.0f\n',2.0^53);
9007199254740992
octave:16> fprintf('%16.0f\n',2.0^53+1);
9007199254740992
octave:17> fprintf('%16.0f\n',2.0^53+2);
9007199254740994
```

ここで '2.0^53+1' が浮動小数点数 '2.0^53' と同じ値になっていますね．先程述べたように 2^{53} を越えた実数 x に対しては，その ulp(x) が 1 を越えてしまうために $2^{53}+1$ は浮動小数点数 '2.0^53.0' で近似されます．この例のように区間 $[-x_{\max}, x_{\max}]$ に含まれる実数 x を浮動小数点数 $\mathcal{F}$ の元で対応付けがあります．この対応作業は「**丸め**」や「**切捨**」と呼ばれ，次の 4 種類の操作が規定されています:

丸めと切捨

丸めの種類	概要
1. 上向きの丸め	a 以上の浮動小数点数の中で最小のものを採用:
	$\triangleright\, a \stackrel{def}{=} \min(\{\, x \in \mathcal{F} \;\wedge\; a \le x \,\})$
2. 下向きの丸め	a 以下の浮動小数点数の中で最大のものを採用:
	$\triangleleft\, a \stackrel{def}{=} \max(\{\, x \in \mathcal{F} \;\wedge\; a \ge x \,\})$
3. 最近値への丸め	a に最も近い浮動小数点数を採用:
	$\circledcirc\, a$ と表記
4. 切捨	絶対値が $\lvert a\rvert$ 以下で a に最近の浮動小数点数を採用:
	$\trianglelefteq\, a$ と表記

ここで演算 "$\wedge$" は「論理積 (連言)」で，論理式 $A \wedge B$ が真になるのは A と B が共に真の場合のみです．さて 1. と 2. の丸めによって実数 x と対応する浮動小数点 $\check{x}$ との差の絶対値は ulp(x) 以下，3. と 4. の操作による浮動小数点数からの距離は ulp(x)/2 以下になります．

[7]Octave を利用した理由は，Yorick の write 函数には表示桁に 15 桁の制約があるためです (9.3.1 参照).

丸めと切捨の性質:　演算子 $\bigcirc : \mathbb{R} \to \mathcal{F}$ を演算子 "$\triangleright$", "$\triangleleft$", "$\circledcirc$" か "$\trianglelefteq$" の何れかとします:

丸めの演算子の性質

1)	$\bigcirc\, x = x$	任意の $x \in \mathcal{F}$ に対して
2)	$x \leq y \Rightarrow \bigcirc\, x \leq \bigcirc y$	任意の $x,\ y \in \mathbb{R}$ に対し
3)	$\circledcirc\,(-x) = -(\circledcirc\, x)$	
4)	$\triangleleft\,(-x) = -(\triangleright\, x)$	
5)	$\triangleright\,(-x) = -(\triangleleft\, x)$	

性質 2) から丸めの演算子は「順序を保つ」ことが判ります．ここで比較の演算子 "$\leq$" を "$<$" で置換えることはできません．実際, $x \in \mathcal{F}$ に対して $0 < |x-y| < \mathrm{ulp}(y)/2$ を満す実数 $y \in \mathbb{R}$ は $\circledcirc y = x$ となって性質 2) を満さないためです．
浮動小数点数は本質的に「近似値」としての性格を持っていますが, この「近似値」が数値計算で意味を持つためには, 四則演算の結果もちゃんと近似値になっていなければなりません．すなわち丸めの演算子 "$\bigcirc$"$\in${"$\triangleright$", "$\triangleleft$", "$\circledcirc$", "$\trianglelefteq$"} に対し, 実数上での演算子 "$\circ$"$\in$ {"+","−","×","/"} と, それに対応する浮動小数点上の演算子 "$\bullet$"$\in${"$\oplus$","$\ominus$","$\otimes$", "$\oslash$"} との間には条件 6) を満たすことが IEEE 754 で要求されています:

丸めの演算子の性質 (四則演算との関係)

6)	$(\bigcirc x) \bullet (\bigcirc y) = \bigcirc\,(x \circ y)$

計算機イプシロン:　「**計算機イプシロン**」とは「$1.0 + \varepsilon \neq 1.0$ を満す最小の浮動小数点数 ε」として定義されます．具体的に Yorick で観察してみましょう:

```
> 2.0^(−52)
2.22045e−16
> 2.0^(−52)+1.0==1.0
0
> 2.0^(−53)+1.0==1.0
1
> 2.0^(−53)==0.0
0
```

この例で判るように倍精度での計算機イプシロンは 2^{-52} , つまり '2.^(-52)' で与えられ, 浮動小数点数 '1.' に最も近い規格化数は $1 + 2^{-52}$, すなわち '1.+2.^(-52)' で与えられます．計算機イプシロンと計算機が扱える最小の浮動小数点数とは別物で, '2.^(-53)' は '0' と異なるものの $\mathcal{F}$ で '2.0^(-53)+1.0==1' を満します．計算機イプシロンはあくまでも浮動小数点数 '1.0' から '1.0' よりも大きな浮動小数点数との距離

を意味し, 浮動小数点数が ‘1.0’ でなければ, 隣接する浮動小数点数との距離は計算機イプシロンと異なります. たとえば実数 2^{16} に近接する浮動小数点数はどのような数でしょうか？ 実数 2^{16} を表現する浮動小数点数 ‘2.^16’ の内部表現は $s = 0$, $f = 1$, $e = 16 + b = 1039$ となるので, 求める浮動小数点数は $s = 0$, $f = 1 + 2^{-52}$, $e = 1039$ を表現する ‘2.^(16)+2.^(-36)’ です. したがって浮動小数点数 ‘1.0’ の場合よりも間隔が広くなっています. このように双方の数値の大きさが極端に異なる場合はもちろんですが, 数値の大きさによっては内部的に極端に隣接することもあります. このように浮動小数点数による数値計算では表現の問題が出易いために「**数式どおり**」に計算するまえに内部表現や丸めによる真の値からの「**ズレ**」, すなわち「**誤差**」を予め考慮する必要があります. たとえば次の計算結果を眺めて下さい:

```
> 1.0+2.^(−52)∗5./6.==1.0
0
> 1.0+2.^(−52)∗5./6.==1.0+2.^(−52)
1
> 1.0+2.^(−53)+2.^(−54)==1.0
1
> 1.0+(2.^(−53)+2.^(−54))==1.0
0
> 1.0+2.^(−52)∗1./3.==1.0
1
```

最初の結果から計算機イプシロンは 2^{-52} ではないように思えませんか？残念ながら式 ‘1.0+2.0^(-52)*5./6.’ の値は式 ‘1.0+2.0^(-52)’ と同じ値の浮動小数点数になります. つまり, 実数 $1 + 2^{-52}$ に対応する浮動小数点数に「**丸められた**」からなのです. そして式 ‘1.0+2.^(-53)+2.^(-54)’ と式 ‘1.0+(2.^(-53)+2.^(-54))’ は数学的には同値でも左から順に処理されるために前者は浮動小数点数 ‘1.0’ に等しく, 後者では括弧の処理が優先されるために浮動小数点数 ‘1.0+2.^(-52)’ に丸められます. また式 ‘1.0+2.^(-52)*1./3.’ は式 ‘1.0+2.^(-52)’ が表現する浮動小数点数よりも近い浮動小数点数 ‘1.0’ に丸められています.

Yorick の浮動小数点数の与件型

さて計算機一般では実数は先程述べたように浮動小数点数で表現します. Yorick の場合, 扱える浮動小数点数は「**倍精度**」と「**単精度**」の二種類が基本で, 与件型はそれぞれ「**double 型**」と「**float 型**」です. なお, Yorick は様々な計算機環境の浮動小数点数の表現に対応できます. 詳細は §9.4.4 を参照して下さい. ただし MATLAB や Octave とは違って非規格化浮小数点数は扱えません.

double 型の実数:　倍精度浮動小数点数による実数表現は ASCII 文字の "0" から "9" 迄の数字と一つの小数点を表現するピリオド "." で構成された文字の羅列で '123.45' のような表記です．また, Yorick では C 風に '2.3e+4' のような E 表現も可能です．Yorick で実数は通常, 倍精度の double 型として処理されます．この double 型の対象は他の数値から `double(⟨ 対象 ⟩)` で生成できます．

float 型の実数:　float 型を利用するためには '1.23f' や '12.3e-4f' のように通常の実数表記に続けて "f" や "F" を付加しなければなりません．float 型の対象は他の数値から `float(⟨ 対象 ⟩)` で生成できます．

ieee.i ライブラリについて

Yorick には IEEE 754 に関連する "ieee.i" ライブラリがあります．このライブラリは include 函数を用いて `include,"ieee.i"` で予め読込む必要がありますが, このライブラリに含まれる大域変数によって Yorick の浮動小数点数に関する情報を容易に得ることができます．たとえば大域変数 native_dlim から Yorick が扱える最大の浮動小数点数と最小の浮動小数点数, 及び計算機イプシロンの情報が得られます:

```
> native_dlim
[1.79769e+308,2.22507e−308,2.22045e−16]
```

詳細は `help,ieee` から関連する項目を参照して下さい．

添字としての整数

Yorick の配列の添字には整数が用いられます．MATLAB や Octave と異なり, 与件型を厳密に見る仕様となっているために添字として与える与件の型に注意が必要になります．たとえば MATLAB や Octave では次の処理が可能です:

```
octave:4> a=[1,2,3]
a =

   1   2   3

octave:5> a(1.)
ans =  1
octave:6> a(2*sin(pi/2))
ans =  2
```

この処理ではベクトル '[1,2,3]' を生成して '1.' 番目の成分と '2*sin(pi/2)' 番目の成分を取出しています．ここで '1.' と '2*sin(pi/2)' はOctaveでは浮動小数点数の '1.' と '2.' です．このようにMATLABやOctaveでは少数点以下に0以外の数がない場合は特別に添字としても使えます[8]．ところがYorickで同様の処理を行うと叱られます:

```
> a=[1,2,3];
> a(1.)
ERROR (*main*) bad data type for array index
WARNING source code unavailable (try dbdis function)
now at pc= 1 (of 12),  failed  at  pc= 5
 To enter debug mode, type <RETURN> now (then dbexit to get out)
>
```

Yorickでは, 配列の添字や特定の型を引数として要求する函数に対して適合する対象を与えなければエラーになります．Yorickは他の言語と比べて与件の型について厳しい言語ではありませんが, MATLABが緩いこともあって, MATLAB言語で記述したプログラムをYorickに移植する場合に最も注意しなければならない点です．また, Yorickの異常終了の多くは配列の添字の処理に関わることも指摘しておきましょう．

2.2.4 複素数の表現

complex型: この型は実はstruct文を用いて構築された型で, 二つの倍精度の浮動小数点数の組合せとなっています．Yorickではcomplex型の数値は '123+45i' のように虚部となる数値の末尾に純虚数を表現する記号 "i" を付与するだけです．ただし記号 "i" は虚数表記の一部であって変数や定数ではありません．実部と虚部を取出す場合はCの構造体のと同様に末尾に ".re" や ".im" を付加すれば良いのです:

```
> a=123+4.5i
> typeof(a)
"complex"
> a.re
123
> a.im
4.5
> (123+4.5i).im
4.5
> (123+4.5i).re
```

[8] さすがにOctaveでも 'a(1.5)' はエラーになります．

```
123
> structof(a)
struct  complex {
  double re;
  double im;
}
```

この例では最初に複素数 $123 + 4.5i$ を ‘123+4.5i’ と表記し, この対象を変数 a に割当てて typeof 函数で調べています. それから ‘a.re’ で実部, ‘a.im’ で虚部を取出しています. ここで複素数の実部と虚部は ‘(123+4.5i).re’ や ‘(123+4.5i).im’ でも取出せますが, このときは記号 “( )” で複素数を括る必要があります. また整数, 有理数と実数は複素数に包含されるので, 表記 ‘(123).re’ は Yorick で意味を持ちそうに思えますが, 表記 ‘(123)’ は long 型であり, complex 型ではないので誤りになります. さらに structof 函数の結果から判るように complex 型は Yorick の構造体を用いて表現された与件型になります. この complex 型の対象の生成は `complex(⟨ 対象 ⟩)` でも行えます.

2.3　数値の変換函数

Yorick が扱える「**数値**」に対しては次の変換函数が存在します:

数値の型変換函数

構文	**概要**
char(⟨ 対象 ⟩)	char 型に変換
short(⟨ 対象 ⟩)	short 型に変換
int(⟨ 対象 ⟩)	int 型に変換
long(⟨ 対象 ⟩)	long 型に変換
float(⟨ 対象 ⟩)	float 型に変換
double(⟨ 対象 ⟩)	double 型に変換
complex(⟨ 対象 ⟩)	complex 型に変換

ここでの ⟨ 対象 ⟩ は数値です. なお算術演算子でより優位にある型からの変換では当然, 何らかの情報の欠落を伴う可能性があります. また char 型から long 型への変換や long 型から double 型への変換のように, 算術演算子で優位にある型への変換では優位にある型の単位元との積を計算する方法で型を変換させることもできますが, ここで述べる変換函数を用いた処理の方がやや高速です.
実際に確認しておきましょう:

```
> A=array(char(10),1000,1000)
> B=array(1,1000,1000)
> t0=array(double,3)
> t1=t0;
> timer,t0;for(i=1;i<101;i++){C=A*B;};timer,t1;t1-t0
[0.808051,0.512032,1.34023]
> timer,t0;for(i=1;i<101;i++){C=long(A);};timer,t1;t1-t0
[0.428027,0,0.42894]
```

この例では 1000×1000 の配列 A, B を生成し, 配列 A の与件型を char 型, 配列 B の与件型を long 型としています. ここで配列 A を long 型に変換させる方法の 1 つに long 型の配列 B との積を計算する方法もありますが, 成分単位の積を計算するために long 函数で変換するよりも時間がかかることが判ります. 同様に配列 A の double 型への変換も確認しましょう:

```
> B=array(1.0e+0,1000,1000)
> timer,t0;for(i=1;i<101;i++){C=A*B;};timer,t1;t1-t0
[1.11207,0.012001,1.16557]
> timer,t0;for(i=1;i<101;i++){C=double(A);};timer,t1;t1-t0
[0.432027,0.004,0.43535]
```

この例からも積演算で変換を行うと変換函数よりも多少, 時間がかかることが判るでしょう. これらの例では変換を 100 回実行することで漸く処理速度の違いが明瞭になっています. だからちょっとした変換を行う程度では, 処理速度の違いは明瞭に判らないでしょう.

char 函数: 与えられた数値を char 型の対象に変換する函数です. ここで対象が complex 型の場合は実部を char 型に変換し, float 型や double 型の場合は小数点以下を切り捨てます.

short 函数: 与えられた数値を short 型の対象に変換する函数です. ここで対象が complex 型の場合は実部を short 型に変換し, float 型や double 型の場合は小数点以下を切り捨てて short 型に変換します.

int 函数: 与えられた数値を int 型の対象に変換する函数です. ここで対象が complex 型の場合は実部を int 型に変換し, float 型や double 型の場合は小数点以下を切り捨てて int 型に変換します.

long 函数:　与えられた数値を long 型の対象に変換する函数です．ここで対象が complex 型の場合は実部を long 型に変換し, float 型や double 型の場合は小数点以下を切り捨てて long 型に変換します.

float 函数:　与えられた数値を float 型の対象に変換する函数です．ここで, 対象が complex 型の場合は実部を float 型に変換します.

double 函数:　与えられた数値を double 型の対象に変換する函数です．ここで, 対象が complex 型の場合は実部を double 型に変換します.

complex 函数:　与えられた数値を complex 型の対象に変換する函数です．対象が complex 型でなければ '0i' を加えた対象を返します.

2.4　文字列

2.4.1　文字列の表現

Yorick では文字列に関連する型として「**string 型**」だけがあります．char 型も ASCII 文字に対応していますが算術演算が可能な整数に包含されています．ところが string 型では演算子 "+" を文字の結合のために有していますが, この演算子 "+" は可換演算子ではありません．また文字列を扱う函数で文字列に含まれる文字を指示するために整数を利用しますが, このときに文字列の先頭は左端の文字で, この先頭の文字を指示する整数は配列と異なり 0 から開始します.

string 型:　函数内部での注釈や text_stream 型のストリームでやりとりで用いられる与件です．基本的に ASCII 文字[9]や各国の言語の文字を二重引用符 " " " で括った "1234" のような対象です．この string 型は数値表現とは全く違う型であり, さらに Yorick の函数名や変数名として使えない対象でもあります．ここで string 型には包含される特殊文字を纏めておきましょう:

[9]ASCII コード表に登録された文字

特殊文字

記号	概要	記号	概要
\n	改行	\t	タブ
\”	二重引用符	\’	単引用符
\\	バックスラッシュ	\a	ベル
\b	BS	\f	改頁
\r	CR		

これらの文字は通常の文字と組合せて, write 函数のように書式が指定可能な函数で利用できます．たとえば, 標準出力に「これでいいのだ」と出力する際にベルを鳴らせたければ `write,format="%s\n","これでいいのだ\a"` とすればよいのです．

2.4.2 Yorick で用いられる正規表現

strword 函数や strgrep 函数では「**正規表現**」を用いた文字列操作ができますが, これらの函数で用いられる正規表現は限定的なものです．ここでは Yorick で用いられる正規表現を簡単に説明します．まず「**正規表現**」は「**文字の並び**」の照合で用いられる Yorick の「**文字列による表現**」で, 「**文字の並び**」を表現するために「**メタ文字**」と呼ばれる文字から構成された文字列が用いられます．これらメタ文字を次に纏めておきます:

Yorick の正規表現で用いられるメタ文字

メタ文字	概要
\	メタ文字で用いられる文字に適合
.	任意の 1 文字に適合
[]	内部で指定した文字が取り得る領域に対して 1 文字が適合．領域の指定では記号 “-” で取り得る文字の領域, 記号 “ˆ” で除外すべき文字を指定
ˆ	文字列の先端に適合
$	文字列の末端に適合
|	論理和 (選言)
()	メタ文字の作用域を制限
*	0 個以上の文字に対して照合
+	1 個以上の文字に対して照合
?	0 個, あるいは 1 個の文字に対して照合

メタ文字 “\”: 正規表現のメタ文字を文字の並びの中の通常の文字として指定するときにメタ文字 “\” を先頭に置きます．たとえばメタ文字 “.” を文字の並び中の文字と指定するときに “\.” とします.

メタ文字 “.”: このメタ文字は与えられた文字列の任意の 1 文字に適合します.

メタ文字 “[]”: このメタ文字の内部に上述の文字の領域を指定する表現を記述することで, この表現が文字列中の一つの文字に適合します.
ここでメタ文字 “[]” と併用するメタ文字は “-” と “^” の 2 つで, メタ文字 “-” で文字の領域指定が行え, ある文字や領域を除外する場合にはメタ文字 “^” を用います. たとえばアルファベットの小文字の a から n までが適合する場合は”[a-n]” とします. これに加えて大文字の C から X までを適合させる場合は ”[a-nC-X]” と表記し, さらに数字の 3 から 8 までを適合させる場合は ”[a-nC-X3-8]” とします．同様に小文字の a から n までを除外する場合は ”[^a-n]”, さらに大文字の C から X も除外する場合は”[^a-n^C-X]”, 加えて 3 から 8 を除外させるには”[^a-n^C-X^3-8]” となります.

**メタ文字 “^” とメタ文字 “$:”** メタ文字 “^” が文字列の先頭に適合し, メタ文字 “$” が文字列の末尾に適合します.

メタ文字 “|”: このメタ文字は「論理和 (選言)」を意味します．つまり ”a|b” で正規表現 a か正規表現 b の何れかが適合することを表現します.

メタ文字 “()”: このメタ文字は正規表現のメタ文字による作用域を制限するために用います．たとえば ”Y(O|o)rick” は文字列 ”YOrick” と文字列 ”Yorick” の双方に適合しますが, 正規表現”YO|orick” は文字列 ”YO” と文字列 ”orick” の双方に適合すること, すなわち正規表現”(YO)|(orick)” の意味になります.

メタ文字 “*”, メタ文字 “+” とメタ文字 “?”: これれの文字はメタ文字 “[]” 等で一つの文字に適合する正規表現に続けて用いることで正規表現の反復を意味します. まず “*” は直前に置かれた一つの正規表現に対して 0 個以上の文字との照合を行うことを意味し, “+” は直前に置かれた正規表現に対して 1 個以上の文字との照合を行うことを意味します．そして最後の “?” は直前の正規表現に対して 0 個か 1 個の文字との照合を行うことを意味します.
正規表現の扱いの詳細は正規表現を用いる函数の解説で再び議論します.

2.4.3 文字列操作

Yorick は各種の文字列操作函数を保持しています. ここでは文字列操作の演算子や函数について述べることにしますが, 文字列は配列とは異なり, 先頭の文字に整数 0 を対応させます.

文字列の結合

文字列結合に関連する演算子と函数

構文	概要
$\langle$ 文字列$_1\rangle$ + $\langle$ 文字列$_2\rangle$	二項演算子 "+" を用いた記法
$\langle$ 配列 $\rangle$(.., sum, ..)	添字 "sum" を用いた Yorick 独自の記法
sum($\langle$ 配列 $\rangle$	sum 函数を用いた記法

演算子 "+": 二つの string 型の与件は演算子 "+" で結合できます:

```
> "1234"+"56789"
"123456789"
```

演算子 "+" の二つの被演算子の双方が string 型でなければエラーになります. また文字列の結合の演算子 "+" は非可換演算子[10]です.

添字 "sum" と函数 sum: これは与えられた文字列で構成された配列に対して総和を取る操作を行います:

```
> A=[[["A","B"],["C","D"]],[["E","F"],["G","H"]]]
> dimsof(A)
 [3,2,2,2]
> A(sum,..)
 [["AB","CD"],["EF","GH"]]
> A(,sum,)
 [["AC","BD"],["EG","FH"]]
> A(,,sum,)
 [["AE","BF"],["CG","DH"]]
> sum(A)
"ABCDEFGH"
```

このように文字列配列の添字に sum を用いることは数値配列の場合と同様です:

[10] $a \circ b \neq b \circ a$ となる二項演算子 "$\circ$" のこと

$$A(\underbrace{,\ldots,}_{k-1}\overset{k}{\check{\text{sum}}},\underbrace{\ldots,}_{n-k-1}) = \sum_{i}^{N_k} A(i_1,\ldots,i_{k-1},\overset{k}{\check{i}},i_{k+1},\ldots,i_n)$$

ここでの総和記号 "$\sum$" は結合演算子 "+" による文字列の結合を意味します.
このときに配列 A の大きさが $N_1 \times \cdots \times N_n$, 添字 "sum" を第 k 次添字としたときに得られた配列の大きさは $N_1 \times \cdots \times N_{k-1} \times N_{k+1} \times N_n$ です.
その一方で sum 函数を用いると全ての成分を配列 A を平坦化した状態で結合して一つの文字列として返却するため, 0 次の配列を返却する函数とも言えます.

文字列の長さを求める函数

文字列の長さは文字列を構成する文字数が対応します. ここで "\t" のような特殊文字は 1 文字として扱われます:

文字列の長さを求める函数

構文 (strlen)
strlen(⟨ 文字列 ⟩)

strlen 函数:　与えられた文字列の長さを long 型の与件として返却する函数です.
特殊な string 型の対象"", すなわち nil は長さ 0 の文字列で, "\t" のような特殊文字は構成する文字が "\" と "t" の 2 つですが, ASCII 文字では 1 文字のために全体で一つとして加算されます:

```
> strlen("abc\t")
4
> strlen("abc \a")
5
```

文字列の変換や置換

与えられた文字列の変換を行う函数を纏めておきましょう:

文字列の変換や置換を行う函数

構文 (strcase, strchar, streplace)
strcase(⟨ 整数 ⟩, ⟨ 文字列 ⟩)
strcase, ⟨ 整数 ⟩, ⟨ 変数 ⟩
strchar(⟨ 配列 ⟩)
streplace(⟨ 文字列$_1$⟩, [⟨ 整数$_1$⟩, ⟨ 整数$_2$⟩], ⟨ 文字列$_2$⟩)

strcase 函数: 第 2 引数に与えられた ⟨ 文字列 ⟩ を第 1 引数で指定した数値で大文字や小文字に変換する函数です. 第 1 引数の ⟨ 整数 ⟩ が 0 のみであれば第 2 引数の ⟨ 文字列 ⟩ を小文字に変換し, 0 以外の整数値が与えられると第 2 引数の ⟨ 文字列 ⟩ を大文字に変換します. なお引数を括弧 "( )" で括らない場合, 第 2 引数には変換すべき文字列や文字列を成分とする配列が割当てられた変数で函数を充当し, その処理結果は第 2 引数の変数にポインタを介して代入されます:

```
> y=strcase(1,"abc");y
"ABC"
> strcase,0,y;y
"abc"
```

strchar 函数: 配列が string 型であれば各成分を char 型の配列に変換し, char 型の配列が与えられれば各成分を string 型に変換した配列を返却する函数です:

```
> x=strchar("ABC")
> x
[0x41,0x42,0x43,0x00]
> y=strchar(x)
> y
"ABC"
```

streplace 函数: 第 1 引数で指定した ⟨ 文字列$_1$⟩ に対して ⟨ 整数$_1$⟩ を始点, ⟨ 整数$_2$⟩ を終点とする ⟨ 文字列$_1$⟩ の領域を ⟨ 文字列$_2$⟩ で置換える函数です:

```
> streplace("abcde",[2,3]," BC")
"abBCde"
```

この例では文字列 "abcde" の先頭から 2 + 1 番目から 3 + 1 番目の文字を "BC" で置換えます. ここで文字列では配列とは違って先頭を 0 とするために [2,3] の 2 が先頭から 3 番目の文字, [2,3] の 3 が先頭から 4 番目の文字に対応します.

空白文字の処理に関連する函数

空白文字の処理に関連する函数を纏めておきましょう:

空白文字の処理に関連する函数

構文 (strword, strtrim, strtok)
strword(⟨ 文字列 ⟩)
strword(⟨ 文字列$_1$⟩, ⟨ 文字列$_2$⟩, ⟨ 整数 ⟩)
strword(⟨ 文字列$_1$⟩, ⟨ 整数$_1$⟩, ⟨ 文字列$_2$⟩, ⟨ 整数$_2$⟩)
strtrim(⟨ 文字列 ⟩)
strtrim(⟨ 文字列 ⟩, ⟨ 整数 ⟩)
strtrim(⟨ 文字列 ⟩, ⟨ 整数 ⟩, blank =)
strtok(⟨ 文字列$_1$⟩, ⟨ 文字列$_2$⟩, ⟨ 整数 ⟩)
strtok(⟨ 文字列$_1$⟩, ⟨ 文字列$_2$⟩)
strtok(⟨ 文字列 ⟩)

strword 函数:　strword 函数は第 1 引数に与えた文字列に対して, 空白文字の直後の文字の位置と総数の対を成分とするベクトルで返却する函数です. ここで空白文字は既定値として " ", "\n" や "\t" が与えられています:

```
> strword("  123")
[2,5]
> strword("  123 ")
[2,6]
> strword("123 ")
[0,4]
```

この例で示すように引数が一つだけであれば引数の文字列に対して文字列の先頭からの空白の文字数と全体の文字数の対で構成されたベクトルを返却します. ここで空白文字は ⟨ 文字列$_2$⟩ で正規表現のメタ文字を用いて指定することもできます:

```
> strword("123\n\t678 ")
[0,9]
> strword("123\n\t678 ","1−3")
[3,9]
```

この例では文字列 "123\n\t678 "に対して, 既定値の空白文字を用いた場合と, 空白文字として正規表現 "1-3"によって 1 から 3 までの数字を指定した場合の空白文字数と全体の文字数の対の違いを示すものです.
第 3 引数の ⟨ 整数 ⟩ を指定することで, 第 1 引数に与えた文字列から第 2 引数で指定する空白文字を ⟨ 整数 ⟩ で指定した回数分, 検出させます. ここで空白文字を既定値のままで利用し, 第 3 引数の整数を指定する場合には空白文字や nil を第 2 引数とします:

```
> strword("123\n\t12341234 ","1−3")
[3,14]
> strword("123\n\t12341234 ","1−3",1)
[3,14]
> strword("123\n\t12341234 ","1−3",2)
[3,5,8,14]
> strword("123\n\t12341234 ","1−3",3)
[3,5,8,9,12,14]
> strword("123\n\t12341234 ","1−3",4)
[3,5,8,9,12,14,14,−1]
> strword("123\n\t12341234 ",nil,4)
[0,3,5,13,14,−1,14,−1]
```

ここでの例では空白文字を '1', '2' と '3' にしています. 最初の 2 つの例では第 3 引数を指定しないものと, 第 3 引数に 1 を指定したものの結果が同じものになることを示しています. 以降, 第 3 引数に 2 を指定することで次の空白文字の直後に来る文字の位置と次の空白文字群の末端の位置を検出しています.

strtrim 函数: Yorick 言語で記述された函数で, 第 1 引数の ⟨ 文字列 ⟩ の空白文字 (" ", "\t" と "\n") を削除する函数です. この函数内部では strword 函数が用いられており, 空白文字の指定は strword 函数の第 2 引数の表記に準じます. この表記にいては strword 函数と正規表現を参照して下さい. 次に, strtrim 函数の第 2 引数の整数は 1, 2, 3 の何れかを指定し, これらの整数には次の意味があります:

strtrim 函数の第 2 引数の意味

1	文字列先頭の空白文字を削除
2	文字列末尾の空白文字を削除
3	文字列の先頭と末尾の空白文字を削除

ここで第 2 引数が無指定であれば 3 と同じです.
それから空白文字は blank キーワードで無指定であれば " " , "\t" と "\n" が既定値となります. blank キーワードの指定には正規表現も使えます. たとえば "a" から "c"

までの文字を削除するための正規表現は "a-c" になります:

```
> strtrim("abcdefghijklmn",3,blank="a−c")
"defghijklmn"
> strtrim("abcdefghijklmn",3,blank="l−n")
"abcdefghijk"
```

strtok 函数:　第 1 引数で与えられた ⟨ 文字列$_1$⟩ を第 2 引数の ⟨ 文字列$_2$⟩ を区切文字として分割した配列を返却する函数です. 第 2 引数が無指定の場合に区切文字は空白文字 " ", タブ "\t" と 改行 "\n" になり, 第 3 引数の整数値を指定しなければ区切文字が最初に現われた箇所の左側と残りの右側で文字列を分解します. 整数値を指定した場合, その整数値の成分の文字列ベクトルを返却します. ただし区切文字の個数が指定した整数値よりも少なければ残りの成分には 'nil' で置換えられます:

```
> strtok("12 34 56 78")
["12","34 56 78"]
> strtok("12 34 56 78"," ",4)
["12","34","56","78"]
> strtok("12 34 56 78"," ",7)
["12","34","56","78",( nil ),( nil ),( nil )]
```

文字列の切出に関連する函数

文字の切出に関連する函数

構文 (strpart)
strpart(⟨ 文字列 ⟩, ⟨ 領域 ⟩)
strpart(⟨ 文字列 ⟩, [⟨ 整数$_1$⟩, ⟨ 整数$_2$⟩])
strpart,⟨ 変数 ⟩, [⟨ 整数$_1$⟩, ⟨ 整数$_2$⟩]

strpart 函数:　第 1 引数の ⟨ 文字列 ⟩ から第 2 引数で指示した文字列の領域や位置を示す配列から部分文字列を取出す函数です:

```
> strpart("abcdABCD",2:3)
"bc"
> strpart("abcdABCD",[3,5])
"dA"
> strpart("abcdABCD",[[2,3],[3,5]])
["c","dA"]
```

文字列の検出

文字列の検出に関連する函数を次に纏めておきます:

文字列の検出を行う函数

構文 (strfind, strgrep, strglob, strmatch)
strfind(⟨ 文字列$_1$⟩, ⟨ 文字列$_2$⟩, n=, case=, back=)
strgrep(⟨ 並び ⟩, ⟨ 文字列 ⟩)
strgrep(⟨ 並び ⟩, ⟨ 文字列 ⟩)
strglob(⟨ 並び ⟩, ⟨ 文字列 ⟩)
strglob(⟨ 並び ⟩, ⟨ 文字列 ⟩, ⟨ 整数 ⟩)
strmatch(⟨ 文字列$_1$⟩, ⟨ 文字列$_2$⟩)
strmatch(⟨ 文字列$_1$⟩, ⟨ 文字列$_2$⟩, ⟨ 整数 ⟩)

strfind 函数: ⟨ 文字列$_1$⟩ が ⟨ 文字列$_2$⟩ に包含されるときに始点と終点で構成されるベクトルを返却する函数です. ここでの始点は 0 から開始することに注意が必要です:

```
> strfind("12","123456789")
 [0,2]
> strfind("56","123456789")
 [4,6]
```

もしも ⟨ 文字列$_1$⟩ が ⟨ 文字列$_2$⟩ に包含されないは ⟨ 文字列$_2$⟩ の長さと-1 の対のベクトルを返却します:

```
> strfind("x","123456789")
[9,−1]
```

strgrep 函数: 「正規表現」で指定した「文字の並び」が与えた文字列に含まれている場合は適合する文字数と 1 で構成されるベクトルを返却し, そうでない場合は第 2 引数の文字列の長さと-1 で構成されるベクトルを返却する函数です.

strglob 函数: ⟨ 並び ⟩ には UNIX の正規表現を用い, その正規表現に ⟨ 文字列 ⟩ が適合する場合に 1 を返却し, そうでない場合に 0 を返却する函数です. ここで ⟨ 並び ⟩ は UNIX での正規表現が用いられます. 具体的には次のメタ文字が許容されています:

利用可能なメタ文字

メタ文字	概要
*	0 から 1 個以上の全ての文字に適合
?	全ての 1 文字に適合
[]	1 文字に適合し, 内部には領域を示す正規表現を記述

簡単な例で解説しましょう:

```
> strglob("?.text","naa.text")
0
> strglob("?.text","n.text")
1
> strglob("*.text","naa.text")
1
```

この例は “?” と “*” の違いを示すもので, “?” は任意の 1 文字に適合するため, “?.text” は “n.text” に適合するものの “naa.text” は “naa” が 3 文字のために適合しませんが, “*.text” の場合は末尾が “.text” であれば全てが適合します. 次に “[]” の例を示しましょう:

```
> strglob("[a−m]*.text","ann.text")
1
> strglob("[a−m][^b]*.text","ann.text")
1
> strglob("[a−m][^b][^n]*.text","ann.text")
0
```

この例では正規表現を用いています. まず記号 “-” を用いて領域が指定可能です. 除外を行う場合には “^” を用いますが, strglob 函数では strgrep 函数のような正規表現ではなく, UNIX の正規表現となります. たとえば通常の正規表現で ”[a-m]*” は複数の文字に対して ”[a-m]” の照合を行いますが, UNIX の正規表現では, 先頭の文字が ”[a-m]” に適合し, 以降の文字は, 末尾が”.text”となる個所を除いて全ての文字が適合します.
ここで strglob 函数のキーワードには次のものがあります:

strglob 函数のキーワード

キーワード	概要
case=	既定値は 1
path=	1 の場合に記号 “/” は文字 “/” に適合する. 2 の場合に記号 “.” は文字 “.” に適合する. この既定値は 0
esc=	0 の場合に記号 “\” は ESC として扱わない. 既定値は 1

strmatch 函数: ⟨ 文字列$_1$⟩ に ⟨ 文字列$_2$⟩ が包含されるかどうかを判断する真理函数です. 第 3 引数の整数値が 0 の場合, 大文字と小文字を判別しますが, 0 以外の整数値であれば大文字と小文字を判別せずに判断を行います:

```
> strmatch(”ABCD”,”AB”)
1
> strmatch(”ABCD”,”Z”)
0
> strmatch(”ABCD”,”ab”)
0
> strmatch(”ABCD”,”ab”,0)
0
> strmatch(”ABCD”,”ab”,1)
1
```

この例で判るように第 3 引数が無指定の場合は 0 の大文字と小文字を判別することを指定して処理が行われます.

2.5 配列

Yorick の配列は MATLAB とその互換アプリケーションと比較して非常に柔軟にできています. MATLAB はその名前の示すように行列処理が容易になるように工夫されています. これに対し Yorick の配列の考え方はテンソルを根底に置いているために行列演算は独特の添字表記で記述する必要があります. そして, この点が Yorick で行列処理を行う上での障害となっていますが, この Yorick の表記に慣れてしまえば非常に柔軟で高度な処理ができます.
なお Yorick の配列の与件型は MATLAB の行列と同様に配列を構成する各成分の与件型の優位度で一意に決定されます. そのために MATLAB と同様に数値と文字列が混同した配列は生成できません. このような与件が必要な場合, MATLAB と同様に構造体を用いるか LISP 風のリストを用いることになります.
ここでの配列の生成については §3, 配列の演算については §4 を参照して下さい.

2.6 LISP 風のリスト

2.6.1 Yorick のリスト

Yorick には LISP 風の「リスト」も実装されており, このリストは単純な木構造のみを持っています. Yorick のリストはベクトルに似ていますが, 第 1 成分の与件型が全

体に波及する性質を持しません. したがって数値, 配列, 文字列や範囲といった対象が混在するリストが生成可能です.
この本では _lst(⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩) で生成されるリストを '(⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩)' と LISP 風に表記します. ただし Yorick のリストは LISP のリストと比較して単純なもので, 他の対象のように演算子 "==" を用いてその同値性を確認することはできません.

2.6.2 Yorick のリストを生成する函数

ここではリストを生成したり, リストの複製を行う函数を解説しましょう. なお Yorick ではリストを処理してリストを返す函数の名前の先頭に記号 "_" が置かれているので, 函数の性質が判り易くなっています:

LISP 風のリストを生成する函数

構文 (_lst, _cat, _cpy)
_lst(⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩)
_cat(⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩)
_cpy(⟨ 対象 ⟩, ⟨ 正整数 ⟩)
_cpy(⟨ 対象 ⟩)

_lst 函数: LISP 風のリストを Yorick の対象から直接構成する函数で, 後述の _cat 函数のようなリストの結合を行わずに引数全てを成分とするリストを生成します. なお Yorick のリストは配列とは異なり, 対象の型について均質的である必要はありません:

```
> neko=_lst("テスト 1",123,sin,[1,2,3])
> typeof(neko)
" list "
```

この例では文字列, 数値, 函数と 1 次元配列で構成されたリストを定義しています.
なお '_lst(nil)' や '_lst([])' で生成される空リスト "()" は空配列 '[ ]' や 'nil' とは別物です.

_cat 函数: LISP 風のリストを Yorick の対象から直接構成する函数で _lst 函数と同様の性質を持ちます. この函数名は catenate に由来する函数で本来は Yorick のリストを結合して一つのリストを返す函数ですが, 通常の Yorick の対象を引数とすると, これらの対象を成分とするリストを生成します.
具体的な例で _lst 函数と _cat 函数の違いを確認しておきましょう:

```
> a1=_cat(1,2,_lst (3,4),5);
> a2=_lst(1,2, _lst ( _lst (3,4)),5);
> _len(a1)
5
> _len(a2)
4
```

最初の a1 では_cat 函数を用いています．この a1 を LISP 風に記述すると '(1 2 3 4 5)' が対応します．この a1 に対して_lts 函数を用いた a2 は '(1 2 (3 4) 5)' が対応し，これが長さを返す函数_len による結果の違いに繋がります．

_cpy 函数: LISP 風のリストの複製を生成する函数です．第 2 引数の ⟨ 正整数 ⟩ はリストの先頭から複製すべき成分数になりますが，未指定の場合は全てを複製します:

```
> neko=_lst(1,2,3,4)
> mike=_cpy(neko,2)
> _car(_cdr(mike))
2
> _cdr(_cdr(mike))
[]
> mike2=_cpy(neko)
> _car(_cdr(mike2))
2
> _car(_cdr(_cdr(mike2)))
3
> _car(_cdr(_cdr(_cdr(mike2))))
4
```

リストを LISP 風に表現すると，ここでの例では mike にリスト '(1 2 3 4)' の先頭から二つの成分で構成されたリスト '(1 2)' が割当てられ，第 2 引数を指定しない mike2 にはリスト '(1 2 3 4)' が割当てられています．

2.6.3 リスト処理のための函数

Yorick にはリスト処理の基本的な処理ために LISP の length 函数，car 函数や cdr 函数に相当する基本的な函数を持っています．これらの函数の構文を纏めておきます:

リスト処理の基本函数

構文 (_len, _car, _cdr)
_len(⟨ 対象 ⟩)
_car(⟨ 対象 ⟩, ⟨ 正整数 ⟩)
_car(⟨ 対象 ⟩)
_cdr(⟨ 対象 ⟩, ⟨ 正整数 ⟩)
_cdr(⟨ 対象 ⟩)

_len 函数:　LISP の length 函数と同じ働きをする函数で, リストの長さを返します.

_car 函数:　引数が 1 個であれば LISP の car 函数と同じ働きをする函数です. 引数が 2 個であれば第 2 引数の正整数で指定した成分を返します.

_cdr 函数:　引数が 1 個であれば LISP の cdr 函数と同じ働きをする函数です. 引数が 2 個であれば第 2 引数の正整数で指定した成分以降を含むリストを返します. すなわちリストを '(1 2 3 4 5 6 7)' とするときに '_cdr(_lst(1,2,3,4,5,6,7),3)' にリスト '(4 5 6 7)' が対応します. このことを実際に確認しておきましょう:

```
> _car(_cdr( _lst  (1,2,3,4,5,6,7),3))
4
```

これらの最も基本的な函数に加え, 次に纏めたリスト処理の函数を持っています. ここで特記すべき函数は LISP の map 函数に相当する _map 函数でしょう:

リスト処理の函数

構文 (_prt, _rev, _nxt, _map)
_prt, ⟨ リスト ⟩
_rev(⟨ リスト ⟩)
_nxt⟨ リスト ⟩)
_map(⟨ 函数 ⟩, ⟨ リスト ⟩)

_prt 函数:　引数として 1 個のリストを取り, そのリストの成分を表示する函数です.

_rev 函数:　引数として 1 個のリストを取り, そのリストの成分の順序を逆にしたリストを返す函数です. つまり, リスト $(\langle$ 成分$_1\rangle, \ldots, \langle$ 成分$_n\rangle)$ に _rev 函数を作用さた結果, リスト $(\langle$ 成分$_n\rangle, \ldots, \langle$ 成分$_1\rangle)$ が得られます.

```
> a2=_rev(_lst (1,2,3,4))
> _prt,a2
 list  items:
   4
   3
   2
   1
```

_nxt 函数: 引数として 1 個のリストを取り, そのリストの第 1 成分を返して残りのリストを引数として与えた変数に代入する函数です. 函数内部では_car 函数の値を返却値として引数の変数に_cdr 函数の値を代入しています:

```
> test=_lst (1,2,3,4)
> _nxt(test)
1
> _prt,test
 list  items:
   2
   3
   4
```

この例で示すように引数 test の値が書換えられる副作用があることに注意が必要です. ここで引数として変数を与える必要は必ずしもなく, '_next(_lst(1,2,3))' のような処理も問題がありません.

_map 函数: LISP の map 函数に対応する函数で第 1 引数で与えた Yorick の 1 変数函数を第 2 引数で指定したリストに作用させます:

```
> a1=_map(sin,_lst (1,2,3,4,5,6))
> _prt,a1
 list  items:
   0.841471
   0.909297
   0.14112
   −0.756802
   −0.958924
   −0.279415
```

この例ではリストに sin 函数を作用させた結果を prt 函数で表示させています.

2.7 構造体

Yorick は struct 文を用いて C 風の「**構造体**」が利用できます:

struct の構文

struct	⟨ 構造体名 ⟩ {
	⟨ 既存の (型/構造体)$_1$⟩ ⟨ 変数名$_1$⟩;
	⟨ 既存の (型/構造体)$_2$⟩ ⟨ 変数名$_2$⟩;
	...
	⟨ 既存の (型/構造体)$_n$⟩ ⟨ 変数名$_n$⟩;}

この構造体の定義では long 型, double 型, pointer 型といった Yorick に組込まれた与件型のほかに利用者が定義した構造体も利用することができます．ここで構造体の要素としての変数を「**成員変数**」と呼びます.
では構造体の簡単な例を示しましょう．まず実数値の実験データとその注釈で構成される新しい構造体 datum は struct 文を使って定義できます:

```
> struct datum {
cont> double dat;
cont> string cm;
cont> }
```

ここで構造体の内容を知りたければ構造体名をそのまま入力します．また typeof 函数を使えば対象が構造体の定義であるかどうかが判別できます:

```
> datum
struct  datum {
  double dat;
  string  cm;
}
> typeof(datum)
" struct_definition "
```

構造体に対応する対象の生成は ⟨ 対象 ⟩ = ⟨ 構造体名 ⟩(成員変数$_1$ = ⟨ 値$_1$⟩, ...) によって行えます．また ⟨ 対象 ⟩ = ⟨ 構造体名 ⟩() や ⟨ 対象 ⟩ = `array`(⟨ 構造体名 ⟩) からも生成が行えます:

```
> test0=datum(dat=1.0,cm="test");
> test0
datum(dat=1,cm="test")
> a=datum()
> a.dat=1
> test1=datum();
```

```
> test2=array(datum);
> test1==test2
1
> test1
datum(dat=0,cm=(nil))
```

また構造体の成員変数に値を代入したければ $\boxed{\langle\ 構造体\ \rangle.\langle\ 成員変数\ \rangle = \langle\ 値\ \rangle}$ で行えます:

```
> test2=datum();
> test2.dat=1.0e-8
> test2.cm="test1, 2008/11/27,10:00:30";
> test2
datum(dat=1e-08,cm="test1, 2008/11/27,10:00:30")
> typeof(test2)
"struct_instance"
```

そして配列を用いるときに $\boxed{\langle\ 配列の型\ \rangle\langle\ 成員変数名\ \rangle(\langle\ 大きさ_1\rangle,\ldots,\langle\ 大きさ_n\rangle);}$ で定義します. たとえば `double point(15,10,5);` で大きさ $15 \times 10 \times 5$ の配列が成員として加えられます. また既存の構造体を用いるときも Yorick の型を用いる場合と同様です:

```
> struct cdata{ datum d1;int i;}
> c=array(cdata)
> c.d1=test1;
> c.i=1;
> c
mike(d1=datum(dat=1e-08,cm="test1, 2008/11/27,10:00:30"),i=1)
> (c.d1).dat
1e-08
```

既存の構造体の対象を配列の成分として利用するときは通常の配列の定義が行えます:

```
> struct test1{double x,y;}
> struct test2{test1 xy(10);}
> a=array(test2)
> a
test2(xy=[test1(x=0,y=0),test1(x=0,y=0),test1(x=0,y=0),test1(x=0,y=0),test1(x=
0,y=0),test1(x=0,y=0),test1(x=0,y=0),test1(x=0,y=0),test1(x=0,y=0),test1(x=0,y=
0)])
>
```

複合的な構造体を用いるときは括弧 “()” も使えます. たとえば ‘(c.d1).dat’ は構造体 c から成員変数 d1 の値を取り出し, その値 ‘c.d1’ から成員変数 dat の値を取り出しています. もちろん ‘c.d1.dat’ でも同様の処理が行えます. なお成員変数の値の取出では get_member 函数も使えます:

```
> struct PET{string cat,dog;}
> MyPET=PET(cat="Mike",dog="Pochi")
> get_member(MyPET,"cat")
"Mike"
```

なおバイナリファイル処理で用いられる stream 型の対象も構造体に似た構造を持っています (§9.4.4 参照).

2.7.1 構造体と配列

構造体をその成分とする配列の生成は他の配列の生成と違いはありません. 多少, 勝手か異なるのは構造体の成員変数に配列を割当てる場合です. たとえば次の構造体を定義しましょう:

```
struct Cat{string name; int order;};
```

それから 'myCat=array(Cat)' で Cat 型の配列を生成し, 私の飼い猫の Mike と Tama の順位が 1 番, 2 番だとします. ここで安易に 'myCat.name=["Mike","Tama"]' と入力するとエラーが出ます:

```
> struct Cat{string name; int order;};
> myCat=array(Cat)
> myCat.name=&["Mike","Tama"]
ERROR (*main*) cannot convert rhs of assign = to pointer or string
WARNING source code unavailable (try dbdis function)
now at pc= 1 (of 17), failed at pc= 12
 To enter debug mode, type <RETURN> now (then dbexit to get out)
```

このように構造体の成員に配列を直接代入することができません. 構造体の成員に配列を代入したければ pointer 型による値の引渡しを利用します. この myCat では猫の名前と順位の双方に配列を用いたいために name も order の双方は pointer 型にしなければなりません. また pointer 型を介するときには番地の引渡しとなるために直接配列を割当てるのではなく, 配列の番地を引渡すために記号 "&" を対象の先頭に付け, 値を参照する場合には変数の先頭に記号 "*" を付けます:

```
> struct Cat{pointer name; pointer order;};
> myCat=array(Cat)
> myCat.name=&["Mike","Tama"]
> myCat.order=&[1,2]
> myCat
Cat(name=0x74d740,order=0x74d778)
```

```
> print,*myCat.name,*myCat.order
["Mike","Tama"] [1,2]
```

2.8　函数

2.8.1　Yorick の函数の概要

Yorick の「**函数**」には二種類あります. 一つは与件型が **buildin** 型, もう一つは与件型が **function** 型の対象です. ここでの「**函数**」は双方を意味しますが, 誤解を招かない場合には「**function** 型」の対象を特に「**函数**] と呼び, 「**buildin** 型」を「**組込函数**」と呼びます. また変数と括弧を除いた函数のことを「**函数名**」と呼ぶことにします. また函数名と変数で構成されて Yorick で評価される文のことを簡単に「**函数項**」と呼びます. たとえば Yorick の式 'sin(pi/2)+1' の中の "sin(pi/2)" が「函数項」, 函数項で "(pi/2)" を除外した "sin" が「函数名」で, この「函数名」"sin" を typeof 函数で調べると 'buildin' が返却されるので「組込函数」であることが判ります.
Yorick の函数では戻り値を必要としない函数では引数を括る括弧 "( )" を省略できます. すなわち函数項が $f(x_1,\ldots,x_n)$ であれば $f,\ x_1,\ldots,x_n$ と表記できるのです. この括弧を省略した表記では, 演算子 "=" による函数の返却値を変数に割当することができなくなりますが, 後述のポインタを用いた方法で値を返却することができます. ここでは簡単な例を示しておきましょう:

```
> func test1(x){x=x+1;return(x);}
> test1(10)
11
> test1,10
>
```

この例では引数に 1 を加えたものを返却する函数 test を定義し, それに対して 2 通りの表記で処理を行っています. 'test(10)' では結果が表示されていますが, 'test,10' では結果が表示されていませんね. そのために変数への割当で問題が生じます:

```
> a1=test1(19)
> a1
20
> a2=test1,19
SYNTAX: syntax error near ,19
SYNTAX: syntax error near <EOF>
```

このように 'a2=test1,19' のような割当はできません. この構文で変数に結果の割当を行うためにはポインタを利用しなければなりません:

```
> func test2(&x){x=x+1;return(2*x);}
> x1=1
> a1=test2(x1)
> [x1,a1]
 [2,4]
> test2,x1
> x1
3
```

この例の函数 test2 は return 函数とポインタによる二種類の値の返却があります. 通常の括弧を省略しない表記では双方が使えますが, 括弧を利用しない表記はポインタによる返却のみが使えます.

2.8.2　function 型の函数の定義

function 型の函数は func 文を使って生成されます:

函数の生成

func ⟨ 函数名 ⟩	(⟨ 変数$_1$⟩, . . . , ⟨ 変数$_m$⟩)
⟨ 注釈 ⟩	
{	
	⟨ 文$_1$⟩;
	. . .
	⟨ 文$_n$⟩;
}	

函数の引数:　函数定義で指示できる引数 ⟨ 変数 ⟩ の書式には次の三種類があります:

引数の種類

引数の種類	函数定義での表記	引数としての表記例
通常の引数	x	x
ポインタ	&x	x
キーワード	x=	x=1

一つ目の引数の表記は通常の表記方法で函数内部の処理で利用する変数の初期値を定めるために用いられます. この表記は C や FORTRAN でもお馴染の通常の表記です. 二つ目の引数の表記は pointer 型で引渡される引数の表記方法です. この場合は C と同様に定義の段階で変数名の先頭に記号 “&” を付けます. ただし, この記号 “&” は

函数定義のときだけで, 実際の利用で引数にわざわざ “&” を付与する必要はありません.
三つ目の引数の表記はキーワード変数と呼ばれる変数を用いる表記で定義の際に “x=” のように変数名のうしろに記号 “=” を付与したものです. この変数は省略可能な引数であることを示し, 既定値を変更するときに用いられます.

函数の解説と注釈　函数の解説は func 文の中で函数名と引数の並びに続けて記入します. この解説は後述の help 函数で表示される “/* DOCUMENT” で開始して “*/” で終了する文で複数行にわたっても構いません. 実際に help 函数で表示されるのは Yorick の検索経路上にあるライブラリファイルに記述した場合だけで, この場合は help 函数が該当個所をファイルから取出して表示し, メモリ上に展開された函数から取出すものではありません.
函数の通常の注釈は “/*” から開始して “*/” で終える複数行にわたる文や, 一行だけであれば “//” で開始する文も使えます.
ここで簡単な例を示しておきましょう:

```
func iceil(a)
/* DOCUMENT iceil
 *
 * ceil 函数の値を long 型の整数で返す函数.
 */
{
 // long 函数で long 型に変換.
 return long( ceil(a));          /* 値の返却 */
};
```

この内容を直接入力したり, 適当なファイルに記述して読込むことができます. 特にファイルに記述して include 函数や require 函数で読込めば help 函数で函数名を引数として与えた場合に ‘/* DOCUMENT ... */’ に記述した解説が表示されます:

```
> include,"test.i"
> help, iceil
 /* DOCUMENT iceil
  *
  * ceil 函数の値を long 型の整数で返す函数.
  */
 defined at:  LINE: 1  FILE: /home/yokota/Yorick/test.i
```

文:　ここで文は通常の Yorick の入力可能な式です．函数内部の注釈以外の文の末端にはセミコロン “;” を置きます．Yorick では基本的に上から順番に文が実行され, 函数の返す値は return 文で明示的に指示しなければ最後に値を返却する文の値が函数の返却値になります.
文は記号 “{ }” を使って纏めることができます．この記号 “{ }” の使い方は C と同様で, if 文, for 文や While 文で処理内容を記述した文が複数あるときは記号 “{ }” で括って纏めなければなりません.

変数の宣言文:　Yorick でも C 風に変数の宣言が行えます．ここでの変数の宣言は函数内部で利用する変数が局所変数であることを明示的に宣言する local 文, 函数内部で割当や代入, あるいは参照で用いる変数が大域変数であることを明示的に宣言する extern 文があります:

函数内部での変数宣言文

構文 (local, extern)
local $\langle$ 変数$_1\rangle, \ldots, \langle$ 変数$_n\rangle$;
extern $\langle$ 変数$_1\rangle, \ldots, \langle$ 変数$_n\rangle$;

Yorick では函数やライブラリ等にて extern 文で宣言されていない変数は局所変数として扱われるために local 文は変数の局所性を明示的にする程度で実質的には不要です. また extern 函数はライブラリの制御変数として用いる大域変数の定義で用いることが可能です．この目的のために函数定義のような書式で help 函数向けの解説を入れることができます．たとえば次の内容のファイル (“test.i”) を記述したとしましょう:

```
extern neko;
/* DOCUMENT neko -- これはテストのための変数で深い意味はありません。
 *
 */
```

このファイルを一度 include 函数で読込むと help 函数で解説を読むことができるようになります:

```
> include,"test.i"
> help,neko
 /* DOCUMENT neko -- これはテストのための変数で深い意味はありません。
  *
  */
 defined at:  LINE: 1  FILE: /home/yokota/Yorick/test.i
>
```

この手法を使って buildin 型の函数の解説が “std.i” ライブラリ等で記述されています.

2.8.3 処理結果の返却

処理結果を戻す方法としては次の三種類の代表的な方法が挙げられます.

return 文を利用する方法: 返却値を return 文を用いて明示的に示すこともできます. ここで返却値が代入された変数を x とすると “return(x)” や “return x” の双方の表記が可能です.

```
> func tama(x){neko=2*x+1;return neko;}
> x1=tama(10)
> x1
21
```

return 函数は func 内部の任意の場所に置けますが, return 文ののうしろに続く文は処理されないことに注意が必要です:

```
> func test(x, &y) {return 2*x^2+1;y=array(0,4);print,y;}
> test(2,y)
9
> y
[]
```

この例から判るように return 文のうしろの文は全て実行されていません.

ポインタを利用する方法: C のようにポインタ を介して値の返却ができます:

```
> func test(x,&y){
cont> y=array(x,3,3);
cont> return(2*x^2);
cont> }
> test(123,z2)
30258
> z2
 [[123,123,123],[123,123,123],[123,123,123]]
```

このポインタを介することで複数の値を同値に返却させることが容易に行えますが, 構造体をポインタで引き戻す場合には注意が必要です. たとえば次の例を考えましょう:

```
struct mike{double x,y ;};
struct tama{mike a,b;}
func a1(&m){
```

```
 m.x=1;
 m.y=2;};
func a2(&n){
 tmp=array(mike);
 a1(tmp);
 n.a=tmp;
 a1(n.b);};
```

最初に構造体 mike と mike を使った構造体 tama を定義し, それから構造体 mike 型を持つ与件に対する代入函数 a1, この函数 a1 を用いた構造体 tama 型を持つ与件に対する代入函数 a2 を定義しています. ここで注意して頂きたいことは函数 a2 内部の構造で, 構造体 tama を構成する成員 a については array 函数を使って mike 型の対象 tmp を生成し, この tmp に対して函数 a1 を作用させる方法を採っているのに対し, 成員 b に対しては直接函数 a1 を作用させていることです. では動作を確認しましょう:

```
> x1=array(tama)
> a2(x1)
[]
[]
[]
> x1
tama(a=mike(x=1,y=2),b=mike(x=0,y=0))
```

x1.a への代入はちゃんと処理されていますが, x1.b への代入はできていません. このように親の手続内部で構造体を生成すると, その構造体に対してはポインタを介した値の引渡しができますが, 構造体の生成と実際の値の代入が行われる手続の関係が 2 段以上空くと値の引渡しに失敗します. ただし通常の変数であれば, このような現象は生じません:

```
> func a1(&m){m=m*2;}
> func a2(&m){m=m*3;}
> func a3(&m){a1(m);a2(m);}
> func a4(m){a3(m);return m;}
> a4(1)
[]
[]
[]
6
```

この例では函数 a1, a2, a3 がポインタによる値の返却を行い, 函数 a4 で return 文による値の返却を行います. 函数 a4 で用いる変数は m のみですが, この場合には問題

なくポインタを介した値の返却が利用できています.
このことからポインタを利用する場合は親子よりも関係が離れることを避けるように注意すべきです. もし親子よりも間隔が空く場合, 親子関係が維持できるように中間的な変数を生成するべきです.

大域変数を利用する方法: 函数内部で extern 文を使って宣言した大域変数に値を代入する方法です:

```
> neko=1;
> func tama(x){neko=2*x+1;}
> tama(10)
[]
> neko;
1
> func tama(x){extern neko;neko=2*x+1;}
> tama(10);neko
[]
21
```

ここでの例では変数 neko は extern 文を用いていないために函数 tama の局所変数の neko の値が反映されていません. ところが新しい函数 tama 内部で extern 文による宣言を行ったために今度は函数内部の値がちゃんと反映されています.
大域変数の宣言は函数外部でも行えますが, 値の引渡しを行う場合は函数内部で参照を行わない限り値の書換は行われません:

```
> extern _Z;
> func tama(x){_Z=2*x+1;print,_Z};
> tama(10)
21
[]
> _Z
[]
> func tama(x){print,_Z;_Z=2*x+1;};
> tama(10);
[]
[]
> _Z
21
```

この例で示すように最初の函数では単純に大域変数_Z に $2*x+1$ の値を代入させてから print 函数でその値を表示させていますが大域変数_Z の書換は生じていません.
一方で二番目の例では大域変数_Z の値を print 函数で表示させてから代入を行っているために今度は書換が生じています.

このように大域変数を函数内部から参照するだけであれば extern 文は不要ですが大域変数の書換を行う場合は予め函数内部で extern 文を使って宣言しておくか, 値の参照を行ってから書換を行う必要があります.
以上から判るように, 函数内部でも明示的に大域変数の宣言を行なっておく方が大域変数の値の書換処理を行う場合には無難です.

2.8.4 函数定義に関連する函数

函数定義に関連する函数

構文 (symbol_def, funcdef, funcset, disassemble)
symbol_def(⟨ 文字列 ⟩)
symbol_set,⟨ 文字列 ⟩,⟨ 対象 ⟩
funcdef(⟨ 文字列 ⟩)
funcset(⟨ 変数$_1$⟩,⟨ 値$_1$⟩,...,⟨ 変数$_8$⟩,⟨ 値$_8$⟩)
disassemble(⟨ 函数名 ⟩)

symbol_def 函数:　函数を表象として扱えるようにする函数です. symbol_def 函数は引数として既存の函数名を ⟨ 文字列 ⟩ として取って函数を返却します. この返却された与件に引数を与えることで, 函数として利用することが可能です:

```
> alpha=symbol_def("sin");
> alpha(pi/4)
0.707107
```

symbol_set 函数:　基本的に ⟨ 文字列 ⟩ で指示されａた変数に ⟨ 対象 ⟩ を割当・代入する函数です:

```
> symbol_set,"x",1
> x
1
```

funcdef 函数:　文を二重引用符で括ることで構成された ⟨ 文字列 ⟩ を処理する無名函数を生成する函数です. ここで ⟨ 文字列 ⟩ は次の書式でなければなりません:

funcdef 函数の引数の書式

” ⟨ 函数名 ⟩ ⟨ 引数$_1$⟩ ... ⟨ 引数$_n$⟩ ”

ここでの引数は変数名, 10 進整数, 浮動小数点数や文字列に限定されます. また一つの文字列を funcdef の引数として引渡すことから, 文字列に対しては “\”” で括る必要があります:

```
> funcdef("write \"これはテストです \"");
 これはテストです
```

演算子は funcdef ではそのまま反映されないため, 別途, 函数を用いることになります. 実際, 変数に対して値を割当・代入を行うために函数 funcset が用意されています:

```
> extern _X,_Y;
> _X=2;_Y=3;
> func Add(x,y){print, x+y;}
> funcdef("Add _X _Y")
5
> funcdef("funcset _Z 4")
> _Z
4
```

funcset 函数: ⟨ 変数$_i$⟩ に対して ⟨ 値$_i$⟩ を割当・代入する函数です. funcdef 函数では通常の代入・割当の演算子 “=” が使えないために funcset 函数を併用することで変数への代入・割当を実行します. この funcset 函数で代入・割当可能な変数は最大 8 個です.

disassemble 函数: 引数として function 型の函数名を取り, 函数内部での処理の流れを string 型のベクトルとして返却する函数です. ここで disassemble 函数がサブルーチンとして呼ばれており, 引数が nil であれば親プログラムに対して dissasemble 函数が作用します.

2.9 対象や与件の情報を返す函数

Yorick の対象に対して情報を返す函数を纏めておきます:

対象や与件の情報を返す函数

構文 (typeof, nameof, info, structof)
typeof(⟨ 対象 ⟩)
nameof(⟨ 対象 ⟩)
info(⟨ 対象 ⟩)
info, ⟨ 対象 ⟩
structof(⟨ 対象 ⟩)

typeof 函数:　Yorick の対象の与件型を調べる函数です．この函数の構文は typeof(⟨ 対象 ⟩) のみで info 函数のような括弧 “( )” を外した表記は使えません．

nameof 函数:　引数の対象が函数ときに函数名を文字列で返却します:

```
> func test(x){return 2*x;};
> alpha=test;
> nameof(alpha);
"test"
```

構造体の概要を返却する structof 函数と nameof 函数を併用すれば構造体の名前を返却させることができます:

```
> struct CG{double xg,yg;};
> a=array(CG);
> structof(a)
struct CG {
  double xg;
  double yg;
}
> nameof(structof(a))
"CG"
```

この手法は structof と組合せたときだけ有効で, ‘nameof(a)’ だけでは構造体名ではなく nil が返却されます.

info 函数:　対象の型を調べる函数で, typeof 函数とは違い第 2 の括弧 “( )” を外した表記も許容します．typeof 函数が対象の与件型のみを返却したのに対し, info 函数では対象の与件型に加え, 配列であればその大きさ, 函数であれば函数名と引数の情報も返します.
ここでは引数として配列と函数を与えたときの様子を示しておきます:

- 配列のとき

```
> info(a)
 array(long,10,9,8)
[]
```

配列が long 型で大きさが $10 \times 9 \times 8$ であることを示しています.

- 函数のとき

```
> func test(x,&y){
cont> y=array(x,3,3);
cont> return(2*x^2);
cont> }
> info(test)b
func test(x,&y)
[]
```

函数の引数が x と y で第 2 引数が pointer 型であることを示しています.

structof 函数: 引数の構造体としての情報を返す函数です:

```
> struct cdata{ datum d1;int i;}
> c=array(cdata,10)
> structof(c)
struct cdata {
  datum d1;
  int i;
}
```

typeof 函数による判別で struct 型ではない配列の対象に対しては “struct 与件型 { }” という書式を返却します:

```
> a=1
> structof(a)
struct long {
}
> structof("1234")
struct string {
}
```

2.10　型に関連する述語函数

2.10.1　基本的な述語

Yorick には対象の型に関連する述語函数があります．これらの函数は対象が特定の型の対象の場合に真, すなわち 1 を返し, それ以外は偽, すなわち 0 を返す函数です．最初に “std.i” ライブラリに含まれている基本的な述語について纏めておきましょう:

型に関連する基本的な述語

構文	概要
is_array(⟨ 対象 ⟩)	対象が配列 (array) の場合に 1 を返す函数.
is_func(⟨ 対象 ⟩)	対象が function 型の場合に 1, 組込の函数の bulit-in 型の場合に 2, autoload 函数で読込まれる autoload 型の函数の場合に 3 を返す函数.
is_list(⟨ 対象 ⟩)	対象がリスト (list) の場合に 1 を返す函数.
is_range(⟨ 対象 ⟩)	対象が「値域」(range) の場合に 1 を返す函数
is_stream(⟨ 対象 ⟩)	対象が stream 型の場合に 1 を返す函数.
is_struct(⟨ 対象 ⟩)	対象が構造体 (struct) の場合に 1 を返す函数.
is_void(⟨ 対象 ⟩)	対象が void 型, すなわち ‘nil’ や ‘[ ]’ の場合に 1 を返す函数．値が束縛されていない自由変数であれば 1 を返却
am_subroutine()	引数を必要としない函数で, 他の函数から呼出された場合に 1, それ以外は 0 を返却.

2.10.2　yutils パッケージに含まれている述語

“std.i” ライブラリ以外にも述語は定義されています．ここでは yutils パッケージに含まれる述語について述べます．この yutils パッケージは KNOPPIX/Math 2009 以降で収録された Yorick には含まれていますが, MS-Windows 版の Yorick には含まれていないので, 利用するためにはインストールを行う必要があります．詳細は §7.11 を参照して下さい:

utils.i に含まれる述語

構文	概要
is_scalar(⟨ 対象 ⟩)	対象がスカラーであるかどうかを判定.
is_vector(⟨ 対象 ⟩)	対象がベクトルであるかどうかを判定.
is_matrix(⟨ 対象 ⟩)	対象が 2 次元配列であるかどうかを判定.
is_real(⟨ 対象 ⟩)	対象が浮動小数点数であるかを判定.
is_complex(⟨ 対象 ⟩)	対象が complex 型であるかを判定.
is_integer(⟨ 対象 ⟩)	対象が long 型, int 型, short 型か char 型の何れかであるかを判定.
is_numerical(⟨ 対象 ⟩)	対象が整数か実数の何れかであることを判定.
is_integer_scalar(⟨ 対象 ⟩)	対象が整数のスカラーであることを判定.

第3章　配列について

Hamlet

Alexander died,
Alexander was buried,
Alexander returneth into dust;
the dust is earth;
of earth we make loam;
and why of that loam,
whereto he was converted,
might they not stop a beer-barrel?

ハムレット

死せるアレキサンダー大王、
葬られるや、
塵芥に還る;
その塵芥は土塊だ;
土塊から粘土を捏ね;
それから粘土故に,
ついに彼は変じて、
酒樽の蓋となってしまうのか？

Hamlet: 第五幕, 第一場

3.1　配列の概要

3.1.1　配列に関連する用語と表記について

$n \geq 1$ 個の自然数 $m_k,\ (1 \leq k \leq n)$ から定まる集合 $M_k = \{1, \ldots, m_k\},\ (1 \leq k \leq n)$ に対し, その直積集合 $M \stackrel{def}{=} M_1 \times \cdots \times M_n$ から Yorick の数や文字列といった対象 D への写像

$$
\begin{array}{cccccc}
A & : & M & \to & D \\
 & & \cup & & \cup \\
 & & (i_1, \ldots, i_n) & \mapsto & A(i_1, \ldots, i_n)
\end{array}
$$

を「n **次元配列**」A と呼びます. そして直積集合 M を「**添字集合**」, 添字集合の元 $(i_1, \ldots, i_n) \in M$ を「**添字**」と呼び, 添字の k 番目の成分 i_k のことを「k 次の添字」と呼びます. また添字 $(i_1, \ldots, i_n) \in M$ に対応する対象 $A(i_1, \ldots, i_n)$ を配列 A の添字 $(i_1, \ldots, i_n)$ に対する成分, あるいは簡単に配列 A の $(i_1, \ldots, i_n)$ 成分と呼びます. この本では配列の成分の表記として $A(i_1, \ldots, i_n)$ の他に $A_{i_1,\ldots,i_n}$ も用います.

ここで添字集合 $M_1 \times \cdots \times M_n$ の 1 次から n 次の添字の最大値から構成される数列 $m_1 \cdot m_2 \cdots m_n$ を「**配列の大きさ**」と呼び, 各次の添字の大きさが判るように $m_1 \times m_2 \times \cdots \times m_n$ と表記します. また, この数列の長さを「**配列の次元**」と呼びます.

この本では配列の表記として $\{A\}_{i_1=1,\ldots,i_n=1}^{m_1,\ldots,m_n}$ や $\{A\}_{m_1,\ldots,m_n}$ によって配列 A が大きさ $m_1 \times \cdots \times m_n$ の n 次元配列であることを明記するために用います. 同様に配列の添字と大きさを明記するために $\{A(i_1, \ldots, i_n)\}_{m_1,\ldots,m_n}$ や $\{A_{i_1,\ldots,i_n}\}_{m_1,\ldots,m_n}$ といった表記も用います.

なお, Yorick の配列の添字は '1' から開始しますが, 特別な添字として '0' と '負の数' があります. たとえば k 次の添字 $i_k \in \{1, \ldots, m_k\}$ に対して '0' を指示することは m_k を指示することと同値で, 一般的に正整数 h に対して $i_k = -h$ は $i_k = m_k - h$ と同値です. これらの表記によって利用者が添字集合の大きさを明確に把握していなくても, うしろ側から成分指定ができるのです.

3.1.2　具体的な配列の姿

Yorick の配列は記号 "[ ]" で括られた対象で, この記号 "[ ]" による括られ方で配列の次元や大きさが定まります. また配列の与件型も配列の成分の与件型の優位度によって一意に定まります. ここでは簡単な配列である空配列, 0 次元配列と 1 次元配列について解説します.

Yorick では特別に **nil** と表記され, ここでは「**空の配列**」とも呼びます. この配列の与件型は void 型で, 次元と大きさは 0 ではなく '[]', すなわち 'nil' になります. このことを dims 函数を使って確認しましょう:

```
> nil==[]
1
> typeof(nil)
"void"
> dimsof([]);
[]
```

0 次元配列: Yorick では**数や文字列は単独で 0 次元の配列**になります:

```
> dimsof(1)
[0]
> dimsof("abcd")
[0]
```

dimsof 函数の結果が全て '[0]' であることから, Yorick の数や文字列は 0 次元の配列としての性質を持ちます. このことは Yorick が配列を基礎的な与件とした言語であることを示しています.

1 次元配列 (ベクトル): Yorick の数, あるいは文字列だけで構成された対象の列 $a_1, \ldots, a_n$ を記号 "[ ]" で括ることで構成された配列 '$[a_1, \ldots, a_n]$' のことです. 1 次元配列は特別に「**ベクトル**」と呼ぶことにします. ここで Yorick 附属のマニュアルでは 1 次元ベクトルとリストを区別せずに用いていますが, Yorick には LISP 風のリストが別途存在するため, ここでは両者を区別して **1 次元配列をベクトルと呼び, リストとは呼びません**. 1 次元配列は記号 "[ ]" を用いた最も基本的な配列であり, 多次元配列は記号 "[ ]" を用いて帰納的に定義, すなわち, 記号 "[]" を使って捏ね回すことで得られるのです. 配列の添字は 1 から開始しますが, 添字 0 と負の整数も使えます. ここで添字を 0 とすると, 添字の次数の大きさが n であれば, 添字に n を指示することと同値です. また添字 $-i$ はうしろから i 番目の成分, つまり添字 $n-i$ に相当します:

```
> x=[1,2,3,4,5]
> print,x(-1),x(0),x(1)
4  5  1
```

この表記は実は Python でも見られる表記です. この他にも Python には Yorick で見られる配列の処理方法が取り入れられています. ここでは配列を創り出す演算子としての記号 "[ ]" に着目して次の操作を実行してみましょう:

```
> dimsof(1)
[0]
> dimsof([1])
[1,1]
> dimsof ([[1]])
[2,1,1]
> dimsof ([[[1]]])
[3,1,1,1]
```

整数 1 は 0 次元の配列です．この整数 1 を使って構成した配列 '[1]' は dimsof 函数によると '[1,1]' の大きさの配列ですが, これは成分が 1 個の 1 次元の配列であることを意味します．配列 '[[1]]' と配列 '[[[1]]]' はそれぞれ 2 次元の 1×1 の配列と 3 次元の $1 \times 1 \times 1$ の配列であることが分ります．このように **Yorick の配列では数値や文字列といった対象を括る記号 "[ ]" の深さが配列の次元に対応する**ことに注目して下さい.

記号 "[]" の作用:　記号 "[]" の配列への作用を確認しておきましょう.

```
> dimsof ([1,2,3]);
[1,3]
> dimsof ([[1,2,3]]);
[2,3,1]
> dimsof ([[[1,2,3]]]);
[3,3,1,1]
> dimsof ([[[[1,2,3]]]]);
[4,3,1,1,1]
```

最初の配列 '[1,2,3]' は成分が 3 個の 1 次元配列 (ベクトル) です．このベクトルを記号 "[ ]" で括った配列 '[[1,2,3]]' は 3×1 の 2 次元配列となり, 以後, 全体を記号 "[ ]" で覆うことで次元は 1 つ上り, dimsof 函数が返すベクトルでは右側に 1 が追加されて行くことが判ります．では複数の成分を持つ配列の成分側に記号 "[ ]" を追加するとどうなるでしょうか？

```
> dimsof ([[1],[2],[3]]);
[2,1,3]
> dimsof ([[[1]],[[2]],[[3]]]);
[3,1,1,3]
> dimsof ([[[[1]]],[[[2]]],[[[3]]]]);
[4,1,1,1,3]
```

配列 '[[1],[2],[3]]' は dimsof 函数によると '[2,1,3]' の配列, すなわち 1×3 の大きさの 2 次元配列になります．以降, 配列の成分に括弧を 1 つ増やすことでも次元が 1 つ上り, dimsof 函数の返却するベクトルの次数の左側に 1 が追加されます.

したがって配列 a の添字 $(i_1, \ldots, i_k)$ の長さ k, つまり次元が記号 "[ ]" による深さに対応し, 左端の 1 次の添字が記号 "[ ]" による階層で最下段の成分の個数, 右端の n 次の添字が記号 "[ ]" による階層で最上段の成分の個数に対応することが分ります.
つまり, **記号 "[ ]" が Yorick の配列の構造を決定します**. これは MATLAB 系の言語と大きく異なる点です:

```
octave:1> b =[[1,2],[2.,3],[3+3* i ,3]]
b =

   1 + 0i   2 + 0i   2 + 0i   3 + 0i   3 + 3i   3 + 0i

octave:2> size(b)
ans =

   1   6
```

この Octave の例で示すように MATLAB のリストの成分は全て複素数に変換されている一方でリストの構造は入力を反映せずに 6 成分の平リストに変換されています.
また MATLAB 系の言語で多次元配列は添字を用いた定義になります ([17] 参照).

3.1.3 帰納的な配列の構成方法

Yorick の配列は帰納的な方法で構成されます:

1. 0 次元の配列は数や文字列の単体で与えられる
2. n 次元の大きさ $m_1 \times m_2 \times \cdots \times m_n$ の配列 $\{A\}_{m_1,\ldots,m_n}$ は $n-1$ 次元配列で大きさが $m_1 \times \cdots \times m_{n-1}$ の配列 $\{B_k\}_{m_1,\ldots,m_{n-1}}$, $(1 \leq k \leq m_n)$ を成分とする $[\{B_1\}_{m_1,\ldots,m_{n-1}}, \ldots, \{B_n\}_{m_1,\ldots,m_{n-1}}]$ で与えられます.
   ここで配列 B_k, $(1 \leq k \leq m_n)$ を
   $$\{B_k(i_1, \ldots, i_{n-1})\}_{m_1,\ldots,m_{n-1}} \stackrel{def}{=} \{A(i_1, \ldots, i_{n-1}, k)\}_{m_1,\ldots,m_{n-1}}$$
   で定義しておきます.

この構成方法を具体例で観察しましょう:

```
> a=[1,2]
> dimsof(a)
 [1,2]
> b=[a,a,a,a]
> print,b,dimsof(b)
```

```
   [[1,2],[1,2],[1,2],[1,2]]        [2,2,4]
> c=[b,b,b,b,b,b]
> print,c,dimsof(c)
          [[[1,2],[1,2],[1,2],[1,2]],[[1,2],[1,2],[1,2],[1,2]],[[1,2],[1,2],[1,2],[1,2]],
          [[1,2],[1,2],[1,2],[1,2]],[[1,2],[1,2],[1,2],[1,2]],[[1,2],[1,2],[1,2],[1,2]]]
 [3,2,4,6]
```

この例では 1 次元配列 a, すなわち '[1,2]' から開始し, 2 次元 2×4 の配列を '[a,a,a,a]' で, 3 次元 $2 \times 4 \times 6$ の配列を '[b,b,b,b,b,b]' から構成しています. この例からも分るように添字の右端の次数が最上部の記号 "[ ]" の成分数に対応しています.

3.1.4 配列の与件型について

Yorick の配列の与件型は成分の型の優位度によって自動的に決定されます.
次の簡単な例で確認しておきましょう:

```
> a=[1,2,3,4];b=[1,2.];c=[1,2.f];d=[1,2.,3+4i];
> typeof(a)
"long"
> typeof(b)
"double"
> typeof(c)
"float"
> typeof(d)
"complex"
```

異なる型をそのまま包含させるためには構造体や LISP 風のリストを利用します. 構造体については §2.7, LISP 風のリストについては §2.6 を参照して下さい.
Yorick の配列の構成では配列の成分の型の自動変換だけではなく, 配列の成分の大きさについてもある程度の自動変換を行います:

```
>    [[1,2,[3]],[2.,3,[4]],[[3+3  i ,3],5]]
[[[1+0i],[2+0i],[3+0i ]],[[2+0 i],[3+0i],[4+0i ]],[[3+3 i],[3+0i ],
[5+0i ]]]
```

この例の入力式の配列の成分の大きさと型はまちまちですが, 出力側で配列の各成分の記号 "[ ]" による深さ, すなわち次元と与件の型を最高位にある成分に合せています. これは数値や文字列自体が 0 次元の配列だからできることです. さらに Yorick は次元の補完だけではなく, 次に示す成分の繰り返しによる補完も行えます:

```
>    [[1,2,[3,4]],[2.,3,[4,5]]]
     [[[1,1],[2,2],[3,4]],[[2,2],[3,3],[4,5]]]
```

この例では配列の成分として "[1,2,[3,4]]" と "[2.,3.,[4,5]]" を成分に持つ配列を入力し体ます．ところで配列が「均質的」であるとは型だけではなく, 成分の大きさについても同様です．そのために型に関しては各成分の数値は全て浮動小数点数に変換され, 大きさについては "[1,2,[3,4]]" では成分 '1' と成分 '2' が成分 '[3,4]' と均質的ではないために "[1,1]" と "[2,2]" に自動的に拡張されます．このような柔軟さが Yorick の非常に大きな特徴なのです．そして, これと似た処理が配列の拡大や配列の二項演算で見られます．

3.2 配列を生成する函数

3.2.1 ベクトルを生成する函数

ベクトルを生成する函数に indgen 函数, span 函数と spanl 函数の 3 種類の函数があります:

ベクトルを生成する函数

構文 (indgen, span, spanl)
indgen(⟨ 整数 ⟩)
indgen(⟨ 整数$_1$⟩ : ⟨ 整数$_2$⟩)
indgen(⟨ 整数$_1$⟩ : ⟨ 整数$_2$⟩ : ⟨ 整数$_3$⟩)
span(⟨ 実数$_1$⟩, ⟨ 実数$_2$⟩, ⟨ 整数 ⟩)
span(⟨ 実数$_1$⟩, ⟨ 実数$_2$⟩, ⟨ 整数$_1$⟩, ⟨ 整数$_2$⟩)
spanl(⟨ 実数$_1$⟩, ⟨ 実数$_2$⟩, ⟨ 整数$_3$⟩)
spanl(⟨ 実数$_1$⟩, ⟨ 実数$_2$⟩, ⟨ 整数$_1$⟩, ⟨ 整数$_2$⟩)

indgen 函数: ベクトルや配列の添字の生成で重要な函数で, long 型の整数ベクトルを生成します．引数が 1 つであれば 1 から指定した整数までのベクトルを生成し, 引数が 2 つの場合と引数が 3 つの場合が MATLAB の [⟨ 整数$_1$⟩ : ⟨ 整数$_2$⟩] と [⟨ 整数$_1$⟩ : ⟨ 整数$_2$⟩ : ⟨ 整数$_3$⟩] にそれぞれ対応します．つまり ⟨ 整数$_1$⟩ で開始し ⟨ 整数$_2$⟩ で終了するベクトルで ⟨ 整数$_3$⟩ で刻幅が指定されます．ここで示した MATLAB 流儀の表記は Yorick では「range 型」であって「配列」ではありません．

span 函数: 第 1 引数で指定した数値から開始して第 2 引数で指定した数値で終える double 型の浮動小数点数のベクトルを生成する函数です．
たとえば 'span(0,1,11)' で 0 から開始し 1 で終了する開区間 $[0, 1]$ を 10 等分する点列のベクトル '[0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]' を返却します．ここで第二の構文

は少し厄介です. この構文は $\langle$ 整数$_2\rangle$ が 1 の場合は第一の構文と同じ意味となります. $\langle$ 整数$_2\rangle$ が 2 以上であれば各成分は $\langle$ 整数$_2\rangle - 1$ 個のベクトルの入れ子になります:

```
> span(0,1,11,1)
   [0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]
> span(0,1,11,2)
        [[0],[0.1],[0.2],[0.3],[0.4],[0.5],[0.6],[0.7],[0.8],[0.9],[1]]
> span(0,1,11,3)
          [[[0]],[[0.1]],[[0.2]],[[0.3]],[[0.4]],[[0.5]],[[0.6]],[[0.7]],
   [[0.8]],[[0.9]],[[1]]]
> span(0,1,11,5)
              [[[[[0]]]],[[[[0.1]]]],[[[[0.2]]]],[[[[0.3]]]],[[[[0.4]]]],[[[[0.5]]]],
           [[[[0.6]]]],[[[[0.7]]]],[[[[0.8]]]],[[[[0.9]]]],[[[[1]]]]]
```

spanl 函数: span 函数では等間隔のベクトルを生成しましたが, spanl 函数は対数刻で等間隔となるベクトルを返す函数です. その他は span 函数と同様です. 簡単な例として e^{10} から 1 までを対数刻 spanl で 10 等分してみましょう:

```
> spanl(exp(10),1,11)
 [22026.5,8103.08,2980.96,1096.63,403.429,148.413,54.5982,20.0855,
7.38906,2.71828,1]
> log(spanl(exp(10),1,11))
  [10,9,8,7,6,5,4,3,2,1,0]
```

この例で示すように log 函数を作用させると等間隔になっています. なお, ここで生成するベクトルは対数刻となるために $\langle$ 実数$_1\rangle$ と $\langle$ 実数$_2\rangle$ の値は 0 より大でなければなりません.

3.2.2 一般の配列

一般の配列の生成は括弧を入れ子にして定義する方法もありますが, それよりも効率的に array 函数を用いて生成できます:

配列を生成する array 函数

構文 (array)
array($\langle$ 値 $\rangle$,$\langle$ 整数列 $\rangle$)
array($\langle$ 型 $\rangle$,$\langle$ 整数列 $\rangle$)

array 函数: 配列の型や値, 配列の大きさを定めて配列の生成を行う函数です. 配列の型には整数の char 型, short 型, int 型と long 型, 実数の float 型と double 型, 複

素数の complex 型と文字列の string 型, さらには利用者が定義した構造体が指定できます. 配列の型や値を第 1 引数に指定し, 配列の大きさは第 2 引数で整数の列, すなわち整数をコンマ “,” で区切った $n_1, n_2, \ldots, n_k$ で指示します. このとき n_i, $(1 \le i \le k)$ が i 次の添字の大きさを定めます. この整数列 n_i, $(1 \le i \le k)$ による配列の構成は $(\ldots((n_1), n_2), \ldots, n_k)$ と帰納的に行われます. つまり ‘array(1,$n_1, n_2, \ldots, n_k$)’ が生成する配列と ‘array(…(array(array(1,n_1),n_2),…),n_k)’ が生成する配列は一致します:

```
> array(1,3)
 [1,1,1]
> array(1,3,4)
    [[1,1,1],[1,1,1],[1,1,1],[1,1,1]]
> array(1,3,4,2)
         [[[1,1,1],[1,1,1],[1,1,1],[1,1,1]],[[1,1,1],[1,1,1],[1,1,1],[1,1,1]]]
> array(array(1,3),4)
    [[1,1,1],[1,1,1],[1,1,1],[1,1,1]]
> array(array(array (1,3),4),2)
         [[[1,1,1],[1,1,1],[1,1,1],[1,1,1]],[[1,1,1],[1,1,1],[1,1,1],[1,1,1]]]
```

3.3 添字による処理

3.3.1 添字を使った処理の概要

MATLAB 系の言語による添字を用いた処理は行列の領域を指定することで部分行列を取出するものです. ところが Yorick の添字を使った処理では配列の成分の取出から, 配列の次元の拡大や統計量の計算といった様々な配列の生成が行えます. この柔軟性は他の言語ではあまり見られるものではなく, Python の配列で「**ゴム添字 (可変次元添字)**」がある程度です. Yorick にはゴム添字と呼ぶ添字は添字 “..” と添字 “*” の二種類があり, 前者の添字 “..” は添字の次元の引き延ばしに関連し, 後者の添字 “*” は次元の縮めに関連します[1].

3.3.2 疑似添字 “-” と可変次元添字 “..”

記号 “[ ]” を使って直接, 配列や成分を括りましたが, この処理は「**疑似添字 “-”**」を使うことで容易に実行できます:

```
> 1(-,)
[1]
```

[1]次元を延したり縮めたりするのでゴム添字.

```
> 1(−,)(,−)
 [[1]]
> 1(−,)(−,)(−,)
 [[[1]]]
> 1(−,)(−,)(−,)(−,)
 [[[[1]]]]
> 1(−,)(−,)(−,)(−,)(−,)
 [[[[[1]]]]]
```

この処理を 2 成分以上の 1 次元配列に行えばどうなるでしょうか？

```
> [1,2,3,4](−,)
 [[1],[2],[3],[4]]
> [1,2,3,4](−,)(−,)
 [[[1]],[[2]],[[3]],[[4]]]
> [1,2,3,4](−,)(−,)(−,)
 [[[[1]]],[[[2]]],[[[3]]],[[[4]]]]
> [1,2,3,4](−,)(−,)(−,)(−,)
 [[[[[1]]]],[[[[2]]]],[[[[3]]]],[[[[4]]]]]
```

このように添字 “(-,)” は配列の各成分を記号 “[ ]” で括る処理と同じ結果になります. すなわち添字 “(-,)” は配列の次元を拡大し, dimsof 函数の返すベクトルの左側に ‘1’ を追加する作用になります. では添字 “-” を入れ替えた添字 “(,-)” はどのような作用になるでしょうか？

```
> 1(,−)
[1]
> 1(,−)(,−)
 [[1]]
> 1(,−)(,−)(,−)
 [[1]]
> 1(,−)(,−)(,−)(,−)
 [[1]]
```

添字 “(,-)” とは様子が異なりますね. これは添字 “(,-)” による配列拡大に追い付けなくなったためで, 次元に合せて添字 “,” を追加する必要があるのです:

```
> 1(,−)
[1]
> 1(,−)(,,−)
 [[1]]
> 1(,−)(,,−)(,,,−)
 [[[1]]]
> 1(,−)(,,−)(,,,−)(,,,,−)
 [[[[1]]]]
> 1(,−)(,,−)(,,,−)(,,,,−)(,,,,,−)
 [[[[[1]]]]]
```

実は添字 “(-,)” も同様なのです:

```
> [1,2,3](−,)(−,,)(−,,,)
   [[[[1]]],[[[2]]],[[[3]]]]
> [1,2,3](,−)(,,−)(,,,−)
 [[[[1,2,3]]]]
```

dimsof 函数が返す配列の次元と大きさのベクトルで添字 “-” の箇所に ‘1’ が置かれていますね. ここで添字 “(-,)” の場合は大きさの数列の左端に追加されために後続の数列を考慮する必要がありませんが, 添字 “(,-)” は配列を 1 次元として回りを添字 “[ ]” で括る作用となってしまいます. そこで配列を 1 次元と見做さないために配列の次元分の “,” を添字 “(-)” の左側に入れなければならないのです.

ここで見たように適切な添字 “,” を入れることは間違え易い処理です. そのために Yorick では「**ゴム添字**」と呼ばれる「**可変次元添字 “..”**」を使って次数の補完が行えるのです:

```
> 1(..,−)
[1]
> 1(..,−)(..,−)
[[1]]
> 1(..,−)(..,−)(..,−)
[[[1]]]
> 1(..,−)(..,−)(..,−)(..,−)
[[[[1]]]]
> 1(..,−)(..,−)(..,−)(..,−)(..,−)
 [[[[[1]]]]]
```

この可変次元添字は Python にはありますが MATLAB 系の言語にはありません.

この疑似添字の面白い点は容易に反復処理が行える点です:

```
> a =[1,2,3,4,5,6,7,8,9,10]
> a(,−)
 [[1,2,3,4,5,6,7,8,9,10]]
> a(,−:1:3)
   [[1,2,3,4,5,6,7,8,9,10],[1,2,3,4,5,6,7,8,9,10],
 [1,2,3,4,5,6,7,8,9,10]]
> a(−,)
    [[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]]
> a(−:1:3,)
     [[1,1,1],[2,2,2],[3,3,3],[4,4,4],[5,5,5],[6,6,6],[7,7,7],[8,8,8],
 [9,9,9],[10,10,10]]
```

ここで式 ‘a(,-:1:3)’ によって配列 a が大きさ 10 の 1 次元配列から大きさ 10×3 の 2 次元配列に拡大され, 本来の配列 a が ‘a(:,1)’, ‘a(:,2)’ と ‘a(:,3)’ に複製されます. 同

様に ‘a(-:1:3,)’ によって今度は 3×10 の 2 次元配列に拡大されて ‘a(1,:)’ が ‘a(2,:)’ と ‘a(3,:)’ に複製されます．このように添字 “-” を置いた側に配列が拡大され, 添字 “-” の直後に記号 “:” に続けて “m:n” の range 型の与件を置くことで列の長さほど配列が拡大されます．また添字 “-” を range 型の対象 ‘-:1:1’ の省略形と考えても良いでしょう．ここで添字 “-:” に続ける列表記は 1 から開始する必要はありません:

```
> a(,−:1:3)
      [[1,2,3,4,5,6,7,8,9,10],[1,2,3,4,5,6,7,8,9,10],[1,2,3,4,5,6,7,8,9,10]]
> a(,−:−100:−98)
      [[1,2,3,4,5,6,7,8,9,10],[1,2,3,4,5,6,7,8,9,10],[1,2,3,4,5,6,7,8,9,10]]
```

この例で示すように領域は “1:3” でも “-100:98” でも構いません．要するに指定した個数の列ができれば良いのです．また添字 “-:” は “m:n” の表記, すなわち range 型以外の与件を受け付けません．たとえば ‘a(,-:[1,2,3])’ といった表記はエラーになります．

3.3.3 平坦化添字 “*”

添字 “[ ]” と逆の操作を行うものが添字 “*” です．特性を考慮すると「**平坦化添字**」というべき添字で, この添字の動作を例で確認しておきましょう:

```
> a=[1,2,3,4,5]
> a(−,)
   [[1],[2],[3],[4],[5]]
> (a(−,))(*)
 [1,2,3,4,5]
> a(,−)
  [[1,2,3,4,5]]
> a(,−)(*)
 [1,2,3,4,5]
```

このように疑似添字の逆操作を行う添字となっていることが判ります．
この平坦化添字と後述の空添字や添字 “:” を併用することで, 平坦化する程度を調整することが可能です:

```
> a=[1,2,3,4,5]
> b1=a(−:1:5,)
> b1
      [[1,1,1,1,1],[2,2,2,2,2],[3,3,3,3,3],[4,4,4,4,4],[5,5,5,5,5]]
> b2=b1(−,)(−,)
> b2
            [[[[1]],[[1]],[[1]],[[1]],[[1]]],[[[2]],[[2]],[[2]],[[2]],[[2]]],
            [[[3]],[[3]],[[3]],[[3]],[[3]]],[[[4]],[[4]],[[4]],[[4]],[[4]]],
```

```
      [[[5]],[[5]],[[5]],[[5]],[[5]]]]
> b2(,,*)
            [[[1]],[[1]],[[1]],[[1]],[[1]],[[2]],[[2]],[[2]],[[2]],[[2]],
            [[3]],[[3]],[[3]],[[3]],[[3]],[[4]],[[4]],[[4]],[[4]],[[4]],
      [[5]],[[5]],[[5]],[[5]],[[5]]]
> b2(,*)
          [[1],[1],[1],[1],[1],[2],[2],[2],[2],[2],[3],[3],[3],[3],[3],
      [4],[4],[4],[4],[4],[5],[5],[5],[5],[5]]
> b2(*)
    [1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,4,4,4,4,4,5,5,5,5,5]
> b2(*,)
      [[1,1,1,1,1],[2,2,2,2,2],[3,3,3,3,3],[4,4,4,4,4],[5,5,5,5,5]]
```

この例では配列 a を '[1,2,3,4,5]' とします. まず 'a(-:1:5,)' で '[1,2,3,4,5]' の各成分を 5 個複製した配列を生成し, 添字 "(-,)(-,)" を作用させることで次元を増やして 4 次の配列にしています. それから添字 "(,,*)" で次元を 1 つ減らし, 今度は添字 "(,*)" で次元を 2 つ減らして添字 "(*)" で配列を平坦化して 1 次元配列にしています. 一般的に全体の次元を N, 添字の区切のコンマ "," の総数を n とするときに潰される次元は $N-n-1$ で与えられます.

3.3.4 空添字と添字 ":"

「空白文字」" ", "nil" や "[ ]" を添字として利用するとき, これらの添字を「空添字」と呼びます. 空添字は MATLAB での全体を表現する添字 ":" に相当します:

```
> a(,)
  [[[1,2,3,4,5,6]]]
> a (,,,)
  [[[1,2,3,4,5,6]]]
> a(nil,)
  [[[1,2,3,4,5,6]]]
```

ちなみに Yorick でも記号 ":" が使えますが, その意味は空添字とは微妙に異なります:

```
> a()
  [[[1,2,3,4,5,6]]]
> a (:,)
  [[[1,2,3,4,5,6]]]
> a (:,1)
 [1,2,3,4,5,6]
> a(,1)
 [1,2,3,4,5,6]
> a(:)
 [1,2,3,4,5,6]
```

空添字と添字 “:” の使い方と意味はほぼ同じものですが ‘a(:)’ のように単体で用いればば添字 “(:,1)” の意味になります.

3.4　配列の成分の取出

3.4.1　添字による成分抽出の考え方

ここでは 1 次元配列, 2 次元配列と順を追って考えてみましょう. まず配列 A を次元配列 ‘$[a_1, \ldots, a_n]$’ とします. ここで x を添字とするときに ‘$A(x)$’ は 配列 A から x に適合する成分を取出す操作になります.
次に配列 A を大きさが $n_1 \times n_2$ の 2 次元配列とします. このとき ‘$A(x,)$’ が n_2 成分の 1 次元配列 ‘$[A(x,1), \ldots, A(x,n_2)]$’, ‘$A(,x)$’ が n_2 成分の 1 次元配列 ‘$[A(1,x), \ldots, A(n_1,x)]$’ になります.
これらのことから予想されるように一般の m 次元配列に対し, k 次に添字 x を置いた配列 ‘$A(,\ldots,\overset{k}{\check{x}},\ldots,)$’ は $n-1$ 次元配列 ‘$\{A(i_1,\ldots,i_{k-1},\overset{k}{\check{x}},i_{k+1},\ldots,i_n)\}$’ になります. したがって添字 x の作用は x が配置された k 次の添字部分で生じます.

3.4.2　範囲指定の添字 “:”

MATLAB の配列操作で表的なものの一つが ‘3:10:2’ のような添字 “:” を用いる手法です. この添字 “:” の利用は Excel で列や行の総和を指定するときに, たとえば, D 列 1 行から D 列 5 行までの成分の総和を計算するときに使う計算式 `=sum(D1:D5)` に現われる記号 “:” と同じ意味です. Yorick では ‘1:4’ のような表記は range 型の対象になりますが, range 型の対象は単体で存在できる対象ではなく, 配列の添字や函数の引数として利用できる対象です:

```
> L=span(0,1,101);
> L(2:6)
 [0.01,0.02,0.03,0.04,0.05]
> L(2:6:1)
 [0.01,0.02,0.03,0.04,0.05]
> L(2:6:2)
 [0.01,0.03,0.05]
> L(6:2:−2)
 [0.05,0.03,0.01]
```

この例では最初に span 函数を使って閉区間 $[0,1]$ を 101 個の点で等間隔に区分した数列の配列を生成します. ‘L(2:6)’ は MATLAB でお馴染の表記で配列 L の第 2 成分

から第 6 成分までを取出した配列を返す操作です．これは Yorick の ‘L(2:6:1)’ と同じ意味で，〈 始点 〉:〈 終点 〉:〈 刻幅 〉 となっています．ところが MATLAB の場合は 〈 始点 〉:〈 刻幅 〉:〈 終点 〉 となっていることに注意が必要です．この刻幅は整数が指定可能ですが，〈 始点 〉+ 正整数 × 〈 刻幅 〉=〈 終点 〉となる正整数が存在しなければ意味がありません．

3.4.3 添字 “::-1”

指定した次数の添字に対して順序を逆にする操作になります:

```
> A   =[[1,2,3,4],[4,5,6,7],[8,9,8,7]]
> A(1,)
 [1,4,8]
> A(1,::−1)
 [8,4,1]
> A(,1)
 [1,2,3,4]
> A(::−1,1)
 [4,3,2,1]
> A(,::−1)
    [[8,9,8,7],[4,5,6,7],[1,2,3,4]]
> A(::−1,)
    [[4,3,2,1],[7,6,5,4],[7,8,9,8]]
```

MATLAB で類似の操作を行う添字は ‘成分数:1:-1’ になりますが，ここで “成分数” 自体は対象によって一意に決まる値ですね．Yorick にとって自明なこれらの値を省略した添字が “::-1” なのです．

3.4.4 添字による統計量の抽出

Yorick では配列に対して次の記号を添字として与えることで，その添字に対応する統計量が取り出せます．

添字による統計量の取出

添字	例	概要
min	a(min)	最小値を取得
max	a(max)	最大値を取得
sum	a(sum)	総和量を取得
avg	a(avg)	平均値を取得
ptp	a(ptp)	最大値と最小値の差を取得
rms	a(rms)	統計量 RMS(標準偏差) を取得
mnx	a(mnx)	最小値を返す位置を取得
mxx	a(mxx)	最大値を返す位置を取得

これらの添字によって返却される値は, 添字 avg が配列が複素数でなければ double 型, 添字 mnx と添字 mxx が long 型になることを除いて配列の型と一致します. これらの添字は 1 次元配列の場合には非常に明瞭ですが, 多次元配列に対しては Yorick の配列操作方法も関係して注意が必要になります.

つまり n 次元配列 $A_{i_1=1,\ldots,i_n=1}^{N_1,\ldots,N_n}$ に対して $A(,\ldots,\overset{k}{\check{x}},\ldots,)$ によって $n-1$ 次元配列を成分とする $[A(,\ldots,\overset{k}{\check{1}},\ldots,),\ldots,A(,\ldots,\overset{k}{\check{N}_k},\ldots,)]$ の中から添字 x の意味に対応する値が返却されます. たとえば配列を '[[1,2,3],[4,5,6]]' とするときに添字 avg は 'a(avg,)' や 'a(,avg)' の二箇所に置けますが, これらの結果は異なります:

```
> a =[[1,2,3],[4,5,6]]
> a(avg,)
[2,5]
> a(,avg)
 [2.5,3.5,4.5]
```

これからも分るように 'a(avg,)' は '[[a(1,1),a(2,1),a(3,1)](avg),[a(1,2),a(2,2),a(3,2)](avg)]' を計算し, 'a(,avg)' は '[[a(1,1),a(1,2)](avg),[a(2,1),a(2,2)](avg),[a(3,1),a(3,2)](avg)]' を計算しています. Yorick では添字として記号を置いた個所の添字を動かした配列を生成し, その生成した配列に対して処理が行われます. そのために添字に記号を置いた処理では正確に次元の指定を行わなければなりません. たとえば 'a(avg)' は 2 となります. 配列 a を 1 次元の配列と看做したのであれば, ここでの値は 3.5 にならなけばなりませんが, 'a(avg)' は 'a(avg,1)' と同じ意味になります. このように, 添字の右側に置くべき記号 "," を省略したときに該当する箇所の添字は 1 に固定されて処理されることに注意が必要です.

また添字 "min", "max", "sum", "avg" については同様の操作を行う函数が存在し, そ

れらの函数を用いた方が高速な処理が行えます. そして多次元配列に対してこれらの添字を配列の全ての添字に用いると該当する函数と同じ結果が得られます:

```
> a =[[1,2,3],[4,5,6]]
> a(avg,avg)
3.5
> avg(a)
3.5
> a(min,min)
1
> min(a)
1
```

この場合でも添字を用いた場合には処理速度が幾らか低下します.

添字 min: 配列で添字付けた各 1 次元配列の最小値を返す添字で, 返却型は配列の型と同じです:

```
> x=span(0,2*pi,101);
> y=log(x+1);
> y(min)
0
```

添字 max: 配列の最大値を返す添字で返却型は配列の型と同じです:

```
> x=span(0,2*pi,101);
> y=log(x+1);
> y(max)
1.98557
```

添字 sum: 配列の総和を返す添字で返却型は配列の型と同じです:

```
> x=span(0,2*pi,101);
> y=exp(x)*sin(x);
> y(sum)
-4250.55
>  [1,2,3]( sum)
6
> typeof ([1,2,3]( sum))
"long"
```

このように配列を括弧で括って添字を指定することもできます.

なお画像のような大きな整数配列を扱うときに, その総和が Yorick で扱える整数の範囲に収まるかどうか注意を払う必要があります. もし確実に越える可能性があれば, 配列を浮動小数点数に変換しておくといった工夫が必要になります.

添字 avg: 配列の平均値を返す添字です:

```
> a=indgen(1:10:2);
> a(avg)
5
> typeof(a(avg))
"double"
```

このように整数 long 型の配列でも, その平均値は double 型であることに注意して下さい.

添字 ptp: 配列の最大値と最小値の差を返す添字で返却型は配列の型と一致します:

```
> (sin(span(0,pi/2,101)))(ptp)
1
```

なお添字 'ptp' は複素数型の配列に対しては大小関係がそのまま複素数に入らないために使えません.

添字 rms: 複素型の配列を除き, 標準偏差 $\sqrt{(\sum_{i=1}^{N}(x_i-\mu)^2)/N}$ を計算します. したがって返却値は必ず double 型の実数となります. ここで RMS(Root Mean Square) と標準偏差は平均値が 0 の場合を除いて異なる統計量です:

```
> a=sin(span(0,pi*2/3,101))
> a(rms)
0.303653
> A=a-a(avg);
> sqrt((A*A)(sum)/(dimsof(A)(2)))
0.303653
```

添字 mnx: 配列の成分で最小値を返す成分の位置を返却する添字です:

```
> (sin(span(0,pi*2/3,101)))(mnx)
1
> typeof((sin(span(0,pi*2/3,101)))(mnx))
"long"
```

この添字は複素型の配列に対しては使えません.

添字 mxx: 配列の成分で最大値を返す成分の位置を返却する添字です:

```
> (sin(span(0,pi*2/3,101)))(mxx)
76
> typeof((sin(span(0,pi*2/3,101)))(mxx))
"long"
> (sin(span(0,pi*2/3,101)))((sin(span(0,pi*2/3,101)))(mxx))
1
```

この添字は複素型の配列に対しては使えません.

3.4.5 添字を使った配列生成

添字に次の記号を与えることで新しい配列を生成することもできます.

与えられた配列から新しい配列を生成

添字	例	概要
cum	a(cum)	0 から開始する部分和の配列を返却
psum	a(psum)	第 1 成分から開始する部分和の配列を返却
dif	a(dif)	隣合う成分同士の差分配列を返却
zcen	a(zcen)	両端を含まない, 隣合う成分の中点の配列を返却.
pcen	a(pcen)	両端を含む, 隣合う成分の中点の配列を返却
uncp	a(uncp)	pcen の逆操作

これらの添字の利用の最良の例は Y_SITE/i にある "demo2.i" ファイルの中で定義されている laplacian 函数です. Laplacian が非常に簡潔に記述されていることが判るでしょう.

添字 cum: n 成分の配列 '$[a_1, \ldots, a_n]$' に対して '$[a_1, \ldots, a_n]$(cum)' は $n+1$ 成分の配列 '$[b_1, \ldots, b_{n+1}]$' を返します. ここで $b_i = \sum_{k=1}^{i-1} a_k$ とします.

添字 psum: n 成分の配列 '$[a_1, \ldots, a_n]$] に対して '$[a_1, \ldots, a_n]$(psum)' は n 成分の配列 '$[c_1, \ldots, c_n]$' を返します. ここで $c_i = \sum_{k=1}^{i} a_k$ とします.

添字 dif: 配列の隣合う成分の差分を返す添字です. すなわち, n 成分の配列 '$[a_1, \ldots, a_n]$' に対して '$[a_1, \ldots, a_n]$(dif)' は $n-1$ 成分の配列 '$[a_2-a_1, \ldots, a_i-a_{i-1}, \ldots, a_n-a_{n-1}]$' を返却します.

添字 zcen:　n 成分の配列 '$[a_1,\ldots,a_n]$' に対して '$[a_1,\ldots,a_n]$(zcen)' は $n-1$ 成分の配列 '$[(a_2-a_1)/2,\ldots,(a_i-a_{i-1})/2,\ldots,(a_n-a_{n-1})/2]$' を返却します.

添字 pcen:　n 成分の配列 '$[a_1,\ldots,a_n]$' に対して '$[a_1,\ldots,a_n]$(pcen)' は $n+1$ 成分の配列 '$[a_1,(a_2-a_1)/2,\ldots,(a_i-a_{i-1})/2,\ldots,(a_n-a_{n-1})/2,a_n]$' を返却します.

添字 uncp:　pcen の逆操作を行う添字です. 配列 a に対して $a=(a(\mathrm{pcen}))(\mathrm{uncp})$ を満します:

```
> a
 [1,3,4,5,8,9,0,−20]
> a(pcen)
 [1,2,3.5,4.5,6.5,8.5,4.5,−10,−20]
> (a(pcen))(uncp)
 [1,3,4,5,8,9,0,−20]
```

ただし '$(a(\mathrm{uncp}))(\mathrm{pcen})$' は末端の成分の脱落が生じるために配列 a と一致しません.

3.5　配列に関連する函数

3.5.1　配列の情報を返す函数

Yorick には配列の大きさといった情報を返す函数があります. ここでは代表的な函数について概要を述べておきましょう:

対象の配列としての大きさを返す函数

構文 (dimsof, numberof, orgsof, use_origins)
dimsof(⟨ 対象 ⟩)
dimsof(⟨ 対象$_1$⟩ ,. . ., ⟨ 対象$_n$⟩)
numberof(⟨ 対象 ⟩)
orgsof(⟨ 対象 ⟩)
use_origins

dimsof 函数:　配列の大きさをベクトルで返す函数で, 返却ベクトルの第 1 成分が配列の次元, 第 2 成分以降が配列の具体的な大きさを示す整数列になります. 複数の対象を引数とする場合は Yorick 独自の機能が関係したもので, 引数に含まれる対象の最高次数の配列が他の対象で添字 "-" を利用することで同じ大きさに膨らますことができる場合に限って, その対象の大きさを返却し, それ以外は空配列 "[ ]" を返却します:

```
> a1=array(1,10)
> dimsof(a1)
[1,10]
> dimsof(a1(-,))
[2,1,10]
> dimsof(a1(-,,))
[2,1,10]
> dimsof(a1(-,-,))
[3,1,1,10]
> dimsof(a1(-,-,-,))
[4,1,1,1,10]
> dimsof(a1(-,-,-,),a1)
[4,1,1,1,10]
```

numberof 函数: 配列の成分の総数を返却します. dimsof 函数が返却するベクトルの各添字の大きさの積が対応します:

```
> numberof ([1,2,3,4,5,6]);
6
> numberof  ([[1,2],[3,4],[5,6]]);
6
```

orgsof 函数: 引数の添字の始点をベクトルで返却する函数です. dimsof 函数の場合は配列の次元と各次元での配列の長さをベクトルで返却しますが, orgsof 函数の場合も先頭が配列の次元, 各次元の配列の始点が返却されます. 配列の始点は通常は 1 ですが, use_origins 函数を用いることで変更することが可能です.

use_originsf 函数: 引数の添字の始点を指定する函数です. この函数を併用することで C のように配列を 0 から開始するように変更することができます.

3.5.2 配列の拡大

配列拡大の基本的な考え方

m 次元配列 $\{A\}_{M_1,\ldots,M_m}$ に k 個の n_s 次元の配列 $\{P_s\}_{{}_sN_1,\ldots,{}_sN_{n_s}}$, $(1 \leq s \leq k)$ を追加するものとして話を進めます.
配列拡大の考え方は, 1 次元配列 '$[a_1,\ldots,a_m]$' に対して 1 次元配列 '$[b_1,\ldots,b_n]$' を追加した新しい配列 '$[a_1,\ldots,a_m,b_1,\ldots,b_n]$' を生成することが根底にあります. そのために

引数の順序によって得られる配列が異なります．またこの処理を行う上での配列の均質性から第 1 引数の配列 $\{A\}_{M_1,\ldots,M_m}$ とその他の引数の配列 $\{P_s\}_{{}_sN_1,\ldots,{}_sN_s}$, $(1 \leq s \leq k)$ から得られる配列:

$$\{A_t\}_{M_1,\ldots,M_{m-1}} \stackrel{def}{=} \{A(i_1,\ldots,i_{M_{m-1}},t)\}_{M_1,\ldots,M_{m-1}},\ (1 \leq t \leq M_n)$$
$$\{{}_wP_s\}_{{}_sN_1,\ldots,{}_sN_{n_s-1}} \stackrel{def}{=} \{P_s(i_1,\ldots,i_{{}_sN_{n_s-1}},w)\}_{{}_sN_1,\ldots,{}_sN_{n_s-1}},\ (1 \leq w \leq\ {}_sN_{n_s})$$

は配列としての次元と次数が最終的に一致していなければなりません．これらの配列 $\{A_t\}$ と配列 $\{{}_wP_s\}$ から次で定められる新しい配列 $\{B\}$ を生成する操作が配列の拡大になります:

$$\{B\} = [\{A_1\},\ldots,\{A_m\},\{{}_1P_1\},\ldots,\{{}_{N_{n_1}}P_1\},\ldots,\{{}_1P_k\},\ldots,\{{}_{N_{n_k}}P_k\}]$$

配列の拡大を行う函数

ここで配列を拡大する函数を纏めておきましょう:

配列を拡大する函数

構文 (grow, _)
grow(⟨ 配列 ⟩, ⟨ 追加分$_1$⟩, . . . , ⟨ 追加分$_n$⟩)
grow, ⟨ 変数 ⟩, ⟨ 追加分$_1$⟩, . . . , ⟨ 追加分$_n$⟩)
_(⟨ 配列 ⟩, ⟨ 追加分$_1$⟩, . . . , ⟨ 追加分$_n$⟩)
_ , ⟨ 変数 ⟩, ⟨ 追加分$_1$⟩, . . . , ⟨ 追加分$_n$⟩

これらの函数で引数を記号 “( )” を用いない表記を利用する場合, 第 1 引数でポインタを用いた値の代入が行われるために第 1 引数は配列が束縛された変数でなければなりません．記号 “( )” を用いる表記ではポインタを用いずに配列が返却されるので, 第 1 引数は配列そのものであっても問題はありません．この特徴を除くと函数 grow と函数_ との違いはありません．

grow 函数:　grow 函数は第 1 引数の配列に第 2 引数以降の配列を追加する函数です．ここで引数を括る括弧 “( )” を省略した場合, 第 1 引数は配列が束縛された変数で, 処理結果はこの第 1 引数の変数に代入されます:

```
> a=[1,2,3,4];
> grow,a,[5,6,7]
> a
 [1,2,3,4,5,6,7]
```

```
> grow,a ,8,9,[10,11]
> a
  [1,2,3,4,5,6,7,8,9,10,11]
```

函数 "_": 使い方は grow 函数と全く同じです. 引数を括る括弧 "( )" を外した書式も使えますが, その場合は grow 函数と同様に第 1 引数は配列が割当てられた変数に結果が代入されます:

```
>a=[1,2,4];
> _,a ,[1,2,3]
> a
 [1,2,4,1,2,3]
```

配列の自動辻褄合せ

やや面倒なのが配列の拡大の細かな仕組です. まず Yorick は配列の多少の差異を自動的に辻褄を合せることができます. この様子を次の簡単な例で確認しておきましょう:

```
> grow ([1,2,3],4)
 [1,2,3,4]
> grow ([[1,2,3]],4)
  [[1,2,3],[1,1,1]]
> grow    ([[[[1]],[[2]],[[3]]]],4)
        [[[[1]],[[2]],[[3]]],[[[4]],[[4]],[[4]]]]
```

この例では第 2 引数の数 4 は 0 次元配列となっており, この配列を第 1 引数の配列の次元と大きさに合せて Yorick が自動調整を行っています. この調整の状況は配列同士の二項演算で行われている自動調整と似た側面を持っていますが, 引数間の順序が大きく意味を持つために全く同じ処理を行っている訳ではありません:

```
>  grow     ([[5,4,3]],[[[[[1,2,3]]]]])
  [[5,4,3],[1,2,3]]
>      ([[5,4,3]]+[[[[[1,2,3]]]]])
   [[[[[6,6,6]]]]]
> grow     ([[[[[1,2,3]]]]],[[5,4,3]])
      [[[[[1,2,3]]]]],[[[[5,4,3]]]]]
```

この例から判るように配列の次元は第 1 引数の配列 $\{A\}_{M_1,\ldots,M_m}$ を中心にして決定されます. ただし第 1 引数が 'nil', すなわち空の配列 '[ ]' の場合は第 2 引数が中心とされ, 第 1 引数が 0 次元配列 a の場合は上の定義で $m = 1$, $\{A_1\} \stackrel{def}{=} a$ として処理が遂行されます.

この処理の概要を次に纏めておきましょう:

- 引数の配列に nil と同値な配列が存在する場合
  nil に等しい引数配列を除外して配列が 2 つ以上残っていれば, それらの配列で拡大を行い, 配列が 1 つだけ残っていれば, その配列を返却します.
- $m = 0$ の場合
  $A \stackrel{def}{=} [a]$ で配列 A を再定義して, $m = 1$ の場合に帰着させます.
- $m = 1$ の場合
  全ての $s \in \{1, \ldots, k\}$ に対して $n_s = 0$ または $n_s = 1$ の場合, すなわち第 2 引数以降が 0 次元配列か 1 次元の配列である場合に限って配列の拡大を行います. このとき拡大された配列の次元は 1 次元, その大きさは $\sum_{s=1}^{k} n_s + 1$ です.
- $m > 1$ の場合
  拡大した配列の次元は第 1 引数の配列と同じ m 次元になります. 配列の大きさの自動調整は, 第 1 引数の配列の添字 "$(i_1, \ldots, i_m)$" の左側から実行されます.

3.5.3 配列の大きさを変更, 複製する函数

配列の大きさを変更する函数

構文 (reform, reshape, eq_nocopy)
reform(⟨ 配列 ⟩, ⟨ ベクトル ⟩)
reshape,⟨ 変数名 ⟩, ⟨ 番地 ⟩, ⟨ 与件型 ⟩, ⟨ ベクトル ⟩
reshape,⟨ 変数名 ⟩, ⟨ 与件型 ⟩, ⟨ ベクトル ⟩
reshape,⟨ 変数名 ⟩
eq_nocopy,⟨ 変数名 ⟩, ⟨ 対象 ⟩

reform 函数: Yorick 言語で記述された函数で, 与えられた ⟨ 配列 ⟩ を ⟨ ベクトル ⟩ で指定した大きさに変換した配列を返す函数です. 函数内部では grow 函数が用いられています.

reshape 函数: 第 1 引数として ⟨ 変数名 ⟩ を取り, この変数に対して以降の引数で指定した値がポインタを介して代入されます.
第 2 引数の ⟨ 番地 ⟩ は '0x718b48' のような 16 進数表示の long 型の数値, あるいは変数 x に対して '&x' のように記号 "&" を付けた表記となって対象が格納された番地を直接指示します. この第 2 引数に ⟨ 番地 ⟩ を指定する場合は第 1 引数に対して ⟨ 番地 ⟩

で指定された対象が ⟨ ベクトル ⟩ で指示した大きさの ⟨ 与件型 ⟩ で指定した配列に変換されて第 1 引数の ⟨ 変数名 ⟩ に割当・代入されます:

```
> y=&[1,2,3]
> y
0x718ac0
> long(0x718ac0)
7441088
> reshape,a,0x718ac0,long ,[1,3]
> a
 [1,2,3]
> reshape,b,7441088,long ,[1,3]
> b
 [1,2,3]
> reshape,b,y,long ,[1,3]
```

この例では第 2 引数として番地を指定した場合の結果を示しています. ここで番地の指定方法は 16 進数, long 型, 変数名の先頭に文字 “&” を付けたもので行えます. この例では, ベクトルを番地で指定した対象の配列と同じ大きさに指定しているために大きさの変換が実行されていません. ここでの大きさの変更は配列をテンソルとして見た場合に可能な拡大と縮約に限定されます.
第 2 引数に ⟨ 与件型 ⟩ を指定した場合, ⟨ 変数名 ⟩ に割当てられている対象に対して変換と大きさの変更が実行され, その結果が第 1 引数の変数に代入されます.
ここで引数が ⟨ 変数名 ⟩ のみの場合は ⟨ 変数名 ⟩ で指定した変数に対して ‘nil’ が代入されます.
この reshape 函数では, 引数として与える与件型を番地で指定する対象と同じ型とするか, 番地を指定しない場合には第 1 引数の変数に割当てられた対象の与件型と一致させておく必要があります.

eq_nocopy 函数: 第 2 引数で指定された対象を第 1 引数で与えた ⟨ 変数名 ⟩ で指定される変数に割当･代入する函数で, 第 2 引数の対象が配列でなければ $\boxed{\langle \text{変数名} \rangle = \langle \text{対象} \rangle}$ と同値です.

3.5.4　配列の各成分の置換を行う函数

配列の成分の置換を行う函数として sort 函数と transpose 函数があります:

配列の各成分を置換する函数

構文 (sort, transpose)
sort(⟨ 配列 ⟩)
sort(⟨ 配列 ⟩, ⟨ 位置 ⟩)
transpose(⟨ 配列 ⟩)
transpose(⟨ 配列 ⟩, ⟨ 置換$_1$⟩, . . . , ⟨ 置換$_n$⟩)

sort 函数:　配列を第 1 引数とし, 引数が 1 つだけの場合は配列の並び換えを指示する配列を返します. また第 2 引数は多次元配列の場合は並び換えを行う次元の指示を行います.

transpose 函数:　配列の置換を行う函数です. 2 次元配列の場合は行列の転置に相当する変換となります. 一般の配列 $\{A\}_{i_1,\ldots,i_{n-1}i_n}$ に対して置換を指定しない引数が配列 A のみであれば返却する配列 $\{B\}_{j_1,j_2,\ldots,j_{n-1},j_n}$ を $\{B\}_{j_1,j_2,\ldots,j_{n-1},j_n} \overset{def}{=} \{A\}_{j_n,j_2,\ldots,j_{n-1},j_1)}$ で定めます. transpose 函数の第 2 引数以降で指定する置換は整数列, あるいは整数ベクトルで指定します. なお整数列で指定される置換は一つの整数ベクトルで表現可能な置換に限定されて複数の置換の合成で表現された置換を作用させるときは各置換を整数ベクトルで表現してそれらを引数として transpose 函数に引渡します. つまり置換を $\sigma_{i=1,\ldots,m}$ とするとき ‘transpose($\{A\}_{i_1,\ldots,i_n}, \sigma_1, \ldots, \sigma_m$)’ によって配列 $\{A\}$ は $\{A\}_{i_{\sigma_1,\ldots,\sigma_m(1)},\ldots,i_{\sigma_1,\ldots,\sigma_m(n)}}$ に写されます.

3.6　成分の照合

3.6.1　比較の演算子による照合

MATLAB や Octave で「**比較の演算子**」“>=”, “>”, “<=”, “<” や “==” による結果は各成分が整数の ‘0’ と ‘1’ で置換えられます. これは Yorick でも同様です. たとえば配列 ‘[1,2,3,4,5]’ に対する ‘[1,2,3,4,5]>3’ の処理を見てみましょう:

```
> [1,2,3,4,5]>3
 [0,0,0,1,1]
> typeof([1,2,3,4,5]>3)
> "int"
```

このように関係 “>” が真となる成分が ‘1’, 偽となる成分が ‘0’ になり, その結果は int 型になります. この性質から処理結果を使って通常の積演算ができることになります. そこで簡単な例として配列 x が与えられたとき, この配列 x に対して次の処理を考えてみましょう:

- 3 の倍数の成分を 3 で割って-1 倍にする
- 5 の倍数の成分を 2 で置換する
- 7 の倍数となる成分 a に対しては 2^a で置換する

C や FORTRAN なら反復処理と if 文による分岐を利用して処理を行うのでちょっとしたプログラムになるでしょう. ところが MATLAB, Octave や Yorick では ‘(x%3==0)*(-1)+(x%5==0)*2+(x%7==0)*2^x’ と一行で処理が行えるのです:

```
> x=indgen(1:20)
> x
  [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
> (x%3==0)*(-1)+(x%5==0)*2+(x%7==0)*2^x
 [0,0,-1,0,2,-1,128,0,-1,2,0,-1,0,16384,1,0,0,-1,0,2]
```

条件 A を満す場合に a, A を満さない場合に b を計算させる場合は Yorick で次の表記が選べます:

分岐処理の表記

1.	**if**(A) a; **then** b	
2.	A **?** a **:** b	
3.	A * a + (1 - A) * b	a, b が数値の場合
4.	A * a + !A * b	a, b が数値の場合

ここでの上の 2 つが分岐処理で下の 2 つが算術化した表記です. ところで上から二番目の `A?a:b` は Yorick の算術式や函数の引数に埋込める性質を持っています. このような分岐処理は純粋な分岐処理ではなく算術として置換した方が Yorick も含めた MATLAB 系の言語で高速な処理が行えます. すなわち数値行列算術に置換えることで行列計算ライブラリが利用されることで言語のオーバーヘッドが軽減されるためです.

3.6.2 非零点の検出

成分の照合によって ‘0’ と ‘1’ のみで構成された配列が得られます. ここで Yorick には MATLAB 系の言語のように数値配列で ‘0’ と異なる値を検出する函数があり, この

ような函数で検出した添字を用いて該当する成分の値を参照することができます. ここで MATLAB 系の言語では find 函数のみで処理が行えますが, Yorick には配列を 1 次元配列と見做して検出する函数 where と多次元配列のままで検出する函数 where2 の二種類の函数があります:

非零点を検出する函数

構文 (where, where2)
where(⟨ 配列 ⟩)
where2(⟨ 配列 ⟩)

where 函数:　与えられた配列を 1 次元配列と見做し, その 1 次元配列に対して零と異なる成分の位置を成分とする 1 次元配列を返却する函数です:

```
> a =[[1,0,3,4],[0,0,3,1]];
> a(*)
 [1,0,3,4,0,0,3,1]
> where(a)
 [1,3,4,7,8]
```

この例で示すように与えられた配列 a を 1 次元配列化した場合に ‘0’ と異なる個所の位置が返されていることが分ります.

where2 函数:　与えられた配列の零と異なる成分の位置を返却する函数で, where 函数が 1 次元配列として返却するのに対して where2 函数は与えられた配列の添字を成分とする配列を返却します. たとえば配列 A が $n_1 \times \cdots \times n_m$ の大きさで非零となる成分の総数が k 個であれば, k 個の長さ m の 1 次元配列を成分とする配列, すなわち大きさ $m \times k$ の 1 次元配列が返却されます:

```
> a =[[1,0,3,4],[0,0,3,1]];
> where2(a)
    [[1,1],[3,1],[4,1],[3,2],[4,2]]
> dimsof(where2(a))
 [2,2,5]
```

このように where 函数とは違い, 多次元配列であればその添字総数に対応する長さの 1 次元配列を成分とする配列を返却します.

3.6.3 条件指定で配列生成を行う函数

Yorick では与えられた配列をある条件で二分し, その条件に応じた配列を対応させることで新たな配列を生成することができる函数を持っています[2]:

条件指定で配列生成を行う函数

構文 (merge, merge2, mergef)
merge(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩, ⟨ 条件文 ⟩)
merge2(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩, ⟨ 条件文 ⟩)
mergef(⟨ 配列$_1$⟩, ⟨ 函数名$_1$⟩, ⟨ 配列$_2$⟩, ⟨ 函数名$_2$⟩)

merge 函数: merge 函数は二つの配列を混合して新しい配列を生成する函数です. 配列の大きさは ⟨ 条件文 ⟩ で規定される配列の大きさと一致し, さらに ⟨ 配列$_1$⟩ の長さは ⟨ 条件文 ⟩ で規定される配列の非負成分の総数と同じでなければなりません.
つまり, この merge 函数では与えられた配列 X から ⟨ 条件文 ⟩ によって '0' と '1' のみの配列を生成し, 値が '1' の成分には ⟨ 配列$_1$⟩ の成分を値が '0; の成分に ⟨ 配列$_2$⟩ の成分を順番に対応させることで新しい配列を生成します:

```
> rl=(x>4)
> a=x(where(rl))^3
> nrl=!(x>4)
> b=1.0/x(where(nrl))
> merge(a,b,rl)
 [1,0.25,125,0.5,216,512,729]
```

ここで配列は 1 次元に限定されません:

```
> x =[1,4,5,2,6,8,9](,-:1:2)
> x
   [[1,4,5,2,6,8,9],[1,4,5,2,6,8,9]]
> rl=(x>4)
> a=x(where(rl))^3
> nrl=!(x>4)
> b=1.0/x(where(nrl))
> merge(a,b,rl)
   [[1,0.25,125,0.5,216,512,729],[1,0.25,125,0.5,216,512,729]]
```

このように多次元配列に対しても利用可能です.

merge2 函数: merge2 函数は merge 函数を使って Yorick 言語で記述された函数です.

[2]mergeg 函数はバグがあるために解説から除外しています

この函数の引数の二つの配列と条件文から規定される配列の大きさは全て等しく, 条件文から規定される配列の 1 となる部分が $\langle$ 配列$_1\rangle$ から同様に '0; となる部分が $\langle$ 配列$_2\rangle$ に対応付けられて一つの配列に纏めて出力されます:

```
> merge2 ([1,2,3,4]^3,1.0/[1,2,3,4],[1,2,3,4]>3)
 [1,0.5,0.333333,64]
```

この例では '[1,2,3,4]>3' で規定される配列は '[0,0,0,1]' となり, この '1' となる部分に配列 '[1,2,3,4]^3' の第 4 成分が対応させられます. そして残りの部分は配列 '1.0/[1,2,3,4]' の条件文で規定される配列の '0' の部分に対応付けられて結果の配列が得られているのです.

mergef 函数: 第 1 引数と第 3 引数の配列の大きさは同じもので, $\langle$ 配列$_2\rangle$ が 0 となる位置に $\langle$ 函数$_2\rangle$ を作用させ, それ以外の個所に $\langle$ 函数$_1\rangle$ を作用させます.

```
> x =[1,2,3,4,5,6,7,8,9,10]; y=mergef(x,exp ,[−1,2,3,0,0,0,0,0,0,0],
> y
 [2.71828,7.38906,20.0855,1.38629,1.60944,1.79176,1.94591,2.07944,
2.30259]
```

この例で示すように第 3 引数の配列では最初の 3 成分が '0' でないために函数 exp を配列 x の最初の 3 成分のみに作用させ, 残りは log 函数を作用させています. この第 3 引数の配列が丁度条件文に対応する訳ですが論理積 "$\wedge$" と論理和 "$\vee$" を通常の積 "*" と和 "+" で置換えることになります. たとえば次の函数を描いてみましょう:

$$\phi(x) = \begin{cases} \exp\left(1 - \frac{1}{1-|x|^2}\right) & |x| < 1 \\ 0 & |x| \geq 1 \end{cases}$$

この函数を表示するためには $\langle$ 配列$_2\rangle$ の指定方法が重要です. この場合は 'abs(x)>=1' のように函数を使って配列を生成ればよいのですが, これと同値な論理式 '$x \geq 1 \vee x \leq 1$' を用いる場合, Yorick の if 文で用いる論理積 "&&" や論理和 "||" は使えないので和 "+" を用いて '(x>=1)+(x<=-1)' とします. また mergef 函数では第 1 引数と第 3 引数には函数名を引渡さなければなりません. そのために Yorick の函数として定数函数 '0' と函数 'exp(1-1/(1-x^2))' を定義しなければなりません:

```
x=span(−2,2,1001);
func zero(x){return array(0,dimsof(x));};
func p1(x){return exp(1−1/(1−x^2));};
y=mergef(x,zero,(x>=1)+(x<=−1),p1);
fma;
plg,y,x;
```

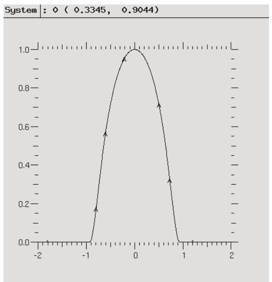


図 3.1: mergef で生成した数列のグラフ

3.7 配列に関連する述語

配列に関連する述語は 0 次元配列の ‘0’ か ‘1’ の何れか一方のみを返却します:

配列に関連する述語

構文	概要
allof(⟨ 配列 ⟩)	数値配列の成分が全て 0 でなければ 1, そうでなければ 0.
anyof(⟨ 配列 ⟩)	数値配列に 0 でない成分が存在すれば 1, そうでなければ 0.
nallof(⟨ 配列 ⟩)	数値配列に 0 が存在すれば 1, そうでなければ 0.
noneof(⟨ 配列 ⟩)	数値配列が全て 0 であれば 1, そうでなければ 0.

第4章　Yorickの演算子

Hamlet	ハムレット
Where be your gibes now? your gambols? your songs?	お前の毒舌は何処へ行った？ お前の踊りは？ お前の歌は？
	Hamletl: 第五幕, 第一場

4.1　Yorick の割当・代入の演算子

演算子 “=”:　Yorick の割当・代入を行う演算子です．この演算子 “=” は左辺の被演算子に右辺の被演算子の値を割当・代入を行いますが `a=b=c=d=1;` のような代入文も許容され, 最右辺の被演算子の値がその他の被演算子に割当・代入されます.

4.2　Yorick の算術演算子

yorick の四則演算 (和, 差, 積, 商) と冪と剰余の表記は C と同様です:

Yorick の算術演算子

演算子	概要	例
+	和	1+2
-	差	1-2
*	積	2*3
/	商	6/3
^	冪	3^2
%	剰余 (complex 型を除く).	5%3

これらの二項演算は同じ型同士の演算であれば同じ型が返され, 異った型の演算であれば優位度の上位にある被演算子の型で返されます．型の優位度を次に示しておきます.

型の優位度

char < short < int < long < float < double < complex

整数同士の二項演算:　二項演算子によって char 型が得られるのば char 型同士に限定され, short 型が得られるのは char 型と short 型の場合と short 型同士の場合, 同様に int 型が得られるのは一方が char 型か short 型でもう一方が int 型の場合と int 型同士の演算となります．それ以外の整数の型の二項演算では全て long 型です．整数同士の二項演算は整数になるために演算子 “/” による処理 ‘a/b’ は ‘a<b’ が真であれば ‘0’ になります.

実数同士の二項演算:　実数同士の二項演算子による演算で float 型となるのは float 型同士に限定され, float 型と double 型, そして double 型同士の二項演算は double 型になります.

整数と実数の二項演算:　一方が整数でもう一方が実数の場合, 実数の float 型や double 型が整数の任意の与件型よりも優位にあるために実数側の型で値が返されます.

複素数の二項演算:　複素数を表現する complex 型の対象同士の二項演算は complex 型が返されます. たとえば $(1+2i)\times(1-2i)$ は数学的には 5 ですが, yorick では '5+0i' になって complex 型のままです. このように complex 型の演算結果で虚部が 0 となる場合でも Yorick は虚部が '0i' と complex 型のままです. また演算子 "%" は被演算子のどちらか一方が complex 型であれば使えません.
次に実数と複素数の二項演算では複素数を表現する complex 型が優位にあるために返却される型が complex 型になります.

配列を被演算子とする場合:　結果の型は優位にある被演算子の型になります. ここでの配列は利用者が定義した構造体を成分とする配列は除外して Yorick 組込の与件を用いた配列に限定します. 何故なら, 利用者定義の構造体に対して二項演算子がそのまま拡張されないためです.
演算子を "$\circ$" として $a\circ b$ の結果を a と b の配列の大きさで分類した表を示します.

演算子の分配

1.	$\mathrm{dimsof}(a)=\mathrm{dimsof}(b)$	$\Rightarrow$
	$(a\circ b)(i_1,\ldots,i_n)=a(i_1,\ldots,i_n)\circ b(i_1,\ldots,i_n)$	
2_a	$\mathrm{dimsof}(a)=0$	$\Rightarrow$
	$(a\circ b)(i_1,\ldots,i_n)=a\circ b(i_1,\ldots,i_n)$	
2_b	$\mathrm{dimsof}(b)=0$	$\Rightarrow$
	$(a\circ b)(i_1,\ldots,i_n)=a(i_1,\ldots,i_n)\circ b$	
3_a	$a\subset_{\mathcal{A}} b$	$\Rightarrow$
	$(a\circ b)(i_1,\ldots,i_k,\ldots,i_m,\ldots,i_n)=a(i_k,\ldots,i_m)\circ b(i_1,\ldots,i_k,\ldots,i_m,\ldots,i_n)$	
3_b	$b\subset_{\mathcal{A}} a$	$\Rightarrow$
	$(a\circ b)(i_1,\ldots,i_k,\ldots,i_m,\ldots,i_n)=a(i_1,\ldots,i_k,\ldots,i_m,\ldots,i_n)\circ b(i_k,\ldots,i_m)$	

1. は被演算子が同じ大きさの配列の場合の処理で, 各配列の成分単位の演算になります. そして, 2_a. と 2_b. はベクトルのスカラ積に対応する処理です. 最後の 3_a. と 3_b. はベクトルのスカラ積を拡張した演算になります. ここで 3_a. と 3_b. の式 $a\subset_{\mathcal{A}} b$ の意味は配列 a の大きさの部分配列から配列 b が構成されること, 具体的には '$b=[a,\ a,\ldots,\ a]$', '$b=[,\ldots,[a,\ldots,a],\ldots,]$' や '$b=[[,\ldots,[a],\ldots,],\ldots,[,\ldots,[a],\ldots,]]$' のように演算子 "[]" と配列 a を使って配列 b が生成されるという意味です. さて 3_a は配列 a の大きさの配列から配列 b が構成されているときに配列 a と演算子 "$\circ$" がスカラの場合

のように対応する b の各成分に配分されます．3_b はその逆で配列 b の大きさの配列から配列 a が構成されるときに配列 b と演算子 “◦” をスカラの場合のように対応する a の各成分に配分されることを意味しています．

4.2.1　C 風の算術に関連する構文

C 風の構文で利用するできるものがあります:

C 風の和と差の演算子

演算子	概要	例
++	対象に 1 を増分する	x ++
−−	対象から 1 を差分する	x −−

これら二つの演算子は変数に対してのみ利用可能な演算子で C で普通に利用される構文です．この演算子を利用すると 1 の加算や減算と代入が容易に行えます．
この他にも次の演算子が使えます．

C 風の演算子

演算子	構文例	同値な表現
+=	x+=a	x=x+a
*=	x*=a	x=x*a
/=	x/=a	x=x/a
%=	x%=a	x=x%a

4.2.2　2 進数表現に関連する演算子

Yorick には整数を 2 進数表現に内部で変換し，その 2 進数表現に対して処理を行う演算子があります:

2 進数表現に関連する演算子

演算子	概要	例
&	桁毎の AND 演算子	1&3
|	桁毎の OR 演算子	1|3
˜	桁毎の XOR 演算子	1˜3
˜	被演算子の 1 の補数を返す	˜3

演算子 “ & ”: 2 個の整数被演算子を取り, それらの 2 進数表現に対しする “AND” を取ります. すなわち 2 進数表現で同じ桁が 1 なら 1, 異なれば 0 で置換えた数を返却します:

```
> 7&1
1
> 7&3
3
```

この例で ‘7’ は 2 進数表現で ‘111’, ‘3’ は ‘11’ となります. ‘7&1’ は 1 桁目だけが共に ‘1’ となるために ‘1’ となりますが, ‘7&3’ は 1 桁目と 2 桁目が共に ‘1’ となるので 2 進数表現で ‘11’ を得ますが, 10 進数表現では整数 2^1+1 が対応するので ‘3’ になります.

演算子 “ | ”: 2 個の整数被演算子を取り, それらの 2 進数表現に対しする OR を取ります. すなわち 2 進数表現で同じ桁が ‘0’ の場合のみ ‘0’ とし, それ以外は ‘1’ に置換えた数を返却します:

```
> 7|1
7
> 7|3
7
```

この例では 7 の 2 進数表現が 111, 3 の 2 進数表現が 11 なので, これらの各桁で OR を取れば ‘7|1’ も ‘7|3’ も ‘7’ になります.

演算子 “ ˜ ”: 演算子 “ - ” と同様の利用が可能ですが意味は異なります. この演算子は基本的に二つの被演算子の各桁毎に “XOR” を取ります. ここで “XOR” とは a と b を bit とするときに ‘a OR b - a AND b’ を計算する演算です:

```
> 1˜0
1
> 1˜1
0
> 0˜0
```

また被演算子を 1 つだけ取る場合は前置式の演算子となり, この場合は被演算子の 1 の補数を返却します. したがって ‘a+(˜a)=-1’ を満します:

```
> 10+(˜10)
−1
> 18908901+(˜18908901)
−1
```

4.3　Yorick の論理演算子

Yorick の論理演算子は「比較の演算子」と Boole 演算のための「論理演算」の演算子から構成されます.

4.3.1　比較の演算子

Yorick の「**比較の演算子**」は C と同様の表記で, これらの演算子の返却値は C と異なって真であれば int 型の ‘1’, 偽であれば int 型の ‘0’ になります:

比較の演算子

演算子	概要	例
>=	以上	2>=1
<	大なり	2<1
<=	以下	2<=3
<	小なり	6<3
==	等しい	3==3
!=	等しくない	3!=1

大小関係:　大小関係の演算子は整数や実数, および文字列で利用できます. ここで被演算子の双方が整数や実数といった数であれば通常の数の比較となり, 与件型が異っていても構いませんが, 被演算子の一方が文字列文字列であれば, もう一方も文字列でなければなりません. また文字列については ‘小文字 > 大文字 > 数字’ の順序があり, アルファベットでは ‘”z”> . . . >”a”’, 数字については ‘”9”> . . . >”0”’ の「辞書式順序」となります:

```
> "a">"b"
0
> "A">"b"
0
> "a">"b"
0
> "a">"B"
1
> "abb">"abcd"
0
```

被演算子が 1 次元以上の配列の場合, 双方が同じ大きさの配列か, どちらか一方が 0 次元の配列であれば成分毎に比較を行います:

```
> y0=span(1,10,10)
> y1=span(0,11,10)
> y0
 [1,2,3,4,5,6,7,8,9,10]
> y1
[0,1.22222,2.44444,3.66667,4.88889,6.11111,7.33333,8.55556,9.77778,11]
> y0<y1
 [0,0,0,0,0,1,1,1,1,1]
> y0>5
 [0,0,0,0,0,1,1,1,1,1]
```

これら性質を利用することで if 文を算術演算に置換えられます. たとえば数値リストの各成分に対して ‘3’ より小であれば ‘10’, ‘3’ 以上なら ‘-5’, ‘3’ ならば ‘0’ を対応させる処理が次の算術演算で行えます:

```
> A=[1,2,3,4,5];
> B=(A>3)*10+(A<3)*(−5)+(A==3)*0
> B
[−5,−5,0,10,10]
```

この処理は MATLAB 系の言語で可能で C と大きく異なる点です.
なお, 演算子 “==” と演算子 “!=” は任意の型に対して利用できますが, これらの演算子は int 型の整数 ‘0’ か ‘1’ のみを返却して成分毎の比較は行えません.

等号の演算子:　大小関係の演算子とやや違った振舞をする演算子です. 被演算子が数値同士や文字列同士の場合は成分単位の処理を行いますが数値と文字列のように完全に被演算子の型が異なれば ‘0’ か ‘1’ のみを返却します:

```
> [”abc”,”cdef”]==[”abc”,”cde”]
[1,0]
> [”abc”,”cdef”]!=1
1
```

この例で示すように型が完全に異なった場合には成分単位の演算ではなく全体の演算となるために int 型の ‘0’ か ‘1’ のみの返却になります.

4.3.2　論理和, 論理積と否定

Yorick の論理和と論理積の演算子は C と同様の表記になりますが整数以外の対象も被演算子になります. また否定の演算子 “！” は数学で用いる階乗の演算子とは違って論理式や対象の前に置きます. ここで論理積と論理和の被演算子は 0 次元の配列に

限定されますが否定の演算子には制約はありません．この点は論理積と論理和との性格の違いによるものです．

論理和, 論理積と否定

演算子	概要	例
||	論理和	(1<2)||(3<2)
&&	論理積	(1<2)&&(3<2)
!	否定	!(1<2)

論理和 “ || ”:　被演算子の双方が ‘0’ の場合のみに ‘0’ を返却し, それ以外は ‘1’ を返却する演算子です:

```
> 0||0
0
> 0||−1
1
> 2||5
1
```

論理積 “ && ”:　被演算子のどちらか一方が ‘0’ ならば ‘0’ を返却し, それ以外は ‘1’ を返却する演算子です:

```
> 0&&0
0
> 0&&−1
0
> 2&&5
1
```

否定 “ ! ”:　被演算子が ‘0’ であれば ‘1’ を返し, ‘0’ 以外の値であれば ‘0’ を返却する演算子です:

```
> !10
0
> !0
1
```

なお, Yorick には階乗演算子 “ ! ” を持たないので否定の演算子と混同することはないでしょう．また, 否定の演算子 “ ! ” の優先度は他の演算子よりも強いので否定する式を括弧 “( )” で括ると間違いが少ないでしょう:

```
> !2*0
0
> !(2*0)
1
```

この例で示しているように最初の式では括弧を用いていないために Yorick は '(!2)*0' と解釈して '0' を返却していますね.

4.4 配列の添字を活用した四則演算

4.4.1 概要

Yorick で四則演算子を通常の数式のように用いる限り, 行列同士や行列とベクトルの積は行えません. そこで, Yorick では添字を活用することで行列の積やベクトルの内積等を表現します. この性質は MATLAB と比較して, ある種の捉え所のなさに繋がりますが, 添字の活用は配列の表記上の順番に捕われないことや 3 次以上の配列に対しても特定の成分に対して行列演算やベクトル演算が容易に行えるという長所を持ちます.

4.4.2 配列の演算処理

同じ大きさの配列の演算

一般論に移る前にベクトルを使った簡単な例から始めてみましょう:

```
> a=indgen(1:5);
> b=a(::-1);
> print,a,b
 [1,2,3,4,5]     [5,4,3,2,1]
> a*b
 [5,8,9,8,5]
> a/b
 [0,0,1,2,5]
> a % b
 [1,2,0,0,0]
```

同じ長さの 1 次元配列 a と b に対して 'a*b' は配列の成分毎の積になります. つまり, n 次元の同じ大きさの配列 a, b に対し, '$(a*b)(i_1,\ldots,i_n) = a(i_1,\ldots,i_n)*b(i_1,\ldots,i_n)$' を満します. つまり, 二項演算 "◯" を論理積 "&&" と論理和 "||" を除く任意の Yorick

の二項演算子とするときに '$(a \bigcirc b)_{i_1 \ldots i_n} = a_{i_1 \ldots i_n} \bigcirc b_{i_1 \ldots i_n}$' が成立しますが, このことから判るように Yorick の二項演算子は MATLAB 系の言語の二項演算子 "◯" の成分毎の演算子 ".◯" [1] に相当します.

4.4.3　一般的な次元と大きさの配列の演算

配列 $\{A\}_{M_1,\ldots,M_m}$ と n 次元配列 $\{B\}_{N_1,\ldots,N_n}$ が与えられて $m \geq n$ とします. このときに算術演算子 "◯" による結果は m 次元の配列であり, この配列 $\{A \bigcirc B\}_{L_1,\ldots,L_m}$ の i 番目の次数の大きさ L_i は次で与えられます:

- $M_i = N_i$ であれば $L_i = M_i$
- $M_i = 1$ または $N_i = 1$ であれば $L_i = \max(M_i, N_i)$
- その他は $A \bigcirc B$ の計算はできない

ここでは簡単な例で確認しておきましょう:

```
> a=indgen(1:5)
> b=a(::-1)
> a2=a(,-:1:3)(-,)
> print,a2,dimsof(a2)
 [[[1],[2],[3],[4],[5]],[[1],[2],[3],[4],[5]],[[1],[2],[3],[4],[5]]]     [3,1,5,3]
> print,b,dimsof(b)
 [5,4,3,2,1]    [1,5]
> ab=a*b
> print,ab,dimsof(ab)
 [5,8,9,8,5]    [1,5]
> a2b=a2*b
> print,a2b,dimsof(a2b)
 [[[5,4,3,2,1],[10,8,6,4,2],[15,12,9,6,3],[20,16,12,8,4],[25,20,15,10,5]],[[5,4,
 3,2,1],[10,8,6,4,2],[15,12,9,6,3],[20,16,12,8,4],[25,20,15,10,5]],[[5,4,3,2,1],
 [10,8,6,4,2],[15,12,9,6,3],[20,16,12,8,4],[25,20,15,10,5]]]     [3,5,5,3]
```

配列 a と配列 b が共に大きさが 1×5 の 1 次元配列のために最初の 'a*b' は成分毎に演算子 "*" による作用となっています. このときの配列 'a*b' の大きさは 5 で, 配列 a と b と同じ大きさとなります.
次に 3 次元の配列で大きさが $1 \times 5 \times 3$ の配列 a2 と 1 次元で大きさが 1×5 の配列 b の積 'a2*b' を計算しています. この配列 a2b は 3 次元の配列となり, 大きさは $5 \times 5 \times 3$

[1] 演算子 "∘" を演算子 "*" とするとき, 演算子 ".∘" には ".*" が対応.

となります. つまり, 次元の小さい方は, 大きい方に合せられ, 各添字も大きい方に合せて新しい配列が生成されます.
そこで具体的に配列 $\{A\}$ と配列 $\{B\}$ の二項演算 “◯” による演算の方法を解説しましょう. ここで配列 A の次元を m, 配列 B の次元を n として話を進めます:

- $m = n$ かつ $M_1 = N_1, \ldots, M_m = N_m$ の場合
  このときは得られる配列の次元を m として成分毎の演算を実行する.

  すなわち $\{A \bigcirc B\}(i_1 \ldots i_m) = \{A\}(i_1 \ldots i_m) \bigcirc \{B\}(i_1 \ldots i_m)$

- $m \neq n$ の場合
  記号 “[]” で配列を括ることによって配列の次元の拡大を行う. このとき配列の不足数 $k = |n - m|$ に対して記号 “(.. , -)” を k 回, 不足側の配列に作用させる:

  1. $m < n$ であれば次で置換える:
  $$\{A\}_{M_1 \ \ldots \ M_m} \underbrace{1 \ldots 1}_{k} \stackrel{def}{=} \{A\}_{M_1 \ \ldots \ M_m} \underbrace{(..,-) \ldots (..,-)}_{k}$$
  2. $m > n$ であれば次で置換える:
  $$\{B\}_{N_1 \ \ldots \ N_n} \underbrace{1 \ldots 1}_{k} \stackrel{def}{=} \{B\}_{N_1 \ \ldots \ N_n} \underbrace{(..,-) \ldots (..,-)}_{k}$$

  これらの処理によって $m = n$ の場合に帰着される.

- $m = n$ だが $M_k \neq N_k$ となる次数が存在する場合

  1. $M_k = 1$ であれば配列 $\{A\}_{M_1 \ldots M_k \ldots M_n}$ の k 成分を拡張する. つまり M_k を N_k に合せ, $i \in \{1, \ldots, M_k\}$ に対して $A(i_1, \ldots, \overset{k}{\check{i}}, \ldots, i_m)$ を $A(i_1, \ldots, \overset{k}{\check{1}}, \ldots, i_n)$ とすることで次数 k を N_k に合せる.
  2. $N_k = 1$ であれば配列 $\{B\}_{N_1 \ldots N_k \ldots N_n}$ の k 成分を拡張する. つまり N_k を M_k に合せ, $i \in \{1, \ldots, M_k\}$ に対して $B(i_1, \ldots, \overset{k}{\check{i}}, \ldots, i_n)$ を $B(i_1, \ldots, \overset{k}{\check{1}}, \ldots, i_n)$ とすることで次数 k を M_k に合せる.

  以上の操作により, $m = n$ かつ $M_1 = N_1, \ldots, M_m = N_m$ の場合に帰着できる.

こように Yorick の配列の演算では配列の次元と大きさを自動的に合せて演算が行なわれます.

4.4.4　添字 “+” を併用した積

積演算 “*” を行う際に添字 “+” を用いることで内積や行列の積が導入できます．この添字の利用方法は sum 等の添字の利用方法と似た考え方になります．ここでは 1 次元配列で観察してみましょう:

```
> a=[1,2,3,4,5]
> b=[5,4,3,2,1]
> a(+)*b(+)
35
> (a*b)(sum)
35
```

‘a(+)*b(+)’ の意味は, 同じ長さのベクトル a と b の各成分の積を計算して, その総和を取るという意味で, ‘(a*b)(sum)’ と同じ意味ですが, 別の見方をすれば添字 “+” が置かれた次数の添字を動かして総和を取ること, すなわち $\sum_{k=1}^{n} a(k) * b(k)$ を意味します．したがって添字 “+” が置かれる添字は同じ大きさでなければなりません．
次に, 2 次元配列 $\{A\}_{m,n}$, 2 次元配列 $\{B\}_{n,s}$ に対して行列積 $\{AB\}_{m,s}$ の i 行 j 列 の成分 $AB_{i,j}$ は $\sum_{k=1}^{n} A_{i,k}B_{k,j}$ で得られます．ここで Yorick 風の添字を用いるのであれば添字を動かす側に添字 “+” を配置することになるので ‘A(,+)*B(+,)’ が対応します．
ここで簡単な例で確認しておきましょう:

```
> a=indgen(1:5)(,-:1:2)
> print,a,dimsof(a)
  [[1,2,3,4,5],[1,2,3,4,5]]        [2,5,2]
> b=transpose(a(::-1)(,-:1:2))
> print,b,dimsof(b)
   [[5,5],[4,4],[3,3],[2,2],[1,1]]          [2,2,5]
> ab=a(,+)*b(+,)
> print,ab,dimsof(ab)
     [[10,20,30,40,50],[8,16,24,32,40],[6,12,18,24,30],[4,8,12,16,20],[2,4,6,8,10]]
 [2,5,5]
> ba=a(+,)*b(,+)
> print,ba,dimsof(ba)
 [[35,35],[35,35]]       [2,2,2]
```

この例では, 二つの 2 次元配列 a と b を構築し, ‘a(,+)*b(+,)’ と ‘a(+,)*b(,+)’ を計算させています．ここで, 配列 a の大きさは 5×2, 配列 b の大きさは 2×5 となっています．そして, Yorick の 2 次元配列では記号 “[ ]” で括られた成分が列に対応するた

めに配列 a を行列 a としてみると,

$$a = \begin{pmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \\ 4 & 4 \\ 5 & 5 \end{pmatrix}$$

配列 b を行列 b としてみると,

$$b = \begin{pmatrix} 5 & 4 & 3 & 2 & 1 \\ 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

となります. このとき 'a(,+)*b(+,)' が行列の積 ab, 'a(+,)*b(,+)' が行列の積 ba にそれぞれ対応することが判ります.
より一般の m 次元配列 A と n 次元配列 B に対しては, 配列 A の次数 k と配列 B の次数 h の大きさが同じ t となる場合に限って積 $A(,\ldots,\overset{k}{\check{+}},\ldots,)*B(,\ldots,\overset{h}{\check{+}},\ldots,)$ が計算可能で, この処理によって新しい $m+n-2$ 次元配列 $\{AB\}$ が

$$\begin{aligned} &AB(i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_M,j_1,\ldots,j_{h-1},j_{h+1},\ldots,j_n) = \\ &\sum_{i=1}^{t} A(i_1,\ldots,i_{k-1},\overset{k}{\check{i}},i_{k+1},\ldots,i_m)*B(j_1,\ldots,j_{h-1},\overset{h}{\check{i}},j_{h+1},\ldots,j_n) \end{aligned}$$

で得られます.

```
> a=indgen(1:5)(,−:1:2)
> a2=a(−,)(−,);print,a2,dimsof(a2)
        [[[[1]],[[2]],[[3]],[[4]],[[5]]],[[[1]],[[2]],[[3]],[[4]],[[5]]]]        [4,1,1,5,2]
> b=a(*,1)(::−1)(−:1:2,)
> b2=[[[b ]]](−,); dimsof(b2)
 [6,1,2,5,1,1,1]
> ab2=a2(,,,+,)*b2(,+,,,,)
> ab2
        [[[[[[[10]],[[20]],[[30]],[[40]],[[50]]]],[[[[8]],[[16]],[[24]],[[32]],
        [[40]]]],[[[[6]],[[12]],[[18]],[[24]],[[30]]]],[[[[4]],[[8]],[[12]],[[16]],
      [[20]]]],[[[[2]],[[4]],[[6]],[[8]],[[10]]]]]]]
> print,dimsof(ab2)
 [8,1,1,5,1,5,1,1,1]
> ba2=a2(,,+,,,)*b2(,,+,,,)
> ba2
     [[[[[[[35]],[[35]]]],[[[[35]],[[35]]]]]]]
> print,dimsof(ba2)
 [8,1,1,2,1,2,1,1,1]
```

このように添字の利用は一見すると非常に煩雑で MATLAB とは別の考え方をしなければなりませんが, この表現は数式表現に密着した表現方法であり, この表現の御陰で高次元の配列の処理が容易になっています.

第5章　Yorickの基本的な函数

First Clown	第一の墓掘人
It must be “se offendendo”;	そりゃあ、「被疑者甲ハ正当攻撃ニシテ...」だ;
it cannot be else.	それっきゃない.
For here lies the point:	つまりだ:
if I drown myself wittingly,	俺がわざと溺れたとすりゃ,
it argues an act:	そりゃ, やっちまったんだ:
and an act hath three branches:	で, やっちまったってこたぁ三段ナリ:
it is, to act, to do, to perform:	即ち,(A) やぁるナリ, (O) 行うナリ, (E) 演ずるナリの AOEって訳さ;
argal,she drowned herself wittingly.	だーから, あの嬢ちゃんはわざと溺れたのさ.

Hamlet: 第五幕, 第 1 場

5.1　i0 ディレクトリに収録されたライブラリ

この章では Yorick の基本的な函数について解説します．ここでの基本的な函数は大域変数 Y_SITE で指示されたディレクトリ下の “i0” ディレクトリに包含されるライブラリ (拡張子が “.i” のファイル) 内で定義された函数を指します．特に基本的な函数は “std.i” ライブラリに多く記述されており, このファイルに含まれる函数を中心に解説を行うことにします．なお, 多くの函数が Yorick 言語ではなく Yorick のソースファイルで定義されており, その場合には extern 文を用いて同名の大域変数を宣言し, その選言に付随する解説としてオンラインマニュアルの内容を記述しています．たとえば, 絶対値を計算する abs 函数は次のように “std.i” ライブラリに記述されています:

```
extern abs;
/* DOCUMENT abs(x)
         or abs(x, y, z, ...)
     returns the absolute value of its argument.
     In the multi-argument form, returns sqrt(x^2+y^2+z^2+...).
   SEE ALSO: sign, sqrt
 */
```

この abs 函数の実体は Yorick のソースファイルの “std0.c” 内で Y_abs 函数として定義されています.
ここで Yorick のライブラリを集めた重要なディレクトリに “i” ディレクトリがあります．このディレクトリは “i0” ディレクトリと同じ大域変 Y_SITE で指示された階層に置かれ, Yorick のライブラリが収納されています．このディレクに置かれた重要なライブラリと函数の概要は §6 を参照して下さい.

5.2　基本的な数値函数

abs 函数:　絶対値を返却する函数です．この函数は引数の型が comple 型以外ならば返却値の型と一致し, 引数の型が complex 型の場合は double 型の結果を返却します.

sign 函数:　実数に対しては符号を返却する函数です．より正確には引数の絶対値で引数を割った値を引数の型に変換して返す函数です:

```
> sign(-4)
-1
> sign(-4.5)
```

```
−1
> typeof(sign(−4))
"long"
> typeof(sign(−4.5))
"double"
```

引数が complex 型であれば複素単位円上の点を返却値とします:

```
> sign(1+1i)
0.707107+0.707107i
> sign(1+0i)
1+0i
```

floor 函数: complex 型以外の数値を引数とし, 引数を越えない整数を返す函数です. ただし, この floor 函数は任意の数値型の引数に対して double 型の数値を返す函数で, 結果が long 型の対象のように見えても long 型ではないことに注意が必要です:

```
> floor(4.5)
4
> floor(4)
4
> floor(−4)
−4
> floor(−4.5)
−5
> typeof(floor(−4.5))
"double"
> typeof(floor(−4))
"double"
```

そのために必要に応じて char, int, short, long 等の型の変換函数を用いれば型に関連する誤作動を減らすことができます:

```
> typeof(int(floor(−4)))
"int"
```

ceil 函数: complex 型以外の数値を引数とし, 引数の絶対値を越えない整数を返す函数です. この ceil 函数も floor 函数と同様に任意の数値型の引数に対して double 型の数値を返す函数で, 結果が long 型のように見えていても long 型ではないことに注意が必要です. この ceil 函数についても必要に応じて型の変換函数を用いると良いでしょう.

conj 函数:　数値を引数とし, その数値に対応する共役複素数を返却する函数です. この函数は complex 以外であれば引数をそのまま返却するため, 結果の型は引数の型と一致します. 当然, complex 型の引数に対しては complex 型の結果を返却します:

```
> conj(4)
4
> typeof(conj(4))
"long"
> typeof(conj(5.f))
"float"
```

avg 函数:　与えられた数値配列の算術平均を計算する函数です. 返却値の型は引数の型が complex 型以外であれば double 型, 引数の型が complex 型であれば返却値も complex 型になります. 添字 "avg" を利用するよりも avg 函数を用いる方が処理は高速ですが配列全体の平均値を返却します:

```
> a =[[1,2,3],[4,5,6]]
> avg(a)
3.5
> a(avg,)
 [2,5]
> a(,avg)
 [2.5,3.5,4.5]
> a(avg,avg)
3.5
```

max 函数:　与えられた complex 型以外の数値配列で, その最大値を返す函数です. 返却値の型は引数の型と一致します. 添字 "max" を利用するよりも max 函数の方が高速ですが, max 函数は配列全体の最大値を返却します:

```
> a =[[1,2,3],[4,5,6]]
> max(a)
6
> a(max,)
 [3,6]
> a(,max)
 [4,5,6]
> a(max,max)
6
```

min 函数: 与えられた complex 型以外の数値配列で, その最小値を返す函数です. 返却値の型は引数の型と一致します. 添字 “min” を利用するよりも min 函数の方が高速ですが全体の最小値を返却します:

```
> a =[[1,2,3],[4,5,6]]
> min(a)
1
> a(min,)
 [1,4]
> a(,min)
 [1,2,3]
> a(min,min)
1
```

sum 函数: 与えられた数値配列の総和を計算する函数です. 返却値の型は引数の型と一致します. 添字 “sum” を利用するよりも sum 函数の方がより高速ですが, 配列全体の総和を返却します:

```
> a =[[1,2,3],[4,5,6]]
> sum(a)
21
> a(sum,)
 [6,15]
> a(,sum)
 [5,7,9]
> a(sum,sum)
21
```

5.3 初等函数

Yorick の初等函数には sin, cos 等の三角函数, および逆三角函数, sinh, cosh 等の双曲函数とその逆双曲函数, exp や log といった指数函数や対数函数があります. これらの函数は整数や実数引数に対しては double 型の結果, comolex 型の引数に対しては complex 型の結果を返します.

5.3.1 三角函数と逆三角函数

三角函数とそれらの逆函数を纏めておきます. これらの函数は全て弧度法 (radian) に基くものです. 逆正接函数 atan を除く函数は全て引数を 1 つだけ取って引数が複素数であれば返却する値も複素数になります:

三角函数と逆三角函数

函数	概要
sin	正弦函数
cos	余弦函数
tan	正接函数
asin	逆正弦函数
acos	逆余弦函数
atan	逆正接函数

ここで atan 函数以外は引数を 1 つのみ取りますが, atan 函数は引数を 1 つ, あるいは 2 つ持たせることができます. また各引数は整数, あるいは実数型のみに対応しています. atan 函数はやや特殊で引数が 1 つであれば値域を $(-\pi/2, \pi/2)$, 引数が 2 つであれば値域を $(-\pi, \pi]$ として複素数は扱えません. そして 'atan(y,x)' に対しては 'atan(y/x)' を計算し, x を 0 と異なる実数とするときに 'atan(x,0)' は 'pi/2' の結果, 'atan(0,0)' は '0' を返却します.

5.3.2　双曲線函数と逆双曲線函数

次に双曲線函数を挙げておきましょう. なお, これらの函数は引数を 1 つだけ取り, 引数が複素数であれば複素数値を返却します:

双曲線函数

函数	概要
sinh	双曲線正弦函数: $(e^x - e^{-x})/2$
cosh	双曲線余弦函数: $(e^x + e^{-x})/2$
tanh	双曲線正接函数: $\sinh x/\cosh x$
sech	双曲線正割函数: $1/\cosh x$
csch	双曲線余割函数: $1/\sinh x$

Yorick には逆双曲線函数もあり, これらの函数は引数を 1 つだけ取り, 引数が複素数であれば複素数値を返却します.

逆双曲線函数

函数	概要
asinh	逆双曲線正弦函数: sinh 函数の逆函数です.
acosh	逆双曲線余弦函数: cosh 函数の逆函数で, その実部は常に 0 以上です.
atanh	逆双曲線正接函数: tanh 函数の逆函数です.

5.3.3 指数函数と対数函数

指数函数と対数函数

函数	概要
exp	指数函数. 引数は 1 つだけ取ります.
expm1	exp(x)-1 を計算します.
log	自然対数函数. 底を Napia 数 e を底とする対数函数で, その引数は 1 つだけです.
log1p	$\log(1+x)$ を計算する函数で引数は 1 つのみ.
log10	底が 10 の対数函数で, 引数は 1 つのみ.

expm1 函数は $e^x - 1$ を計算する函数で, 引数が 1 つの場合と 2 つの場合があります. 引数が 2 つの場合, すなわち, 'expm1(x,y)' は 'exp(x)-1' の計算結果を返却しますが, その際に 'exp(x)' の結果を変数 y に割当てます:

```
> expm1(0)
0
> expm1(0,a)
0
> a
1
```

5.4 統計に関連する函数

統計量に関連する函数

構文 (histogram, histinv, median)
histogram(⟨ 配列 ⟩)
histogram(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)
histinv(⟨ 配列 ⟩)
median(⟨ 配列 ⟩)

histogram 函数:　第 1 引数に与えられた配列をベクトルに置換し, そのベクトルに対する度数分布を返却する函数です. ここで度数分布は, 添字 'i' を持つベクトルの元の総数で, Yorick の配列の添字は整数の '1' から開始するために, 分布も 1 以上でなければなりません. すなわち, 第 1 引数の各成分の値は 1 以上でなければならず, 0, 負数や複素数, および文字列は許容されません.
また, 第 1 引数の配列の処理は, 配列 A が与えられたときに 'long(floor(A(*)))' とほぼ同値なベクトルに変換されます. ここで「ほぼ同等」と記した理由は, 配列 A の元 'a' と最も近い整数 'n' の差が微小の場合は切り上げが発生するからです. このことを簡単な例で確認してみましょう:

```
> y=5*sin(2*pi*span(0,1,10))+6
> histogram(y)
 [2,1,0,1,1,1,1,0,1,2]
> histogram(long(floor(y)))
 [2,1,0,1,1,1,1,0,1,2]
> y
 [6,9.21394,10.924,10.3301,7.7101,4.2899,1.66987,1.07596,2.78606,6]
> long(floor(y))
 [6,9,10,10,7,4,1,1,2,5]
> write,format="%30.30f\n",y(10)
5.999999999999999111821580299875
> 6==y(10)
0
> long(floor(y+2.0^(-51)))
 [6,9,10,10,7,4,1,1,2,6]
```

さて, 'long (floor(y))' では最後の成分が '5' となり, y の最後の成分の '6' と違いますね. その理由ですが, 'y' の最後の成分 'y(10)' の値は '6' ではありません. 実際は誤差によって '5.999999999999999111821580299875' と '6' と異なる浮動小数点数であり, floor 函数を用いれば小数点以下が切り捨てられるので '5' になります. ところが, histogram 函数では切り上げが発生しています. これは, a を浮動小数点数, n を a が最も近い整数, 浮動小数点数 ε を $\varepsilon = n - a$ とするとき, 浮動小数点数として $n = n + \varepsilon/2$ を満す場合に生じます. それ以外は, 小数点数以下は切捨となります.
また, histogram 函数に第 1 引数とベクトルとして同じ長さの第 2 引数を指定すれば, この第 2 引数を重みとして処理した度数分布を返却します.

histinv 函数:　与えられた度数分布の配列から本来の分布を復元する函数です:

```
> histinv ([1,2,3,4,5])
  [1,2,2,3,3,3,4,4,4,4,5,5,5,5,5]
```

なお, 配列の全ての成分が 0 以上の実数であれば利用可能です. また引数の配列が高次元の場合, 一旦, 平坦化してから処理が行われます.

median 函数: 第 1 引数に与えられた配列から中央値を計算する函数です. 中央値の性格上, 配列の長さが偶数であれば配列が整数型であっても浮動小数点数の値を返却します.

5.5 乱数に関連する函数

乱数に関連する函数

構文 (random, randomize)
random(⟨ 整数$_1$⟩, . . . , ⟨ 整数$_n$⟩)
random_seed(⟨ 数値 ⟩)
randomize()
randomize

random 函数: 整数列:⟨ 整数$_1$⟩, . . . , ⟨ 整数$_n$⟩ を引数とし, この整数列を配列の大きさとする乱数配列を返す函数です. 返却型は double 型で区間 $[0, 1]$ の間の値で構成されています. 乱数計算アルゴリズムは Press と Teukolsky によるものです.

rando_seed 函数: 乱数列の初期化を行う函数です. ここで与える数値は区間 $[0, 1]$ に含まれる実数にすべきで, 引数が nil やこの区間を越えた値を入力すると乱数列は Yorick を再起動したのと同様に初期化されてしまいます.

randomize 函数: 引数が不要の函数で, 計算機の時計等に基いて乱数の設定を行います. `randomize()` と入力すると random_seed 函数に引渡された値が表示されます.

5.6 補間と数値積分に関連する函数

補間と数値積分に関連する函数

構文 (digitize, iterp, integ)
digitize(⟨ 配列 ⟩, ⟨ ベクトル ⟩)
interp(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩, ⟨ 配列$_3$⟩)
integ(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩, ⟨ 配列$_3$⟩)

digitize 函数:　第 1 引数と同じ大きさの配列を返却する函数で, 第 1 引数の配列を $A_{M_1,\ldots,M_n}$, 第 2 引数のベクトルを V_N とするときに返却値となる新しい配列 Z の $(i_1,\ldots,i_n)$ 成分を次で定めます:

- $V(i-1) \le A(i_1,\ldots,i_n) \le V(k) \Rightarrow Z(i_1,\ldots,i_n) = k$
- $V(i-1) \ge A(i_1,\ldots,i_n) \ge V(k) \Rightarrow Z(i_1,\ldots,i_n) = k$
- $\max(\mathrm{V}) \le A(i_1,\ldots,i_n) \Rightarrow Z(i_1,\ldots,i_n) = \mathrm{numberof(V)}$
- $\min(\mathrm{V}) \ge A(i_1,\ldots,i_n) \Rightarrow Z(i_1,\ldots,i_n) = 1$

```
> digitize ([−5,1,2,3],[−1,0,1,2,3,4])
 [1,4,5,6]
```

この例では-5 が第 2 引数のベクトルの下限よりも小のために 1, 1 は 1<2 より 2 の添字 4, 2 と 3 も同様で 3 の添字の 5 と 4 の添字の 6 がそれぞれ対応します.

interp 函数:　線形補間を行う函数です. 具体的には $\langle$ 配列$_1\rangle$ を Y 座標のベクトル, $\langle$ 配列$_2\rangle$ を X 座標のベクトルとするとき, これらの配列を用いて区間線形函数を生成して, この区間線形函数による第 3 引数の値を返却する函数です.
この函数の特性上, 第 1 引数と第 2 引数のベクトルの長さは 2 以上の同じ長さでなければなりません.

```
> interp ([2,4,6,8,10],[1,2,3,4,5],2.2)
4.4
> interp ([2,4,6,8,10],[1,2,3,4,5],[2.2,4.56])
 [4.4,9.12]
```

integ 函数:　interp 函数で線形補間した函数を数値積分する函数です. そのために 第 1 引数と第 2 引数の配列は同じ大きさのベクトルに限定されます.

```
> integ ([2,4,6,8,10],[1,2,3,4,5],[2.2,4.56])
 [3.84,19.7936]
```

5.7　FFT に関連する函数

5.7.1　予備知識

ここでは Fourier 変換に関連する知識を軽くまとめておきます.

群: 集合 G が (二項) 演算と呼ばれる写像 $\circ: G \times G \to G$ を持ち, G の任意の元 x, y, z に対して次の公理を満すときに集合 G を「**群 (group)**」と呼びます:

群の公理

1.	$x \circ y \in G$	(演算 $\circ$ について閉じている)
2.	$(x \circ y) \circ z = x \circ (y \circ z)$	(結合律)
3.	$x \circ e = e \circ x = x$	(単位元 e の存在)
4.	$x \circ x^{-1} = x^{-1} \circ x = e$	(逆元 x^{-1} の存在)

ここで任意の $x, y \in G$ に対して $x \circ y = y \circ x$ を満す場合に群 G を「**可換群 (commutative group)**」と呼びます. ここで演算を明記するために $(G, \circ)$ と表記することもありますが, 演算を省略しても問題がなければ単に G と表記します. 群の例として, 整数の集合で構成される群 $(\mathbb{Z}, +)$, 実数の集合で構成される群 $(\mathbb{R}, *)$ を挙げておきます. また $(\mathbb{Z}, *)$ は 1 以外の元で逆元が存在しないために群になりませんが公理の 1. ,2. ,3. を満します. このように公理 4. のみを満さないものを「**半群 (semi group)**」と呼びます.

体: 集合 K が二つの演算: 和 "+", 積 "$*$" を持ち, 次の公理を満すときに「**体 (field)**」と呼びます:

体の公理

1.	$(K, +)$ は可換群
2.	$(K, *)$ は可換群
3.	K の任意の元に対し分配律: $\alpha * (\beta + \gamma) = \alpha * \beta + \alpha * \gamma$ を満す

演算を明記する場合は「体 $(K, +, *)$」と表記しますが, 問題がなければ単に「体 K」と表記します. ここで公理 2. を「公理 2′. $(K, *)$ が (可換) 半群」で置き換えたときに $(K, +, *)$ を「**(可換) 環**」と呼びます. たとえば, $(\mathbb{Z}, +, *)$ は可換環になります.

線形空間 (ベクトル空間): n-次元 Euclid 空間 $\mathbb{R}_n = \{(x_1, \ldots, x_n) | x_i \in \mathbb{R}\}$ を一般化したものです. (体 K 上の) 線形空間 L は演算子 "+" があって $(L, +)$ は可換群になります. さらに線形空間 L には「**係数体**」, あるいは「**基礎体**」と呼ばれる体 K による作用 "$*$" があります. そして $a,\ b \in L$ と $\alpha,\ \beta \in K$ に対して次の性質を満します:

線形空間の公理

1.	$(K,+,*)$ は体, $(L,+)$ は可換群	
2.	$\alpha*(\beta*a)=(\alpha*\beta)*a$	(K の積との関係)
3.	$\alpha*(a+b)=\alpha*a+\alpha*b$	(分配律 I)
4.	$(\alpha+\beta)*a=\alpha*a+\beta*a$	(分配律 II)
5.	$1*a=a$	(K の単元 1 との積)

ここで公理 1. にて K が体ではなく可換体の場合, 空間 L のことを「**K-代数 (K-algebra)**」と呼びます. また空間 L の元を「**ベクトル (vector)**」, 係数体 K の元を「**スカラー (scala)**」と呼びます.

ノルムについて:　n-次元 Euclid 空間 $\mathbb{R}^n$ の長さの概念を, 係数体を実数 $\mathbb{R}$ や複素数 $\mathbb{C}$ とする線形空間 L 上に一般化した概念が「**ノルム (norm)**」です. 具体的には任意の $\alpha\in K$ と $x,y\in L$ に対して写像 $\|\ \|: L\to\mathbb{R}_+$ が次の条件を満すときに写像 $\|\ \|$ をノルムと呼びます:

ノルムの公理

1.	$\|x\|\geq 0$	(正値性)
2.	$\|x\|=0\Leftrightarrow x=0$	
3.	$\|\alpha x\|=\lvert\alpha\rvert\,\|x\|$	(斉次性)
4.	$\|x+y\|\leq\|x\|+\|y\|$	(三角不等式)

ここで $\mathbb{R}_+$ は $\{x\in\mathbb{R}\mid x\geq 0\}$ とし, 写像 $|\ |$ は複素数 $\mathbb{C}$ の絶対値を与える函数

$$\begin{array}{cccc} |\ | & : & \mathbb{C} & \to & \mathbb{R}_+ \\ & & \cup & & \cup \\ & & a+ib & \mapsto & \sqrt{a^2+b^2} \end{array}$$

とします.

ノルムの例としては, 実数に対する絶対値 “$|\ |$” や, 連続函数に対する「**p-ノルム**」:“$\|\ \|_p$” が挙げられます. ここで $\|\ \|_p$ は整数 $p\geq 1$ に対し, その絶対値の p 乗の積分[1]

$$\| f \|_p \stackrel{def}{=} \int_{\mathbb{R}^n} |f(x)|^p dx$$

で定義されるノルムです. そして, 集合 A から集合 B への函数で p-ノルムが有界となる函数の集合を $\mathcal{L}_p(A,B)$ と表記します.

[1] この本で積分は Lebesgue 積分で考えています. Lebesgue 積分を使うことで, 通常の Riemann 積分で難のある函数も上手く扱えるようになります.

記号の導入: ここでは以降用いる幾つかの記号を導入しておきます.
まず $x \in \mathbb{R}^n$ は $x = (x_1, \ldots, x_n)$ とします. そして, $x \in \mathbb{R}^n$ の絶対値 $|x|$ を $\sqrt{x_1^2 + \cdots + x_n^2}$,
また $\langle x \rangle$ を $\sqrt{1 + |x|^2}$ で定めます.
次に多項式の冪や微分で用いる「**多重指標**」$\alpha \in \mathbb{N}^n$ を $(\alpha_1, \ldots, \alpha_n)$ とします. このとき $x \in \mathbb{R}^n$ の冪 x^α を $x_1^{\alpha_1} \cdots x_n^{\alpha_n}$, 微分も同様に

$$\partial_x^\alpha = \partial_{x_1}^{\alpha_1} \cdots \partial_{x_n}^{\alpha_n} \qquad \text{ここで } \partial_{x_i} = \frac{\partial}{\partial x_i}$$
$$D_x^\beta = D_{x_1}^{\beta_1} \cdots D_{x_n}^{\beta_n} \qquad \text{ここで } D_{x_i} = \frac{1}{\sqrt{-1}} \frac{\partial}{\partial x_i}$$

とします. また函数 $f : \mathbb{R}^n \to \mathbb{C}$ に対し, その積分を簡単に

$$\int_{I_1 \times \cdots \times I_n} f(x) dx$$

と記述しますが, この意味は n 重積分です:

$$\underbrace{\int_{I_n} \cdots \int_{I_1}}_{n} f(x_1, \ldots, x_n) \underbrace{dx_1 \cdots dx_n}_{n}$$

最後に函数 $f : \mathbb{R}^n \to \mathbb{C}$ に対し, $f^\vee$ を $f^\vee(x) \stackrel{def}{=} f(-x)$ とします.

開集合と閉集合: 点 $a \in \mathbb{R}^n$ を中心とする半径 δ の n-次元球 U_δ を $U_\delta(a) \stackrel{def}{=} \{x \in A; |x - a| < \delta\}$ で定めます. ここで集合 $A \subset \mathbb{R}^n$ が「**開集合**」であるとは, A の任意の点 a に対して $U_\delta(a) \subset A$ となる適当な正数 $\delta > 0$ が存在する場合, つまり $A = \bigcup_{a \in A} U(\delta_a)$ となることです. 次に $B \subset \mathbb{R}^n$ が「**閉集合**」であるとは, B の補集合 $B^c \stackrel{def}{=} R^n - A$ が開集合となる場合です. そして, 集合 $A \subset \mathbb{R}^n$ の「**閉包**」$\overline{A}$ は包含関係 "$\subset$" で A を包含する最小の閉集合として定義します. このとき, 函数 f の「**台 (support)**」を函数 f が零にならない点の集合の閉包として定めます:

$$\mathrm{supp} f \stackrel{def}{=} \overline{\{x \in \mathbb{R}^n; f(x) \neq 0\}}$$

合成積 (convolution): $\mathcal{L}_p(\mathbb{R}^n, \mathbb{C})$ の元 f, g に対する「**合成積**」, あるいは「**畳込**」を次で定めます:

$$(f * g)(x) \stackrel{def}{=} \int_{\mathbb{R}} g(x - y) f(y) dy$$

この合成積は次の性質を満します:

合成積の性質

1)	$f * g = g * f$
2)	$(f * g) * h = f * (g * h)$
3)	$f * \delta = \delta * f = f$
4)	$D^\alpha(f * g) = (D^\alpha f) * g = f * (D^\alpha g)$
5)	$\mathrm{supp}(f * g) \subset \mathrm{supp} f + \mathrm{supp} g$

まず 1) から合成積 “$*$” は可換, 2) から結合律を満す演算子であり, 3) からは単位元 δ が存在して, $(\mathcal{L}_p(\mathbb{R}^n, \mathbb{C}), *)$ が可換半群の構造を持つことが判ります．ここでの δ は「**Dirac の δ 函数**」と呼ばれ,

$$\int_{-\infty}^{\infty} \delta(x)\varphi(x)dx = \varphi(0)$$

を充します．この Dirac の δ 函数は, たとえば, $n = 1$ の場合,

$$d_\varepsilon(x) = \begin{cases} \dfrac{1}{\varepsilon} & 0 < x < \varepsilon \\ 0 & \text{その他} \end{cases}$$

で定義した函数 d_ε の $\varepsilon \to 0$ の極限として定められます．ただし, $x = 0$ で ∞, 他は 0 となるので厳密な意味の函数ではありませんが, 物理や工学ではよく用いられています．この δ 函数は「**Schwartz の超函数 (distribution)**」や「**佐藤の超函数 (hyperfunction)**」によって厳密に定義できます．特に佐藤の超函数の理論によると 1 次元の Dirac の δ 函数は函数 $f(z) = 1/(2\pi iz)$ の境界 $\lim_{y \to 0} (f(x + iy) + f(x - iy))$ として表現されます．このことを判り易く紹介している文献として「**デジタル信号と超関数**」[11] を挙げておきます．そして, 4) は微分との関係を示し, 最後の 5) は函数の台がどうなるかを示しています．ここで $\mathbb{R}^n$ に含まれる集合 A, B の演算 “$+$” を $A + B \stackrel{def}{=} \{x + y; x \in A, y \in B\}$ で定めています．

急減少函数, 緩増加函数と試料函数:　Fourier 変換が最も有効に使える急減少函数について解説しておきます．まず $\mathbb{R}^n$ から $\mathbb{C}$ への無限回可微分函数の集合を $\mathcal{E}$ と表記します．また函数 $f \in \mathcal{E}$ が任意の α, $\beta \in \mathbb{N}^n$ に対して $C_\alpha > 0$ が存在して $|D_x^\beta f(x)| < C_\alpha$ を満すときに函数 f を無限回可微分有界函数と呼び, その集合を $\mathcal{B}$ と表記します．
函数 $f \in \mathcal{E}$ が任意の $\alpha \in \mathbb{N}^n$ に対して $\lim_{|x| \to \infty} |x^\beta D_x^\alpha f(x)| = 0$ を満すときに函数 f を「**急減少函数**」と呼び, その集合を $\mathcal{S}$ と表記します．この集合 $\mathcal{S}$ は「**Schwartz 空間**」とも呼ばれて線形空間としての構造を持ちます．

函数 $g \in \mathcal{E}$ が「**緩増加函数**」と呼ばれるのは函数 f が無限回微分可能な連続函数で, 任意の $\alpha \in \mathbb{N}^n$ に対して $|\partial_x^\alpha f(x)| < C_\alpha \langle x \rangle^{m_\alpha}$ を満たす $C_\alpha > 0$ と $m_\alpha \in \mathbb{N}^n$ が存在するときです. この緩増加函数の集合を $\mathcal{M}$ と表記します. 緩増加函数は絶対値を多項式で抑えることができる函数で, 急減少函数との積は急減少函数になるという性質を持ちます.
函数 $f : \mathbb{R}^n \to \mathbb{C}$ が「**緩増加連続函数**」であるとは, 緩増加函数の無限回微分可能を外して弱めたもの, つまり, 函数 f が連続函数で $|f(x)| < C\langle x \rangle^m$ を満す $C > 0$ と $m \in \mathbb{N}^n$ が存在する場合です. この緩増加連続函数の集合を $\mathcal{M}_0$ と表記します.
試料函数は有限の領域のみで零にならない急減少函数のことで, その集合を $\mathcal{D}$ と表記します. ここで試料函数の有名な例として次の函数を挙げておきます:

$$\phi(x) = \begin{cases} \exp\left(1 - \frac{1}{1-|x|^2}\right) & |x| < 1 \\ 0 & |x| \geq 1 \end{cases}$$

この函数はパラコンパクト空間における「**1 の分割**」で用いられる函数なので頭の片隅に置いておくとよいでしょう[2]. この試料函数は超函数 (=Schwartz の distribution) を定義する上で重要です.
このときに試料函数 $\mathcal{D}$, 急減少函数 $\mathcal{S}$, 緩増加函数 $\mathcal{M}$, 無限回可微分函数 $\mathcal{E}$ と無限回可微分有界函数 $\mathcal{B}$ との間には, 包含関係 $\mathcal{D} \subset \mathcal{S} \subset \mathcal{B} \subset \mathcal{M} \subset \mathcal{E}$ が成立します.
なお, これらの函数の集合で定義域の次元を明確にする必要があるときは $\mathcal{E}(\mathbb{R}^n)$, $\mathcal{S}(\mathbb{R}^n)$, は $\mathcal{M}(\mathbb{R}^n)$, $\mathcal{M}_0(\mathbb{R}^n)$, $\mathcal{D}(\mathbb{R}^n)$ と表記します.

Fourier 級数: 函数 $f \in \mathcal{L}_2(\mathbb{R}, \mathbb{C})$ を周期 T の周期函数 (i.e., 任意の $x \in \mathbb{R}$ に対して $T > 0$ が存在し, $f(x+T) = f(x)$ が成立) とします.

[2] 3.6.3 の mergef 函数の説明でグラフを描いています.

ここで

$$A_k = \frac{1}{T}\int_{-T/2}^{T/2} f(y)\cos\frac{2\pi ky}{T}dy$$
$$B_k = \frac{1}{T}\int_{-T/2}^{T/2} f(y)\sin\frac{2\pi ky}{T}dy \tag{5.1}$$

とすると函数 f が級数として表現されることが知られています:

$$f(x) = \frac{1}{T}\int_{-1/T}^{1/T} f(y)dy + \sum_{k=1}^{\infty}\left(A_k\cos\frac{2\pi kx}{T} + B_k\sin\frac{2\pi kx}{T}\right) \tag{5.2}$$

この級数表現 (5.2) は函数 f の「**(古典的)Fourier 級数**」と呼ばれ, 周期函数が cos 函数と sin 函数の和に分解されることを意味します. ここで周期 T を無限大にしたとき, つまり, 一般の函数に拡張するとどうなるでしょうか？ 函数 f が $\mathcal{L}_2(\mathbb{R},\mathbb{R})$ の元であれば, その積分が有界となるために級数表現 (5.2) の第 1 項は 0 に収束して総和の個所のみが残ります.
ここで

$$A(\xi) = \int_{-\infty}^{\infty} f(X)\cos y\xi dy$$
$$B(\xi) = \int_{-\infty}^{\infty} f(X)\sin y\xi dy \tag{5.3}$$

とすると

$$f(x) = \frac{1}{\pi}\int_{-\infty}^{\infty}\left(A(\xi)\cos x\xi + B(\xi)\sin x\xi\right)d\xi \tag{5.4}$$

を函数 f の「**(古典的)Fourier 積分表示**」と呼びます. さらに「**cos 函数の和公式**」: $\cos(y\xi - x\xi) = \cos y\xi\cos x\xi + \sin y\xi\sin x\xi$ を使って式 (5.4) を書き換えると

$$\begin{aligned} f(x) &= \frac{1}{\pi}\int_{0}^{\infty}\int_{-\infty}^{\infty} f(y)\cos(y\xi - x\xi)\,dyd\xi \\ &= \frac{1}{2\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(y)\cos(y\xi - x\xi)\,dyd\xi \end{aligned} \tag{5.5}$$

ここで $f(y)\sin(y\xi - x\xi)$ が y の奇函数となることから 0 に等しい

$$\frac{i}{2\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(y)\sin\left(y\xi - x\xi\right) dyd\xi$$

を式 (5.5) に加えて「**Euler の公式**」: $e^{i\theta} = \cos\theta + i\sin\theta$ で纏めると

$$f(x) \quad = \quad \frac{1}{2\pi}\int_{-\infty}^{\infty}\left(\int_{-\infty}^{\infty} f(y)e^{-i\xi y}dy\right)e^{i\xi x}d\xi \tag{5.6}$$

が Fourier 積分表示 (5.4) 得られます．さて, この公式 (5.6) には二つの重要な変換が出現しています．1 つは函数 f の「**Fourier 変換**」

$$F(\xi) = \int_{-\infty}^{\infty} f(y)e^{-i\xi y}dy$$

もう 1 つは函数 F の「**逆 Fourier 変換**」

$$f(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\xi)e^{i\xi x}d\xi$$

です．そして, この式 (5.6 自体は, これらの二つの変換式の合成から元の函数が得られるという「**Fourier の反転公式**」と呼ばれる性質になります.
以上 $\mathcal{L}_2(\mathbb{R},\mathbb{C})$ 上で Fourier 変換とその逆変換を強引に導出しましたが, これらの変換が自然に定義できる急減少函数の空間 $\mathcal{S}(\mathbb{R}^n)$ 上でこれらの変換の定義と性質について述べることにします.

Fourier 変換: 写像 $\mathcal{F}:\mathcal{S}\to\mathcal{S}$ を次で定めます:

$$\mathcal{F}\left[f\right](\xi) \overset{def}{=} \int_{\mathbb{R}^n} f(x)e^{-ix\cdot\xi}dx \tag{5.7}$$

この写像 $\mathcal{F}$ を「**Fourier 変換**」と呼び, 函数 $f\in\mathcal{S}$ の $\mathcal{F}$ による像 $\mathcal{F}\left[f\right]$ を $\hat{f}$ と表記します．なお, 函数 $f(x)$ の変数 x が時間を表現しているときに変数 x が取り得る領域を「**時間領域 (Time domain)**」と呼びます．この場合は函数 $\hat{f}$ の変数 ξ がちょうど周波数に対応するので, 変数 ξ の取り得る領域を「**周波数領域 (Frequency domain)**」と呼びます.

逆 Fourier 変換: 写像 $\mathcal{F}^{-1}:\mathcal{S}\to\mathcal{S}$ を次で定めます:

$$\mathcal{F}^{-1}\left[g\right](x) \overset{def}{=} \frac{1}{(2\pi)^n}\int_{\mathbb{R}^n} g(\xi)e^{ix\cdot\xi}d\xi \tag{5.8}$$

この写像 $\mathcal{F}^{-1}$ を「逆 Fourie 変換」と呼びます．ここで定義式 (5.8) を

$$\begin{aligned}\mathcal{F}^{-1}[f](x) &= \frac{1}{(2\pi)^n}\int_{\mathbb{R}^n} g(\xi)e^{ix\cdot\xi}d\xi \\ &= \frac{1}{(2\pi)^n}\int_{\mathbb{R}^n} g(-\xi)e^{-ix\cdot\xi}d\xi\end{aligned}$$

と変形することで, Fourier 変換 $\mathcal{F}$ との間には $\mathcal{F}^{-1}[g] = 1/(2\pi)^n\mathcal{F}[g^\vee]$ の関係があることが判ります．

Fourier の反転公式:　写像 $\mathcal{F}$ と $\mathcal{F}^{-1}$ の間には「**Fourier の反転公式**」と呼ばれる公式があります:

$$\begin{aligned}f &= \mathcal{F}^{-1}\left[\mathcal{F}\left[f\right]\right] \\ g &= \mathcal{F}\left[\mathcal{F}^{-1}\left[g\right]\right]\end{aligned} \tag{5.9}$$

すなわち, $\mathcal{F}^{-1}\mathcal{F}$ と $\mathcal{F}\mathcal{F}^{-1}$ は $\mathcal{S}$ の恒等写像 id で, さらに $\mathcal{F}$ と $\mathcal{F}'$ は $\mathcal{S}$ の同相写像でもあります．

Fourier 変換の性質　Fourier 変換 $\mathcal{F}$ と逆 Fourier 変換 $\mathcal{F}^{-1}$ に関連する性質を列挙しておきます．なお, ここでは $\alpha, \beta \in \mathbb{C}$, $f, g \in \mathcal{S}$, そして, $n \in \mathbb{N}$ とします．

- 線形性:

$$\begin{aligned}\mathcal{F}\left[\alpha\ f + \beta\ g\right](\xi) &= \alpha\ \mathcal{F}[f](\xi) + \beta\ \mathcal{F}[g](\xi) \\ \mathcal{F}^{-1}\left[\alpha\ f + \beta\ g\right](x) &= \alpha\ \mathcal{F}^{-1}[f](x) + \beta\ \mathcal{F}[g](x)\end{aligned} \tag{5.10}$$

- 微分との関係:

$$\begin{aligned}\mathcal{F}\left[D_x^\alpha f\right](\xi) &= \xi^\alpha\mathcal{F}(\xi) \\ D_x^\alpha\mathcal{F}(\xi) &= \mathcal{F}(g)(\xi)\end{aligned} \tag{5.11}$$

  ここで $g(x) = (-x)^\alpha f(x)$ とします．

- 平行移動:

$$\begin{aligned}\mathcal{F}[g](\xi) &= \mathcal{F}[f](\xi - k) \\ \mathcal{F}[h](\xi) &= e^{-ik\cdot\xi}\mathcal{F}[f](\xi)\end{aligned} \tag{5.12}$$

  ここで $k \in \mathbb{R}^n$, $g(x) = e^{ix\cdot k}f(x)$, $h(x) = f(x-k)$ とします．

- 相似変換との関係:

$$\mathcal{F}[f(tx)](\xi) = t^{-n}\hat{f}(\xi/t) \tag{5.13}$$

- Parseval の公式:

$$\int f(x)\overline{g(x)}dx = (2\pi)^{-n}\int \mathcal{F}[f](\xi)\overline{\mathcal{F}[g](\xi)}d\xi \tag{5.14}$$

- 合成積との関係:

$$\begin{aligned} &\mathcal{F}\left[f * g\right](\xi) = \hat{f}(\xi)\hat{g}(\xi) \\ &\mathcal{F}^{-1}\left[\hat{f}\ \hat{g}\right](x) = (f * g)(x) \end{aligned} \tag{5.15}$$

ここで微分との性質では x_i の微分が ξ_i の冪で置換えられること, すなわち, 微分式が多項式に変換されることを意味し, 微分方程式が代数的処理によって解ける可能性を示唆します.

超函数による Fourier 変換の拡大　ここでは Fourier 変換を都合の良い急減少函数の集合 $\mathcal{S}$ 上で定めています. この場合は減衰するような函数であれば急減少函数になるかもしれませんが, cos や sin といった周期函数は扱えません. そこでより一般的な函数に Fourier 変換を拡張するために Schwartz の超函数を利用します. ここで超函数は内積 $\langle , \rangle$ による急減少函数の双対空間として現われ, 超函数を利用することで通常の函数ではないものに対して, Fourier 変換だけではなく, その微分さえも定義することができるようになります.
まず, 考え方を簡単に解説しておきましょう. 二つの函数 $f, g : \mathbb{R}^n \to \mathbb{C}$ が与えられて

$$\int_{\mathbb{R}^n} f(x)g(x)dx$$

が定義できたとします. このときに定まる複素数を $\langle f, g\rangle$ と表記します. さて, ここで $I_f(x) \overset{def}{=} \langle f, x\rangle$ とすると, I_f は函数 f で決定され, 函数を定義域, 値域を複素数とする函数になります. このような函数の函数のことを「**汎函数**」と呼びます. さて, この汎函数は双線型です. 実際, 積分の性質から,

$$\begin{aligned} I_f(\alpha g + \beta h) &= \alpha I_f(g) + \beta I_f(h) \\ I_{\alpha f + \beta h}(g) &= \alpha I_f(g) + \beta I_h(g) \end{aligned}$$

を満します. そこで函数 g を集合 $\mathcal{A}$ の元とするとき, 汎函数 $I_f : \mathcal{A} \to \mathbb{C}$ の集合を $\mathcal{A}'$ と表記し, $\mathcal{A}$ の「**双対**」と呼びます. ここで集合 $\mathcal{A}$ がベクトル空間であれば, その双対 $\mathcal{A}'$ も汎函数の性質からベクトル空間になるので「**双対空間**」と呼びます.
ここで $f \in \mathcal{M}_0$, $\varphi \in \mathcal{S}$ に対し

$$\langle f, \varphi\rangle \overset{def}{=} \int_{\mathbb{R}^n} f(x)\varphi(x)dx \tag{5.16}$$

とします．このとき, $\langle f, \ \rangle : \mathcal{S} \to \mathbb{C}$ は汎函数となり, この汎函数から $\mathcal{S}$ の双対空間 $\mathcal{S}'$ が得られます．この双対空間 $\mathcal{S}'$ の元を「**緩増加超函数**」と呼び, 汎函数 $\langle f, \ \rangle$ を簡単に f と表記します．さて, 定数函数 0 の超函数は任意の $\varphi \in \mathcal{S}$ に対して $\langle 0, \varphi \rangle = 0$ となるので, 任意の写像を 0 に写す汎函数です．このような函数は双対空間 $\mathcal{S}'$ における 0 の働きをするので $\langle 0, \ \rangle$ と表記するよりも 0 と表記しても良いでしょう．次に函数 f と函数 g の超函数が一致する場合はどうなるでしょうか？このときは任意の $\varphi \in \mathcal{S}$ に対して $\langle f - g, \varphi \rangle = 0$ となるので, Lebesgue 積分であれば「**殆ど至る所**」[3]で $f = g$ となることが判ります．つまり, 函数 f に対して超函数が一意に定まることを意味します．また超函数への要請として, 函数 f に収束する函数の列 $\{f_j\}$ が与えられたとき, それらの超函数の列も収束するべきです．そこで超函数の列 $\{f_j\} \in \mathcal{S}'$ に対して

$$f_j \to f \quad \Leftrightarrow \quad \langle f_j, \varphi \rangle \to \langle f, \varphi \rangle \quad , \quad \forall \varphi \in \mathcal{S}$$

とします．また $f \in \mathcal{M}_0$ であれば超函数 f が $\mathcal{S}'$ の元となり, f が Lebesgue 可積分で殆んど至る所で $|f(x)| < C(1 + |x|)^l$ を満す $C > 0$ と正整数 l が存在する場合も $f \in \mathcal{S}'$ となることが知られています ([8] 参照).

さて, 急減少函数の双対としての超函数の微分について解説しましょう．ここで話を簡単にするために $f \in \mathcal{S}'(\mathbb{R})$, $\varphi \in \mathcal{S}(\mathbb{R})$ とします．ここで函数の積 $f(x)\varphi(x)$ に対して形式的に「**Leibniz 則**」:

$$\frac{d}{dx}(f(x)\varphi(x)) = (\frac{d}{dx}f(x))\varphi(x) + f(x)(\frac{d}{dx}\varphi(x))$$

を適用した上で形式的に積分を適用すると

$$[f(x)\varphi(x)]_{-\infty}^{\infty} = \int_{\mathbb{R}} (\frac{d}{dx}f(x))\varphi(x)dx + \int_{\mathbb{R}} f(x)(\frac{d}{dx}\varphi(x))dx$$

が得られます．ここで $f\varphi \in \mathcal{S}(\mathbb{R})$ より, 左辺は 0 になるので

$$\int_{\mathbb{R}} (\frac{d}{dx}f(x))\varphi(x)dx = -\int_{\mathbb{R}} f(x)(\frac{d}{dx}\varphi(x))dx$$

を得ます．この式は形式的な微分 $df(x)/dx$ を含む式の積分が, 実際に存在する函数の積分として表現されることを主張しています．そこで, この式が函数 f の微分を与えていると考えて一般化し, 超函数 $f \in \mathcal{S}'$ の微分を急減少函数 $\varphi \in \mathcal{S}$ を使って次で与えます:

$$\langle D_x^\beta f, \phi \rangle \stackrel{def}{=} \langle f, (-1)^{|\beta|} D_x^\beta \varphi \rangle \tag{5.17}$$

具体的な例として Heaviside 函数の微分を計算してみましょう．

[3] 例外的な点の集合の測度 (=集合の大きさ) が 0 となる, すなわち, 無視できることを意味します．

ここで「**Heaviside 函数**」は

$$h(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

で定義される $x = 0$ を不連続点とする段差のある函数です. $\varphi \in \mathcal{S}$ に対し

$$\begin{aligned}
\langle \partial_x h, \varphi \rangle &= -\langle h, \partial_x \varphi \rangle \\
&= -\int_0^\infty \partial_x \varphi(x) dx \\
&= \varphi(0) \qquad\qquad \because \varphi \in \mathcal{S} \to \varphi(\infty) = 0
\end{aligned}$$

したがって, Heaviside 函数 h の微分は Dirac の δ 函数であることが判ります.
超函数の Fourier 変換も微分と同様の手法で定義します. このときに用いるのが Parseval の公式 (5.14) で, この公式を

$$\langle f, \overline{g} \rangle = \frac{1}{(2\pi)^n} \langle \mathcal{F}[f], \overline{\mathcal{F}[g]} \rangle$$

と書換えても端数の $1/(2\pi)^n$ や函数の共役もあって美しくありませんね.
そこで "(, $)_x$" と "(, $)_\xi$" を

$$\begin{aligned}
(f, g)_x &\overset{def}{=} \langle f, \overline{g} \rangle_x \\
(\mathcal{F}[f], g)_\xi &\overset{def}{=} \frac{1}{(2\pi)^n} \langle \mathcal{F}[f], \overline{g} \rangle
\end{aligned} \tag{5.18}$$

で定義して Parseval の公式 (5.14) を書き直すと

$$(f, g)_x = (\mathcal{F}[f], \mathcal{F}[g])_\xi \tag{5.19}$$

と美しくなります. この式 (5.19) を見返すと, $f \in \mathcal{S}', g \in \mathcal{S}$ なので左辺が存在し, 右辺の $\mathcal{F}[g]$ も存在するので, 函数 f の Fourier 変換 $\mathcal{F}[f]$ ちゃんと定められることを主張しています. そこで超函数 $f \in \mathcal{S}'$ の Fourier 変換を次で定義しましょう:

$$(\mathcal{F}[f], \mathcal{F}[g])_\xi \overset{def}{=} (f, g)_x \tag{5.20}$$

ここで $\mathcal{F} : \mathcal{S} \to \mathcal{S}$ なので, $\mathcal{F}[g]$ を φ と置くと, g は $\mathcal{F}^{-1}[\varphi] = 1/(2\pi)^n \mathcal{F}[\varphi^\vee]$ となり, さらに函数 f の双対 I_f を使えば

$$\mathcal{F}[I_f](\varphi) \overset{def}{=} I_f(\mathcal{F}[\varphi^\vee]) \tag{5.21}$$

とより定義らしく書き換えられます.
なお, Fourier 変換を超函数に拡大しても Fourier 変換の持つ性質 ((5.10) - (5.15)) は保たれます.

5.7.2 Fourier 変換の計算例

Dirac の δ 函数の場合: δ を Dirac の δ 函数, $\varphi \in \mathcal{S}$ とします.

$$
\begin{array}{rcll}
(\mathcal{F}[\delta], \mathcal{F}[\varphi])_\xi & = & \overline{(\delta, \varphi)_x} & \text{定義 (5.20).} \\
& = & \overline{\varphi(0)} & \text{Dirac の}\delta\text{函数の性質.} \\
& = & \dfrac{1}{2\pi}\displaystyle\int_{\mathbb{R}} e^{i\cdot 0\cdot \xi}\overline{\mathcal{F}[\varphi](\xi)}d\xi & \text{Fourier の反転公式 (5.6)} \\
& = & (1, \mathcal{F}[\varphi])_\xi &
\end{array}
$$

以上から $\mathcal{F}[\delta] = 1$ であることが判ります.

定数 1 の場合: Dirac の δ 函数と同様の処理を行います:

$$
\begin{array}{rcl}
(\mathcal{F}[1], \mathcal{F}[\varphi])_\xi & = & (1, \varphi)_x \\
& = & \overline{(e^{-i\cdot 0\cdot x}, \varphi)_x} \\
& = & \overline{\mathcal{F}[\varphi](0)}
\end{array}
$$

ここで $\overline{\mathcal{F}[\varphi](0)} = (2\pi\delta, \mathcal{F}[\varphi])_\xi$ より $\mathcal{F}[1] = 2\pi\delta$ を得ます.

$e^{i\alpha x}$ の場合: この函数は周期函数でも急減少函数でもありませんが $\varphi \in \mathcal{S}$ に対して

$$\int_{\mathbb{R}^n} e^{i\alpha x}\varphi(x)dx = \mathcal{F}^{-1}[\varphi](\alpha)$$

となるので $e^{i\alpha x} \in \mathcal{S}'$, このことから Fourier 変換も δ 函数と同様に

$$
\begin{array}{rcl}
(e^{\alpha x}, \varphi)_x & = & \displaystyle\int_{\mathbb{R}} e^{i\alpha x}\overline{\varphi(x)}dx \\
& = & \overline{\mathcal{F}[\varphi](-\alpha)} \\
& = & (\delta(\xi+\alpha), \mathcal{F}[\varphi](\xi))_\xi
\end{array}
$$

以上から $\mathcal{F}[e^{i\alpha x}](\xi) = \delta(\xi + \alpha)$ を得ます. ただし, 指数函数 e^x は $\mathcal{S}'$ の元にはなりません. 実際, 急減少函数 $e^{-\langle x\rangle/2}$ との積 $e^{x-\langle x\rangle/2}$ の積分が発散するからです ([8], 二章の問題 4 参照).

cos 函数と sin 函数の場合: cos と sin が指数函数を用いた表現

$$\cos\alpha x = \frac{e^{i\alpha x} + e^{-i\alpha x}}{2} \quad , \quad \sin\alpha x = \frac{e^{i\alpha x} - e^{-i\alpha x}}{2i}$$

と Fourier 変換の線形性 (5.10) と平行移動の性質 (5.12) から容易に計算できます:

$$\mathcal{F}[\cos \alpha x](\xi) = \frac{\delta(\xi+\alpha)+\delta(\xi-\alpha)}{2}$$

$$\mathcal{F}[\sin \alpha x](\xi) = \frac{\delta(\xi+\alpha)-\delta(\xi-\alpha)}{2i}$$

ここで α は周期の 2π 倍なので, この周期の $\pm 2\pi$ 倍の個所で δ 函数によるピークが出ることが判ります.

5.7.3 離散的 Fourier 変換 (DFT) について

さて, 以上では急減少函数 $\mathcal{S}$ とその双対 $\mathcal{S}'$ に対して Fourier 変換が定義できました. これを離散的な数列に対しても適用できます. たとえば, 1 次元配列の場合, n 成分の数列 $\{x_k\}_{k=1,\ldots,n}$ に対して離散的 Fourier 変換を次で定めます:

$$X_k \stackrel{def}{=} \sum_{m=1}^{n} x_m \exp\left(-i2\pi\frac{(k-1)(m-1)}{n}\right) \tag{5.22}$$

なお, 離散的逆 Fourier 変換は次で定られます:

$$x_m \stackrel{def}{=} \frac{1}{n}\sum_{k=1}^{n} X_k \exp\left(i2\pi\frac{(k-1)(m-1)}{n}\right) \tag{5.23}$$

Yorick の fft 函数で計算されるのは離散的 Fourier 変換になります.

5.7.4 fft.i ライブラリに含まれる函数

FFT の関連する函数は “fft.i” ライブラリで定義されていますが, 実際の計算は netlib.org の fftpack 由来の函数で処理しています[4]. ソースファイルを入手していれば “fft” ディレクトリにソースファイルが収録されています.

fft.i ライブラリの主要な函数

最初に高速 Fourier 変換を実行する函数を纏めておきましょう:

[4]README によると, fftpack を f2c で FORTRAN から C に変換したものを利用.

高速 Fourier 変換を行う函数

構文 (fft, fft_inplace)
fft(⟨ 配列 ⟩, ⟨ 方向配列 ⟩)
fft(⟨ 配列 ⟩, ⟨ 左方向配列 ⟩, ⟨ 右方向配列 ⟩)
fft(⟨ 配列 ⟩, ⟨ 左方向配列 ⟩, ⟨ 右方向配列 ⟩, setup =⟨workspace⟩)
fft_inplace,⟨ 配列 ⟩, ⟨ 方向配列 ⟩
fft_inplace,⟨ 配列 ⟩, ⟨ 左方向配列 ⟩, ⟨ 右方向配列 ⟩
fft_inplace,⟨ 配列 ⟩, ⟨ 左方向配列 ⟩⟨ 右方向配列 ⟩, setup =⟨workspace⟩

fft 函数:　第 1 引数に対する高速 Fourier 変換を計算する函数で, 一部が Yorick 言語で記述されています. この fft 函数内部では第 1 引数を complex 函数で複素数に変換したのちに, fft_inplace 函数 で処理を行います. 具体的には数列 $x_{k=1}^{n}$ に対し

$$X_m \stackrel{def}{=} \sum_{k=1}^{n} x_k \exp\left(-i2\pi\sigma\frac{(k-1)(m-1)}{n}\right) \tag{5.24}$$

を計算します. ここで式 5.24 の σ は「変換の向きの指定」で整数値 0, 1, -1 の何れかの値を取り, 1 の場合を順変換, -1 の場合を逆変換と呼び, 0 の場合は無変換になります. この指定は 1 次元配列で第 2 引数以降で行います. この σ の指定は高次元配列に対しても可能で, たとえば方向配列として [1,-1,0] を指定した場合, 第 1 引数として与えられた配列の第 1 添字に対しては順変換, 第 2 添字に対しては逆変換, 第 3 添字に対しては無変換という指定になります. つまり, $\{x\}$ と同じ大きさの配列 $\{X\}$ の (s,t,u) 成分は

$$\sum_{k,j,m=1}^{n_k,n_l,n_m} x_{k,l,m} \exp\left(-i2\pi\left(\frac{(k-1)(s-1)}{n_k} + (-1)\frac{(l-1)(t-1)}{n_l} + 0\cdot\frac{(m-1)(u-1)}{n_m}\right)\right)$$

で計算されます. なお, 順変換が通常の離散 Frourier 変換 (DFT) に相当しますが逆退変換は離散逆 Fourier 変換にはなりません. 何故なら通常の逆 Fourier 変換は式 (5.23) で定義されますが fft 函数の逆変換は単に指数を正負を逆の負にしただけの式 (5.22) が用いられて配列の総数で割るという正規化が行われていません. つまり, $1/n$ の項がないために n 倍になってしまいます. 実際に sin 函数から得られた数列の FFT を計算させてみましょう:

逆変換と順変換を行うことで前よりも 1001 倍の大きさですが正弦波が復元されています. また順変換のみ, あるいは逆変換のみを 2 度行うと振幅が 1001 倍の逆向きの正弦波が得られます. このように Yorick の fft 函数を成分が n 個の 1 次元配列に $2k$ 回作用させれば絶対値が n^k 倍の配列が得られます.

fft 函数は高次元配列に対して添字操作を用いることで方向をより簡易に指定することができます. そのために第 2 引数と第 3 引数を使います. 第 2 引数をここでは左方

```
> x=span(0,1,1001)
> y=sin(4*pi*x)
> fy=fft(y,1)
> Y=fft(fy,−1)
> iY=fft(fy,1)
> plg,Y.re,x
> plg,iY,x,color=”red”
> print,Y.re(max),iY.re(max)
1001   1001
```

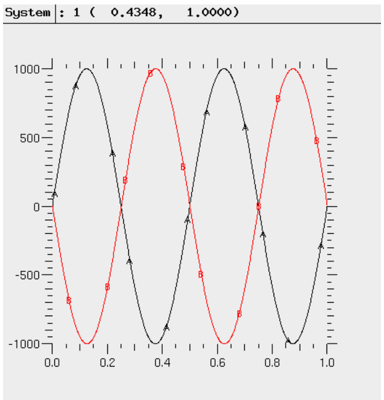


図 5.1: fft(fft(y),-1) と fft(fft(y)) の結果

向配列と呼び, 配列の左端の添字から方向指定を行い, 第 3 引数を同様に右方向配列と呼び, 配列の右端の添字から方向指定を行います. たとえば第 2 引数を [1,-1], 第 3 引数を [-1,0] とした場合, n 次元配列 $x(i_1, i_2, \ldots, i_{n-1}, i_n)$ に対して 1 次添字 (=i_1) が順変換, 2 次添字 (=i_2) が逆変換, $n-1$ 次添字 (=i_{n-1}) が逆変換, n 次添字 (=i_n) が無変換となります. そして, 方向配列として “[ ]” を指定すると, その部分の方向指定は全て 1 に設定されます. たとえば通常の ‘fft(x,f)’ は ‘ftt(x,f,[])’ と同じ意味です.

fft_inplace 函数: fft 函数の実体です. fft 函数との違いは計算結果が pointer を介して第 1 引数の変数に代入される点です.
fft 函数, および fft_inplace 函数にはキーワード setup があります. このキーワード setup で指定されるのは作業領域 (workplace) と呼ばれる配列です. この配列は fft_setup 函数で設定できるもので, fft 函数や fft_inplace 函数で指定していなければ fft_inplace 函数内部で生成する仕様です. この workplace 配列は二つの pointer 型の対象から構成され, 第 1 成分が返却する配列の大きさ, 第 2 成分が fft_raw 函数からの結果を蓄えるようになっています.

高速 Fourier 変換に関連する函数

fft 函数や fft_inpsace 函数を補佐する函数を挙げておきましょう:

高速 Fourier 変換に関連する函数

構文 (fft_setup, fft_init, fft_fraw, fft_braw, roll)
fft_setup(dimsof(⟨ 配列 ⟩))
fft_setup(dimsof(⟨ 配列 ⟩),⟨ 方向配列 ⟩)
fft_setup(dimsof(⟨ 配列 ⟩),$\langle ldir \rangle, \langle rdir \rangle$)
roll(⟨ 配列 ⟩, ⟨ 配列$_{\mathrm{LJOFF}}$⟩, ⟨ 配列$_{\mathrm{RJOFF}}$⟩)
roll,⟨ 配列 ⟩, ⟨ 配列$_{\mathrm{LJOFF}}$⟩, ⟨ 配列$_{\mathrm{RJOFF}}$⟩
roll(⟨ 配列 ⟩)
roll,⟨ 配列 ⟩
_roll2,⟨ 配列$_1$⟩, ⟨ 整数$_1$⟩, ⟨ 整数$_2$⟩, ⟨ 整数$_3$⟩, ⟨ 整数$_4$⟩, ⟨ 配列$_2$⟩

fft_setup 函数:　fft 函数や fft_inplace 函数から呼び出される fft_raw 函数のための作業領域を設定する函数です. この作業領域 (workplace) の実体は二つの pointer 型の対象で構成されたベクトルで, その第 1 成分が処理を行う配列の大きさを表現する配列が格納され, 第 2 成分に格納された配列が fft_raw で用いられます.

```
> x=span(0,1,1001)
> y=[x,x,x]
> test1=fft_setup(dimsof(x))
> test2=fft_setup(dimsof(y))
> print,test1 ,test2
[0x76c050,0x76bfe0]  [0x76c088,0x74df78]
> print,*(test1 (1)),*( test2 (1))
[1001]  [3,1001]
```

roll 函数:　roll 函数は第 1 引数で与えた配列を掻き回す函数です. 引数が 1 つだけの場合, 与えられた配列の各次元で折返しを行います. すなわち, 配列 X を $[X_1, \ldots, X_n]$ と見做せば

$$\mathrm{roll}(X) = [\mathrm{roll}(X_m), \mathrm{roll}(X_{m+1}), \ldots, \mathrm{roll}(X_n), \mathrm{roll}(X_1), \mathrm{roll}(X_2), \ldots, \mathrm{roll}(X_{m-1})]$$

となります. ここで m は n が偶数であれば $n/2$, n が奇数であれば $(n+1)/2$ で与えられる整数です.

```
> roll ([1,2,3,4,5])
 [4,5,1,2,3]
> roll   ([[1,2],[2,3],[3,4],[4,5]])
   [[4,3],[5,4],[2,1],[3,2]]
```

また第 2 引数と第 3 引数の配列はそれぞれ左右のオフセットを指定します.

```
> roll ([1,2,3,4,5],3)
 [3,4,5,1,2]
> roll ([1,2,3,4,5],[])
 [4,5,1,2,3]
> roll ([1,2,3,4,5],[],4)
 [2,3,4,5,1]
```

_roll2 函数: roll 函数が呼出す函数で, サブルーチン roll が実体です. なお, 第 5 引数の ⟨ 配列$_2$⟩ が double 型の fft_setup で構築した作業領域になります.

FFTPAK 由来の函数

"fft.i" ライブラリには FFTPACK 由来の函数を幾つか含んでいます:

FFTPACK 由来の函数

構文 (fft_init, fft_raw,, fft_fraw, fft_braw, _roll2)
fft_init,⟨ 正整数 ⟩,⟨ 配列 ⟩
fft_raw, ⟨ 整数$_1$⟩,⟨ 配列$_1$⟩,⟨ 整数$_2$⟩,⟨ 整数$_3$⟩,⟨ 整数$_4$⟩,⟨ 配列$_2$⟩
fft_fraw, ⟨ 整数$_1$⟩,⟨ 配列$_1$⟩,⟨ 配列$_2$⟩
fft_braw, ⟨ 整数$_1$⟩,⟨ 配列$_1$⟩,⟨ 配列$_2$⟩

fft_init 函数: Swarztrauber の cffti です. fft_setup 函数でのみ用いられる函数で, 第 1 引数の整数を n とすると, 第 2 引数の配列は array(0.0,4*n+15) でなければなりません.

fft_raw 函数: fft_inplace 函数から呼出される函数で, サブルーチン cfft2 が実体です. この cfft2 は内部で方向が 0 以上であれば Fourier 変換を行うサブルーチンの cfftf1, 方向が 0 より小であれば正規化しない逆 Fourier 変換を行うサブルーチンの cfftb1 を呼出します. Yorick 上ではサブルーチン cfftf1 が fft_fraw 函数, サブルーチン cfftb1 が fft_braw 函数に対応します.

fft_fraw 函数: サブルーチン cfftf に対応し, 離散的 Fourier 変換 (5.22) を行う函数です. 第 1 引数の ⟨ 整数 ⟩ は処理するベクトルの長さであり, 第 2 引数の ⟨ 配列$_1$⟩ には complex 型の結果が格納され, 第 3 引数の ⟨ 配列$_2$⟩ が double 型の fft_setup で構築した作業領域になります.

fft_braw 函数:　サブルーチン cfftb に対応し, 正規化しない逆 Fourie 変換を行う函数です．つまり, 通常の離散的逆 Fourier 変換 (5.23) の $1/n$ がない変換です．ここで第 1 引数の ⟨ 整数 ⟩ が処理するベクトルの長さで, 第 2 引数の $\langle$ 配列$_1\rangle$ に complex 型の結果が格納され, 第 3 引数の $\langle$ 配列$_2\rangle$ は double 型の fft_setup 函数で構築した作業領域になります.

5.7.5　convol.i ライブラリに含まれる函数

この函数は "fft.i" とは別の "i" ディレクトリに包含されたライブラリですが Fourier 変換絡みてこちらに記載しておきます．この "convol.i" ライブラリには「**畳込 (convolution)**」に関連する函数が含まれています．このライブラリに収録された函数を利用するためには予め 'include,"convol.i"' を行う必要があります:

convol.i ライブラリの主要な函数

構文 (convol, fft_good)
convol($\langle$ 配列$_1\rangle, \langle$ 配列$_2\rangle, n0 = \langle$ 正整数$_1\rangle, n1 = \langle$ 正整数$_2\rangle$)
fft_good(⟨ 数値 ⟩)

convol 函数:　$\langle$ 配列$_1\rangle$ と $\langle$ 配列$_2\rangle$ を 1 次元配列とするとき, 両者の畳込を計算します．ここで返却される配列の大きさ N_{a*b} は, $\langle$ 配列$_1\rangle$ と $\langle$ 配列$_2\rangle$ の大きさをそれぞれ N_a, N_b としたときに $N_{a*b} = N_a + N_b - 1$ となります．またキーワード n0 と n1 は $1 \leq n0 < n1 \leq N_{a*b}$ の関係があり, 返却する領域指定で用います.

fft_good 函数:　引数として与えられる数値は整数と実数に限定されます．与えられた数値が 7 よりも小であれば, 1 と比較して大きな方が返却値となります．それ以外の場合, $2^x \cdot 3^y \cdot 5^z$ と表記できる整数値で与えられた数値よりも大きなもののなかで最小の数値を返却します.

5.8　matrix.i に含まれる函数

LAPACK 由来の函数を利用して線形 1 次方程式を解くためのライブラリが "matrix.i" です．このライブラリでは LAPACK 由来の函数には先頭に記号 "_" を付けており, それ以外の函数は Yorick 言語で記述されています:

matrix.i ライブラリに含まれる函数

構文 (unit, TDsolve, LUsolve, QRsolve, SVsolve, LUrcond, SVdec)
unit($\langle$ 整数$_1\rangle$, $\langle$ 整数$_2\rangle$)
TDsolve($\langle$ 配列$_1\rangle$, $\langle$ 配列$_2\rangle$, $\langle$ 配列$_3\rangle$, $\langle$ 配列$_4\rangle$,which=)
LUsolve($\langle$ 配列$_1\rangle$, $\langle$ 配列$_2\rangle$)
QRsolve($\langle$ 配列$_1\rangle$, $\langle$ 配列$_2\rangle$)
SVsolve($\langle$ 配列$_1\rangle$, $\langle$ 配列$_2\rangle$, which=)
LUrcond($\langle$ 配列$_1\rangle$, one_norm=)
SVdec($\langle$ 配列$_1\rangle$, $\langle$ 配列$_2\rangle$, $\langle$ 配列$_3\rangle$, full)

unit 函数: $\langle$ 整数$_1\rangle \times \langle$ 整数$_2\rangle$ の大きさの対角成分が 1 となる行列に対応する 2 次元配列を生成する函数です.

TDsolve 函数: 行列 A を帯行列とするとき, 方程式 $AX = B$ を解く函数です.
具体的には $\langle$ 配列$_1\rangle$, $\langle$ 配列$_2\rangle$, $\langle$ 配列$_3\rangle$ と $\langle$ 配列$_4\rangle$ に対応するベクトルをそれぞれ C, D, E, B とするとき, ベクトル D が行列 A の対角成分になります. またベクトル D の成分数を n とするときにベクトル B の成分数は n, ベクトル C,E の成分数が $n-1$ で次の方程式が定義されます:

$$
\begin{pmatrix}
D_1 & E_1 & & & & & & \\
C_1 & D_2 & E_2 & & & & \text{\Large 0} & \\
& C_2 & D_3 & E_3 & & & & \\
& & \ddots & \ddots & \ddots & & & \\
& & & C_{m-1} & D_m & E_m & & \\
& & & & \ddots & \ddots & \ddots & \\
& & \text{\Large 0} & & & C_{n-2} & D_{n-1} & E_{n-2} \\
& & & & & & C_{n-1} & D_n
\end{pmatrix}
\begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_m \\ \vdots \\ X_{n-1} \\ X_n \end{pmatrix}
=
\begin{pmatrix} B_1 \\ B_2 \\ B_3 \\ \vdots \\ B_m \\ \vdots \\ B_{n-1} \\ B_n \end{pmatrix}
$$

なお, TDsolve 函数はキーワード which を持ちます. このキーワードによって次の指定が行われます:

which の指定

方程式	which の値
A(,+)*X(+,..)=B	which=1
A(,+)*X(..,+)=B	which=0

LUsolve 函数:　LU 分解を用いて方程式 $AX = B$ の解を返却する函数です．ここで行列 A は ⟨ 配列$_1$⟩, 行列 B は ⟨ 配列$_2$⟩ に対応します．またキーワードの which は高次元の配列に対し, 方程式を指定するために用いられます:

which の指定

方程式	**which の値**
$A_{(i,j)}X_{(j,k,l,m)} = B_{(i,k,l,m)}$	which=1
$A_{(i,j)}X_{(k,j,l,m)} = B_{(k,i,l,m)}$	which=2
$A_{(i,j)}X_{(k,l,m,j)} = B_{(k,l,m,i)}$	which=4, あるいは which=0

なお, 行列 B に対応する ⟨ 配列$_2$⟩ が省略された場合, ⟨ 配列$_1$⟩ に対応する行列 A の逆行列 A^{-1} を返却します．

QRsolve 函数:　QR 分解を使って, 方程式 $AX = B$ を解く函数です．この QRsolve 函数は LUsolve 函数よりも処理は低速です．ここで行列 A が $m \times n$ の行列とするとき, $m > n$ であれば QR 分解, $m < n$ であれば LR 分解が実行されます．QRsolve 函数は行列 A が特異行列の場合は計算に失敗します．この場合は SVsolve 函数を利用して下さい．キーワードの which は 0 か 1 を指定し, TDsolve 函数と同様の指定が行われます:

which の指定

方程式	**which の値**
A(,+)*X(+,..)=B	which=1
A(,+)*X(..,+)=B	which=0

SVsolve 函数:　方程式 $AX = B$ を解く函数で, 行列 A が特異の場合でも解くことができます．

LUrcond 函数:　LAPACK の dgecon に由来する_dgecox 函数を用いて ⟨ 配列 ⟩ に対応する行列 A の「**条件数の逆数 (reciprocal condition number)**」を返却する函数で, ‘LUrcond(a)+1.0==1.0’ の場合, 配列 a に対応する行列 A は数値的に特異であると判断されます．

SVdec 函数:　LAPACK の dgelx に由来する_dgelx 函数を用いて特異行列の分解を行う函数です．この函数による分解は $m \times n$-特異行列 A に対して $A = (U(,+) * \sigma(+,))(,+) * VT(+,)$ を満す配列を返却します．

5.8.1 LAPACK 由来の函数

“matrix.i” ライブラリで定義されている LAPACK 由来の函数を纏めておきます:

LAPACK 由来の函数

函数	概要
_dgtsv	dgtsv 函数に由来する函数
_dgesv	dgesv 函数に由来する函数
_dgetrf	dgetrf 函数に由来する函数
_dgecox	dgecon 函数に由来する函数
_dgelx	dgels 函数に由来する函数
_dgelss	dgeless 函数に由来する函数
_dgesvx	dgesvx 函数に由来する函数

第6章　主要な数学函数

First Clown	第一の墓掘人
Give me leave. Here lies the water; good: here stands the man; good; if the man go to this water, and drown himself,it is, will he, nill he, he goes-mark you that; but if the water come to him and drown him, he drowns not himself: argal, he that is not guilty of his own death shortens not his own life.	ちょっと待てよ. ここに水があるとせよ; よろしい: ここに人が居るとせよ; よろしい: こいつがこっちの水ん所いって, 溺れちまえば, そりゃ, どうしたところで, てめえで御陀仏って訳さ; が, こっちの水がこいつんところやって来て 御陀仏させりゃ, てめえで御陀仏って訳じゃない: だーから, 甲ハ甲ノ下手人ニアラザルナリ, てめえで御陀仏したんじゃねえ.

Hamlet: 第五幕, 第一場

6.1　i ディレクトリに含まれるライブラリ

ここでは大域変数 Y_SITE で指示されるディレクトリの中の "i" ディレクトリ に収録された主要な函数について概要を解説します. ここで「主要な函数」は "i" ディレクトリの README ファイルに記述されたライブラリに収録された函数のことを指し, その中で数学に関連する函数に限定しています.

ここで紹介する函数は基本的に Yorick 言語で記述された函数であり, その定義は該当する函数が収録されたライブラリを適当なエディタで開くことで確認できます. このライブラリの探し方は, help 函数を用いて函数の解説を表示したときの解説の下段の "defined at:" で開始する行に函数の定義の開始位置とライブラリが表示されています:

```
> help,gcd
 /* DOCUMENT gcd(a,b)
       returns the GCD (greatest common divisor) of A and B, which must
       be one of the integer data types.  A and B may be conformable
       arrays; the semantics of the gcd call are the same as any other
       binary operation.  Uses Euclid's celebrated algorithm.
       The absolute values of A and B are taken before the operation
       commences; if either A or B is 0, the return value will be 0.
     SEE ALSO: lcm, is_prime, factorize
   */
 defined at:  LINE: 11  FILE: /usr/local/yorick-2.1/i/gcd.i
>
```

函数 gcd の解説の例を示していますが, この環境では "`/usr/local/yorick-2.1/i/`" ディレクトリにある "gcd.i" ファイルの 11 行目で gcd 函数が定義されていることが分ります.

6.2　数論に関連する函数

Yorick にも数式処理程ではありませんが数論に関連する函数があります. 浮動小数点数を引数とする場合, 桁数が 15 桁までであれば結果を write 函数を使って丸めない表示にすることができます.

6.2.1　gcd.i ライブラリ

"gcd.i" ライブラリに最大公約数, 最小公倍数や因数分解, および素数判定のための函数が含まれています. なお, $3 \cdot 10^9$ を越える整数 (=32-bit 整数の上限) に対しては素数判定の結果の保証はできません:

gcd.i に含まれる主要な函数

構文	概要
gcd(⟨ 整数$_1$⟩, ⟨ 整数$_2$⟩)	2 つの整数の最大公約数 (GCD) を計算
lcm(⟨ 整数$_1$⟩, ⟨ 整数$_2$⟩)	2 つの整数の最小公倍数 (LCM) を計算
factorize(⟨ 整数 ⟩)	因数分解を 2 次元配列で返却
is_prime(⟨ 整数 ⟩)	素数の述語函数

これらの函数は引数の正負に結果が影響されません．また, factor 函数の引数となる整数を a, 返却する配列を A, n を配列 $A(,1)$ の大きさであれば $a = \sum_{i=1}^{n} A(i,1)^{A(i,2)}$ となります:

```
> A=factorize(168)
> A
  [[2,3,7],[3,1,1]]
> A(,1)^A(,2)
 [8,3,7]
> is_prime(123)
0
> factorize(123)
  [[3,41],[1,1]]
> is_prime(13)
1
> factorize(13)
 [[13],[1]]
```

6.3 特殊函数

特殊函数として, Bessel 函数, dawson 積分, Γ 函数, 楕円函数や Fermi-Dirac 積分が標準で含まれています.

6.3.1 bessel.i ライブラリ

“bessel.i” は Bessel 函数のライブラリです．⟨ 次数 ⟩ は全て complex 型以外の 0 次元の数値配列でなければなりません:

bessel.i の主要な函数

構文	概要
bessj(⟨ 次数 ⟩, ⟨ 配列 ⟩)	第 1 種の Bessel 函数 J_n
bessi(⟨ 次数 ⟩, ⟨ 配列 ⟩)	第 1 種の変形 Bessel 函数 I_n.
bessk(⟨ 次数 ⟩, ⟨ 配列 ⟩)	第 2 種の Bessel 函数 K_n.
bessy(⟨ 次数 ⟩, ⟨ 配列 ⟩)	第 2 種の Bessel 函数 Y_n.
bessy0⟨ 配列 ⟩)	第 2 種の Bessel 函数 Y_0.
bessy1(⟨ 配列 ⟩)	第 2 種の Bessel 函数 Y_1.

6.3.2　dawson.i ライブラリ

“dawson.i” は dawson 積分や erf 函数を含むライブラリです:

dawson.i の主要な函数

構文	概要
dawson(⟨ 配列 ⟩)	Dawson 積分: $\exp(-x^2)\int_0^x \exp(t^2)dt$ を計算.
erf(⟨ 配列 ⟩)	誤差函数:$\frac{2}{\sqrt{\pi}}\int_0^x \exp(-t^2)dt$ を計算.
erfc(⟨ 配列 ⟩)	相補誤差函数:$\frac{2}{\sqrt{\pi}}\int_x^\infty \exp(-t^2)dt$ を計算.

6.3.3　fermi.i ライブラリ

“fermi.i” は完全 Fermi-Dirac 積分函数 $F_j(x)$ を次数 j=-1/2, 1/2, 3/2, 5/2 について計算する函数を含むライブラリです．ここで $F_j(x)$ は次で定義されます:

$$F_j(x) \stackrel{def}{=} \frac{1}{\Gamma(j+1)}\int_0^\infty \frac{t^j}{\exp(t-x)+1}dt$$

‘include,”fermi.i”’ による読込が必要です．このライブラリの概要は `help,fermi` で読むことができます:

fermi.i の主要な函数

構文	概要
fd12(⟨ 配列 ⟩)	次数 $j = 1/2$ の Fermi-Dirac 積分を計算.
fd32(⟨ 配列 ⟩)	次数 $j = 3/2$ の Fermi-Dirac 積分を計算.
fd52(⟨ 配列 ⟩)	次数 $j = 5/2$ の Fermi-Dirac 積分を計算.
fdm12(⟨ 配列 ⟩)	次数 $j = -1/2$ の Fermi-Dirac 積分を計算.
ifd12(⟨ 配列 ⟩)	次数 $j = 1/2$ の逆 Fermi-Dirac 積分を計算.
ifd32(⟨ 配列 ⟩)	次数 $j = 3/2$ の逆 Fermi-Dirac 積分を計算.
ifd52(⟨ 配列 ⟩)	次数 $j = 5/2$ の逆 Fermi-Dirac 積分を計算.
ifdm12(⟨ 配列 ⟩)	次数 $j = -1/2$ の逆 Fermi-Dirac 積分を計算.

6.3.4 fermii.i ライブラリ

不完全 Fermi-Dirac 積分を次数 $j = -1/2, 1/2, 3/2, 5/2$ について計算する函数を含み, “fermi.i” と “dawson.i” を利用するライブラリです. ここで $F_j(x, b)$ は次で定義されます:

$$F_j(x, b) \stackrel{def}{=} \frac{1}{\Gamma(j+1)} \int_b^\infty \frac{t^j}{\exp(t-x)+1} dt$$

‘include,”fermii.i”’ でライブラリを読込む必要がありますが, 読込時に “fermi.i” と “dawson.i” の双方も読込まれます. この “fermii.i” ライブラリの概要は `help,fermii` で読むことができます:

fermii.i の主要な函数

構文	概要
fdi12(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)	次数 $j = 1/2$ の不完全 Fermi-Dirac 積分を計算.
fdi32(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)	次数 $j = 3/2$ の不完全 Fermi-Dirac 積分を計算.
fdi52(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)	次数 $j = 5/2$ の不完全 Fermi-Dirac 積分を計算.
fdm1m2(⟨ 配列 ⟩)	次数 $j = -1/2$ の不完全 Fermi-Dirac 積分を計算.

6.3.5 gamma.i ライブラリ

“gamma.i” は Γ 函数, B 函数や二項係数といった函数が定義されたライブラリです. なお Γ-函数は自然数 $n > 0$ に対して $\Gamma(n) = (n-1)!$ を満します:

gamma.i の主要な函数

構文	概要
lngamma(⟨ 配列 ⟩)	$\log(\Gamma(x))$ を計算.
ln_gamma(⟨ 配列 ⟩)	$\log(\Gamma(x))$ を計算. 内部で lngamma 函数を利用
beta(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)	$\dfrac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$ を計算
bico(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)	$\dfrac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)}$ を計算

beta 函数: Yorick 内部では'beta(z,w)=exp(lngamma(z)+lngamma(w)-lngamma(z+w))'を計算しています.

bico 函数: 二項係数 ${}_nC_k \stackrel{def}{=} n!/k!(n-k)!$ を $\Gamma(n+1)/\Gamma(k+1)\Gamma(n-k+1)$ で計算して double 型で結果を返す函数です. 数値計算上の理由から'floor(0.5+exp(lngamma(n+1)-lngamma(k+1)-lngamma(n-k+1)))' で定義されています.

6.3.6 gammp.i ライブラリ

不完全 Γ 函数に関連するライブラリで, "gamma.i" ライブラリを利用します. ここで第 1 種不完全 Γ 函数 $\gamma(a,x)$ と第 2 種不完全 Γ 函数 $\Gamma(a,x)$ は次で定義される函数です:

$$\gamma(a,x) \stackrel{def}{=} \int_0^x t^{a-1}\exp(-t)dt$$
$$\Gamma(a,x) \stackrel{def}{=} \int_x^\infty t^{a-1}\exp(-t)dt$$

定義から $\Gamma(a) = \gamma(a,x) + \Gamma(a,x)$ を満し, この式の両辺を $\Gamma(a)$ で割って得られた函数をそれぞれ $P(a,x) \stackrel{def}{=} \gamma(a,x)/\Gamma(a)$ と $Q(a,x) \stackrel{def}{=} 1-\gamma(a,x)/\Gamma(a)$ で定義します:

gammp.i の主要な函数

構文	概要
gammp(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)	$P(a,x) \Leftrightarrow \mathrm{gammp}(a,x)$
gammp(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩, ⟨&配列$_3$⟩, ⟨&配列$_4$⟩)	
gammq(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)	$Q(a,x) \Leftrightarrow \mathrm{gammq}(a,x)$
gammq(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩, ⟨&配列$_3$⟩, ⟨&配列$_4$⟩)	
betai(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩, ⟨ 配列$_3$⟩)	$\dfrac{1}{B(a,b)}\displaystyle\int_0^x t^{a-1}(1-t)^{b-1}dt \Leftrightarrow \mathrm{betai}(a,b,x)$

6.3.7 elliptic.i ライブラリ

"elliptic.i" ライブラリには楕円積分に関連する函数が収録されています. このライブラリの概要は`help,elliptic`で読むことができます:

elliptic.i の主要な函数

構文	概要
ell_am(⟨ 配列 ⟩)	Jacobi の楕円函数 am.
ell_am(⟨ 配列 ⟩, ⟨ 母数 ⟩)	Jacobi の楕円函数 am.
dn_(ell_am(⟨ 配列 ⟩, ⟨ 母数 ⟩)	Jacobi の楕円函数 dn.
ell_f(⟨ 配列 ⟩, ⟨ 母数 ⟩)	第 1 種不完全楕円積分函数を計算.
ell_e(⟨ 配列 ⟩, ⟨ 母数 ⟩)	第 2 種不完全楕円積分函数を計算.
ellip_k(⟨ 母数 ⟩)	第 1 種完全楕円積分函数を計算.
ellip_e(⟨ 母数 ⟩)	第 2 種完全楕円積分函数を計算.

ここで ⟨ 母数 ⟩ は complex 型を除く 0 次元の数値配列でなければなりません.

6.3.8 ellipse.i ライブラリ

各種の完全楕円函数を含みます. このライブラリに含まれる函数を利用するためには 'include,"ellipse.i"' でライブラリの読込を行う必要があります:

ellipse.i の主要な函数

構文	概要
EllipticK(⟨ 配列 ⟩)	第 1 種完全楕円函数を計算.
EllipticE(⟨ 配列 ⟩)	第 2 種完全楕円函数を計算

配列の各成分は開区間 $(-1, 1)$ に含まれていなければなりません.

6.3.9 legndr.i ライブラリ

"legndr.i" ライブラリには Legendre 多項式を扱うための函数が収録されています. このライブラリに含まれる函数を利用するためには 'include,"legndr.i"' でライブラリの読込を行っておく必要があります:

legendr.i ライブラリの主要な函数

構文	概要
legendr(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩, ⟨ 配列$_3$⟩)	Legendre 函数 Plm(x) を計算.
ylm_coef(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)	$\sqrt{\dfrac{(2\langle \text{配列}_1\rangle+1)(\langle \text{配列}_1\rangle-\langle \text{配列}_2\rangle)!}{4\pi(\langle \text{配列}_1\rangle+\langle \text{配列}_2\rangle)!}}$ を計算

6.3.10 series.i ライブラリ

"series.i" ライブラリには幾何級数に関連する函数が収録されています:

series.i ライブラリの主要な函数

構文 (series_s, series_r, seriers_n)
series_s(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)
series_r(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)
series_n(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)

series_s 函数:　`series_s`(r,n) で有限幾何級数 $\sum_{i=0}^{n} r^i$ を計算します.
ここで `series_s`$(r,-n)$ = `series_s`$(1./r,n)$ の関係が成立しますが, ⟨ 配列$_2$⟩ は 0 次元の配列か, 長さ 1 の配列, あるいは ⟨ 配列$_1$⟩ と同じ長さでなければなりません.

series_s 函数:　$s=\sum_{i=0}^{n} r^i$ とするときに '`series_r`(s,n)' は r を返します. また `series_r`$(s,-n)=1./$`series_r`(s,n) や `series_r`$(s,0)=s-1$ を満します.
⟨ 配列$_2$⟩ は 0 次元の配列か, 長さが 1 の配列, あるいは ⟨ 配列$_1$⟩ と同じ長さでなければなりません.

series_s 函数:　$s=\sum_{i=0}^{n} n^i$ とするときに `series_n`(s,r) から最小の n を返します.
そのために引数の配列は共に同じ長さでなければなりません.

6.4 補間や近似に関連する函数

Yorick は区分線形補間, 多項式や有理式による補間, spline 補間や Chevyshev 多項式による近似に関連するライブラリを持っています.

6.4.1 fitlsq.i ライブラリ

“fillsq.i” ライブラリには区分線形補間を最小二乗法を用いて計算する函数が収録されています:

fitlsq.i ライブラリの主要な函数

構文 (fitlsq)
fitlsq(⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩, ⟨ 配列$_{\mathrm{X_p}}$⟩,weight=)

fitlsq 函数: 同じ大きさの配列の ⟨ 配列$_{\mathrm{Y}}$⟩ と ⟨ 配列$_{\mathrm{X}}$⟩ から最小二乗法を用いて区分線形函数による補間函数を構築し, ⟨ 配列$_{\mathrm{X_p}}$⟩ に対応する 配列$_{\mathrm{Y_p}}$ を返します:

```
x=span(0,5,1001);
y=sin(x)^13
xp=span(0,5,51)
yp=fitlsq(y,x,xp)
fma;plg,y,x
plg,yp,xp,color="red"
```

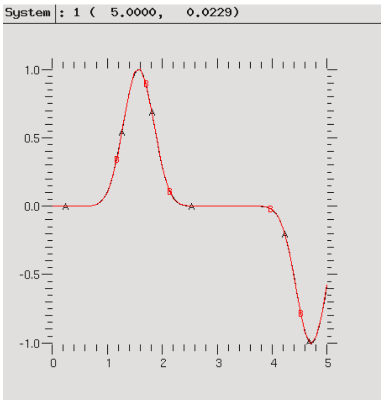


図 6.1: fitlsq 函数による補間のグラフ

キーワード weight は同じ長さの ⟨ 配列$_{\mathrm{X}}$⟩ と ⟨ 配列$_{\mathrm{Y}}$⟩ に対して重みを付加します.

6.4.2 fitrat.i ライブラリ

“fitrat.i” ライブラリは多項式や有理式による補間に関係する函数が収録されています. これらの函数は ⟨ 配列$_{\mathrm{X}}$⟩ の長さを n とするときに $n-1$ 次の多項式や有理式で補間して ⟨ 配列$_{\mathrm{X_p}}$⟩ を評価します:

fitrat.i ライブラリの主要な函数

構文	概要
fitpol(⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩, ⟨ 配列$_{\mathrm{X_p}}$⟩,keep=)	$n-1$ 次の多項式で補間
fitrat(⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩, ⟨ 配列$_{\mathrm{X_p}}$⟩,keep=)	$n-1$ 次の有理式で補間

6.4.3　spline.i ライブラリ

“spline.i” ライブラリは spline 補間に関連する函数を収録します:

spline.i ライブラリの主要な函数

構文 (spline, tspline)
spline($\langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle$)
spline($\langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
spline($\langle$ 配列$_{\mathrm{dXdY}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
tspline($\langle$ 配列$_{\mathrm{T}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle$)
tspline($\langle$ 配列$_{\mathrm{T}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
tspline($\langle$ 配列$_{\mathrm{T}}\rangle, \langle$ 配列$_{\mathrm{d^2Yd^2X}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
sprime($\langle$ 配列$_{\mathrm{dXdY}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)

これらの函数はキーワードとして dydx1 と dydx0 を取ります.

spline 函数:　$\langle$ 配列$_{\mathrm{Y}}\rangle$ を Y 座標, $\langle$ 配列$_{\mathrm{X}}\rangle$ を X 座標とする点列から, 3 次の spline 補間を計算する函数です. $\langle$ 配列$_{\mathrm{X}}\rangle$ は単調減少, あるいは単調増加の何れかでなければなりません. また, $\langle$ 配列$_{\mathrm{X}}\rangle$ と $\langle$ 配列$_{\mathrm{Y}}\rangle$ で定められる点列での微分は $\langle$ 配列$_{\mathrm{dXdY}}\rangle$ で与えます.

tspline 函数:　テンション付き spline 補間を行う函数です. $\langle$ 配列$_{\mathrm{X}}\rangle$ は単調減少, あるいは単調増加の何れかでなければなりません. テンションは $\langle$ 配列$_{\mathrm{T}}\rangle$ で指定され, この配列には 0 次元配列か 1 次元配列が指定できます. ここで指定した配列が 0 次元であれば, tspline 函数内部で用いられるテンションは $k = \langle$配列$_{\mathrm{T}}\rangle*(\mathrm{numberof}-1)/(\max(\langle$配列$_{\mathrm{X}}\rangle) - (\langle$ 配列$_{\mathrm{X}}\rangle))$ になります. また, テンションを 1 次元配列で指定した場合, その配列の大きさは numberof($\langle$ 配列$_{\mathrm{X}}\rangle$) $-$ 1 に等しくなければなりません. なお, テンションが 0 の場合に 3 次 spline 補間と一致し, ∞ であれば区間線形近似になります.

sprime 函数:　指定した点での微分を計算する函数です. ここで $\langle$ 配列$_{\mathrm{X}}\rangle$ は単調減少, あるいは単調増加の何れかでなければなりません.

6.4.4 splinef.i ライブラリ

splinef.i ライブラリの主要な函数

構文 (splinef, splined, splinei, splinelsq)
splinef($\langle$ 配列$_{\mathrm{X_Y_dXdY}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
splinef($\langle$ 配列$_{\mathrm{dYdX}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
splined($\langle$ 配列$_{\mathrm{X_Y_dXdY}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
splined($\langle$ 配列$_{\mathrm{dYdX}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
splinei($\langle$ 配列$_{\mathrm{X_Y_dXdY}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
splinei($\langle$ 配列$_{\mathrm{dYdX}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{X_p}}\rangle$)
splinelsq($\langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{xfit}}\rangle$)

splinef 函数: $\langle$ 配列$_{\mathrm{X_p}}\rangle$ に対する 3 次 spline 補間を行う函数です．ここで $\langle$ 配列$_{\mathrm{X_Y_dXdX}}\rangle$ は [$\langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{dYdX}}\rangle$] で構成された $3 \times n$ の配列 (n は $\langle$ 配列$_{\mathrm{X}}\rangle$ の長さ) になります．

splined 函数: Y 座標のベクトル $\langle$ 配列$_{\mathrm{Y}}\rangle$ と X 座標のベクトル $\langle$ 配列$_{\mathrm{x}}\rangle$ で指定される曲線に対して各点における微分を $\langle$ 配列$_{\mathrm{dXdY}}\rangle$ で定め, これらの配列を基に 3 次 spline 補間の微分式を構成して末尾のベクトル $\langle$ 配列$_{\mathrm{X_p}}\rangle$ で指定した点での微分を返却する函数です．なお, これらの配列を 1 つの [X,Y,DYDX] の形式の配列 $\langle$ 配列$_{\mathrm{X_Y__dXdY}}\rangle$ に纏めることもできます．splinei 函数はこの splined 函数の積分版になります．

splinei 函数: Y 座標のベクトル $\langle$ 配列$_{\mathrm{Y}}\rangle$ と X 座標のベクトル $\langle$ 配列$_{\mathrm{x}}\langle$ で指定される曲線に対し, その各点での定積分を $\langle$ 配列$_{\mathrm{dXdY}}\rangle$ で定め, これらの配列を基に 3 次 spline 補間の積分式を構成し, 末尾のベクトル $\langle$ 配列$_{\mathrm{X_p}}\rangle$ で指定した点での定積分を返却する函数です．なお, これらの配列を 1 つの [X,Y,DYDX] の形式の配列 $\langle$ 配列$_{\mathrm{X_Y__dXdY}}\rangle$ に纏めることもできます．splined 函数はこの splinei 函数の微分になります．

splinelsq 函数: 最小 2 乗法による補間を行う函数です．splinelsq は Y 座標の $\langle$ 配列$_{\mathrm{Y}}\rangle$ と X 座標の $\langle$ 配列$_{\mathrm{X}}\rangle$ に加え, $\langle$ 配列$_{\mathrm{XFIT}}\rangle$ を使って [$\langle$ 配列$_{\mathrm{X}}\rangle, \langle$ 配列$_{\mathrm{Y}}\rangle, \langle$ 配列$_{\mathrm{dXdY}}\rangle$] を生成します．この配列を使って, splinef 函数を使って点 X_p における値が計算できます．ここで $\langle$ 配列$_{\mathrm{XFIT}}\rangle$ は厳密に単調減少, あるいは単調増加のどちらか一方でなければなりません．この splinelsq 函数のキーワードに weight, y0, y1, dxdy0, dxdy1 があります．

6.4.5　digit2.i ライブラリ

2 次元の補間を行う函数が収録されています:

digit2.i ライブラリの主要な函数

構文 (digit2, interp2)
digit2(⟨ 配列$_{\mathrm{Y_0}}$⟩, ⟨ 配列$_{\mathrm{X_0}}$⟩, ⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩)
digit2(⟨ 配列$_{\mathrm{Y_0}}$⟩, ⟨ 配列$_{\mathrm{X_0}}$⟩, ⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩, ⟨ 配列$_{\mathrm{reg}}$⟩)
interp2(⟨ 配列$_{\mathrm{Y_0}}$⟩, ⟨ 配列$_{\mathrm{X_0}}$⟩, ⟨ 配列$_{\mathrm{Z}}$⟩, ⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩)
interp2(⟨ 配列$_{\mathrm{Y_0}}$⟩, ⟨ 配列$_{\mathrm{X_0}}$⟩, ⟨ 配列$_{\mathrm{Z}}$⟩, ⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩, ⟨ 配列$_{\mathrm{reg}}$⟩)

digit2 函数:　点 $(\mathrm{X_0}, \mathrm{Y_0})$ を補間する領域の添字を返却する函数です.

linterp2 函数:　点 $(\mathrm{X_0}, \mathrm{Y_0})$ での函数 $Z(X, Y)$ を補間する函数です.

6.4.6　cheby.i ライブラリ

"cheby.i" ライブラリには Chevyshev 補間に関連する函数が収録されています:

cheby.i ライブラリの主要な函数

構文 (cheby_fit, cheby_eval, cheby_integ, cheby_deriv, cheby_poly)
cheby_fit(⟨ 函数名 ⟩, ⟨ 配列$_{\mathrm{X}}$⟩, ⟨ 正整数 ⟩)
cheby_fit(⟨ 配列$_{\mathrm{f}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩, ⟨ 正整数 ⟩)
cheby_eval(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩)
cheby_integ(⟨ 配列$_1$⟩)
cheby_integ(⟨ 配列$_1$⟩, ⟨ 実数値 ⟩)
cheby_deriv(⟨ 配列$_1$⟩)
cheby_poly(⟨ 配列$_1$⟩)

cheby_fit 函数:　第 1 引数として函数名, あるいは配列を取り, 第 2 引数として X 座標に対応する配列, 第 3 引数として Chebyshev 多項式の次数を指定します. 第 1 引数として配列が指定された場合は函数内部で ⟨ 配列$_{\mathrm{X}}$⟩ と同じ長さの配列を interp 函数を使って生成します. そのために第 1 引数と第 2 引数の配列の大きさは同じものでなければなりません.
返却値はベクトル $[\mathtt{a}, \mathtt{b}, \mathtt{c}_0, \mathtt{c}_1, \ldots, \mathtt{c}_{\mathtt{N}}]$ で, ここで $\{a, b\}$ で Chebyshev 補間が適用される区間 $[a, b]$ を示し, 各 c_i が Chebyshev 多項式の係数になります.

cheby_eval 函数: cheby_fit 函数による Chebyshev 補間の結果 ⟨ 配列$_1$⟩ の評価を ⟨ 配列$_2$⟩ に対して行う函数です．返却される配列の次元は ⟨ 配列$_2$⟩ と同じ大きさです．

```
x=span(-2,2,1001)
y=sin(2*pi*x)/x*exp(-x^2);
c1=cheby_fit(y,x,10)
c2=cheby_fit(y,x,20)
plg,y,x
x1=span(-3,3,1001)
plg,cheby_eval(c1,x1),x1,color="green"
plg,cheby_eval(c2,x1),x1,color="red"
```

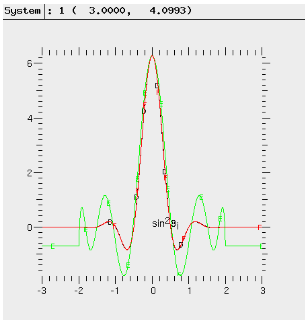


図 6.2: cheby_eval 函数

cheby_integ 函数: cheby_fit 函数で返された補間列に対する微分を返却する函数です．補間函数の積分を計算することに対応します．ここで第 2 引数が積分定数に相当するものです．

cheby_deriv 函数: cheby_fit 函数で返された補間列に対する微分を返却する函数で，補間函数の微分を計算することに対応します．

```
x=span(-2,2,1001)
y=sin(2*pi*x);
c1=cheby_fit(y,x,15)
plg,2*pi*cos(2*pi*x),x
plg,cheby_eval(cheby_deriv(c1),x),
    x,color="green"
```

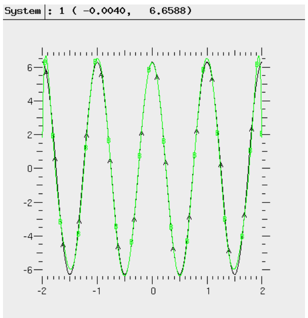


図 6.3: cheby_deriv 函数

cheby_poly 函数: cheby_fit 函数で返された補間列に対し，Chebyshev 多項式 $\sum_{i=1}^{n} A_i x^{i-1}$ の係数 A_i を返却する函数です．その性質上，入力の配列の長さが $n+3$ であれば長さ $n+1$ の配列を返却します．なお，n は Chebyshev 多項式の次数に対応します．

6.5　統計に関連する函数

6.5.1　regress.i ライブラリ

“regress.i” ライブラリは線形重回帰分析に関連する函数を含むライブラリです:

regress.i ライブラリの主要な函数

構文 (regress, regress_cov)
regress(⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩)
regress(⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩, ⟨&配列$_{\mathrm{axes}}$⟩, ⟨&配列$_{\mathrm{vals}}$⟩, ⟨&配列$_{\mathrm{chi}}$⟩,sigy=, rcond=)
regress_cov(⟨ 配列$_{\mathrm{axes}}$⟩, ⟨ 配列$_{\mathrm{vals}}$⟩)

regress 函数:　線形重回帰分析を行う函数です．ここで ⟨ 配列$_{\mathrm{X}}$⟩ は $2 \times N \times \cdots$ 次元の配列でなければなりませんが, $N =$ numberof(⟨ 配列$_{\mathrm{Y}}$⟩) とします．キーワードには sigy と rcond の 2 つがあります．sigy を指定しない場合は内部で ‘sigy=1.+0.*⟨ 配列$_{\mathrm{Y}}$⟩’ として処理が行われ, それ以外の場合, ‘sigy=1./sigy’ で処理が行われます．また `rcond` $\leq 1.e-13$ であれば `rcond` として l1.$e-13$, 0.5 以上であれば `rcond` として 0.5 内部で採用されています.

regress_cov 函数:　regress 函数によって返却されたモデルに対応する共変行列を返す函数です.

6.6　数値積分に関連する函数

6.6.1　romberg.i ライブラリ

“romberg.i” ライブラリに含まれている romberg 函数と simpson 函数を用いて 1 変数函数の数値積分が行えます．ここで誤差を指定しなければ 1.0e-6 が自動で指定されます:

romberg.i ライブラリの主要な函数

構文	概要
romberg(⟨ 函数 ⟩,⟨ 始点 ⟩,⟨ 終点 ⟩)	1 変数函数の Romberg 積分を計算
romberg(⟨ 函数 ⟩,⟨ 始点 ⟩,⟨ 終点 ⟩,⟨ 誤差 ⟩)	
simpson(⟨ 函数 ⟩,⟨ 始点 ⟩,⟨ 終点 ⟩)	Simposn の台形則による 1 変数函数の定積分を計算
simpson(⟨ 函数 ⟩,⟨ 始点 ⟩,⟨ 終点 ⟩,⟨ 誤差 ⟩)	

6.7 方程式の求解

6.7.1 roots.i ライブラリ

"roots.i" ライブラリには方程式の根等を求める函数等が収録されています．このライブラリに収録された函数を利用するためには 'include,"roots.i"' でライブラリの読込を行っておく必要があります．読込のあとで`help,roots`と入力すると "roots.i" ライブラリの概要が読めます:

roots.i ライブラリの主要な函数

構文 (nraphson, f_inverse, mnbrent, mxbrent)
nraphson(⟨ 函数とその導函数 ⟩,⟨ 始点 ⟩,⟨ 終点 ⟩)
nraphson(⟨ 函数とその導函数 ⟩,⟨ 始点 ⟩,⟨ 終点 ⟩,⟨ 誤差 ⟩)
f_inverse(⟨ 函数とその導函数 ⟩,⟨ 函数値 ⟩,⟨ 始点 ⟩,⟨ 終点 ⟩)
f_inverse(⟨ 函数とその導函数 ⟩,⟨ 函数値 ⟩,⟨ 始点 ⟩,⟨ 終点 ⟩,⟨ 誤差 ⟩)
mnbrent(⟨ 函数名 ⟩,⟨ 点$_0$⟩,⟨ 点$_1$⟩,⟨ 点$_2$⟩)
mnbrent(⟨ 函数名 ⟩,⟨ 点$_0$⟩,⟨ 点$_1$⟩,⟨ 点$_2$⟩,xmin)
mnbrent(⟨ 函数名 ⟩,⟨ 点$_0$⟩,⟨ 点$_1$⟩,⟨ 点$_2$⟩,xmin,xerr)
mxbrent(⟨ 函数名 ⟩,⟨ 点$_0$⟩,⟨ 点$_1$⟩,⟨ 点$_2$⟩)
mxbrent(⟨ 函数名 ⟩,⟨ 点$_0$⟩,⟨ 点$_1$⟩,⟨ 点$_2$⟩,⟨ 変数 ⟩)
mxbrent(⟨ 函数名 ⟩,⟨ 点$_0$⟩,⟨ 点$_1$⟩,⟨ 点$_2$⟩,⟨ 変数 ⟩,⟨ 正実数 ⟩)

nraphson 函数:; Newton-Raphson 法による函数の始点と終点で指定した区間内の零点を求める函数．ここで ⟨ 函数とその導函数 ⟩ は次の書式の函数で定義されるものです:

nraphson で引渡すべき函数の形式

func ⟨ 函数名 ⟩ (x, &函数, &導函数)　{ 函数本体 }

つまり, nraphson 函数が引数として要求する函数は, ある点での函数と導函数の値を pointer を介して与える函数に限定されます．このことを具体的な例で解説しましょう．たとえば図 6.4 に示す $\sin^5 x$ の根を区間 $[1, 6]$ で計算させる場合は次のようにします:

```
> func neko(x,&f, &dfdx){f=sin(x)^5;dfdx=5*cos(x)*sin(x)^4;};
> nraphson(neko,1,6)
3.14159
```

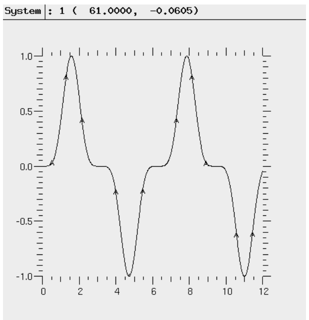


図 6.4: $\sin^5 x$ のグラフ

この函数は NewtonRaphson 法の特性上, 指定した区間内の全ての実根を求めるものではありません.

f_inverse 函数:　Newton-Raphson 法を使って函数の逆函数を計算する函数で nraphson 函数のベクトル化版とも言えます．そのため, 函数と導函数は nraphson 函数で用いた函数と同じ書式で与えることになります.

mnbrent 函数:　この mnbrent 函数は最大値を求める mxbrent 函数と似た構文, 似た機能を持つ函数で, 単変数函数の最小値を計算します.
mnbrent 函数は第 1 引数に函数名 F を取り，第 2, 第 3, 第 4 引数に異なった点 X_1, X_2, X_3 を取ります．ここで与えられた三点は,$F(X_2)$ が $F(X_1)$ と $F(X_3)$ よりも小さく, X_2 は区間 (X_1, X_3) の内部の点でなければなりません.

第 5 引数に変数を指定した場合, この変数に最大値を取るときの函数の変数値が割当てられます. そして, 第 6 変数には計算を行う際の誤差を指定します. 通常は機械精度で計算が行われます.

mxbrent 函数: 単変数函数の最大値を計算する函数で Brent の手法を用いています. mxbrent 函数は第 1 引数に函数名 F を取り, 第 2, 第 3, 第 4 引数に異なった点 X_1, X_2, X_3 を取ります. ここで与えられた三点は $F(X_2)$ が $F(X_1)$ と $F(X_3)$ よりも大きく, さらに X_2 は区間 (X_1, X_3) の内部の点でなければなりません.
第 5 引数に変数を指定した場合, この変数に最大値を取るときの函数の変数値が割当てられます. そして, 第 6 変数には計算を行う際の誤差を指定します. 通常は機械精度で計算が行われます.

```
> mxbrent(sin,0,1,2)
1
> mxbrent(sin,0,1,2,whereis)
1
> whereis
1.5708
> mxbrent(sin,0,1,2,whereis,1.0e−2)
0.999998
> whereis
```

この例で示すように第 4 引数に設定した変数に最大値を取る変数値が割当てられます. ここで第 6 引数で与えた精度が低ければ直接返却される最大値と変数値の精度も低下します.

6.7.2 zroots.i ライブラリ

“zroots.i” ライブラリには複素係数多項式の根を求める Laguerre の手法に関連する函数が収録されています:

zroots.i ライブラリの主要な函数

構文	概要
zroots(⟨ 係数配列 ⟩,imsort=)	複素係数を含む多項式 $\sum_{i=1}^{n} a(i)x^{i-1}$ の根のベクトルを Laguerre の手法を使って計算．キーワード imsort が無指定の場合と 0 を指定すると根の実部の大きさで小さいもの順に並べ，それ以外は虚部の大きさで小さいもの順に並べて返却します．
laguerre(⟨ 係数配列 ⟩, ⟨ 変数値 ⟩)	複素多項式 $\sum_{i=1}^{m} a(i)x^{i-1}$ の根を計算．

6.8　常微分方程式に関連する函数

6.8.1　rkutta.i ライブラリ

“rkutta.i” ライブラリには古典的 4 次の Runge-Kutta 法が収録されています．Runge-Kutta 法は x による微分 $dy/dx = f(y,x)$ と解 (y_n, x_n) が与えられ，x の増分を dx，$x_{n+1} = x_n + dx$ とするときに x_{n+1} に対する解 y_{n+1} を次の式から求める方法です．Yorick では rk4 函数で実体が定義されています:

古典的 4 次 Runge-Kutta 法

k_1	$=$	$f(y_n, x_n)$
k_2	$=$	$f(y_n + k_1 \cdot dx/2, x_n + dx/2)$
k_3	$=$	$f(y_n + k_2 \cdot dx/2, x_n + dx/2)$
k_4	$=$	$f(y_n + k_3 \cdot dx, x_n + dx)$
y_{n+1}	$=$	$y_n + (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) \cdot dx/6$

“rkutta.i” ライブラリの主要な函数を次にまとめておきますが，⟨ 函数名 ⟩ は (y, x) に対して x による微分 dy/dx を返す函数の名前であって，配列ではないことに注意が必要です:

rkutta.i ライブラリの主要な函数

構文 (rk_integrate, rkutta,tk4, rkdumb, bs_integrate, bstoer)
rk4(⟨ 配列$_\mathrm{Y}$⟩, ⟨ 配列$_\mathrm{dxdy}$⟩, ⟨ 配列$_\mathrm{x}$⟩, ⟨ 初期刻幅 ⟩, ⟨ 導函数名 ⟩)
rkutta(⟨ 導函数 ⟩, ⟨ 配列$_{\mathrm{Y}_0}$⟩, ⟨ 初期値$_\mathrm{x}$⟩, ⟨ 終点$_\mathrm{x}$⟩, ⟨ 誤差 ⟩, ⟨ 初期刻幅 ⟩)
rkdumb(⟨ 導函数名 ⟩, ⟨ 初期値$_y$⟩, ⟨ 初期値$_x$⟩, ⟨ 終点$_x$⟩, ⟨ 刻幅数 ⟩)
rk_integrate(⟨ 導函数名 ⟩, ⟨ 配列$_{\mathrm{Y}_0}$⟩, ⟨ 配列$_\mathrm{X}$⟩, ⟨ 誤差 ⟩, ⟨ 初期刻幅 ⟩)
bstoer (⟨ 導函数 ⟩, ⟨ 配列$_{\mathrm{Y}_0}$⟩, ⟨ 初期値$_\mathrm{x}$⟩, ⟨ 終点$_\mathrm{x}$⟩, ⟨ 誤差 ⟩, ⟨ 初期刻幅 ⟩)
bs_integrate (⟨ 導函数名 ⟩, ⟨ 配列$_{\mathrm{Y}_0}$⟩, ⟨ 配列$_\mathrm{X}$⟩, ⟨ 誤差 ⟩, ⟨ 初期刻幅 ⟩)

なお, この rkutta 函数と bstoer 函数では内部で次にまとめた大域変数による様々な設定が行えます:

rkutta.i ライブラリで用いられる大域変数

大域変数	概要
rk_maxits	刻数の最大値, 既定値は 1000.
rk_minstep	Runge-Kutta 法の刻幅の最小値. 既定値は 0.0.
rk_maxstep	Runge-Kutta 法の刻幅の最大値. 既定値は 1.0e+35.
rn_ngood	良好な刻幅の総数.
rk_nbad	悪い刻幅の総数.
rk_nstore	rkutta の呼出後に蓄えておく Y の値の総数.

rk4: 4 次の Runge-Kutta 法による解を計算します. rk4 函数は "rkutta.i" ライブラリの Runge-Kutta 法に関連する函数の実質的な計算を行っています.

rkutta: 解曲線の始点の X 座標を ⟨ 初期値$_\mathrm{x}$⟩, 終点の X 座標を ⟨ 終点$_x$⟩, ⟨ 配列$_\mathrm{y}$⟩ を解曲線の始点の Y 座標とするときに ⟨ 終点$_x$⟩ における Y 座標を 4 次の Runge-Kutta 法で計算する函数です. ここで ⟨ 初期刻幅 ⟩ は刻幅の初期値とし, ⟨ 誤差 ⟩ を誤差の上限とします.

```
> include,"rkutta.i"
> func neko(y,x){return cos(x);}
> rkutta(neko ,[0,1,3],0,1,1.0 e−5,1.0e−2)
[0.841471,1.84147,3.84147]
```

この例では $dy/dx = \cos x$ とする函数 neko を定め, $x = 0$ における初期値 y_0 を 0, 1, 3 としたときの $x = 1$ における解曲線の Y 座標を計算しています. ここで $\sin 1 \fallingdotseq 0.841471$ なので, 十分な精度の近似解が得られていることが判ります.

また, rkutta 函数の計算結果は通常の返却値の他に大域変数 rk_y と rk_x に割当てられて解曲線の始点と終点の Y 座標と X 座標が書込まれます. 大域変数 rk_nstore に 3 以上の値を設定することで, 終点までの rkutta 函数で計算した中間点の値を記入させることができます:

```
> rk_nstore=10
> rkutta(neko ,[0,1,3],0,1,1.0 e−5,1.0e−2)
[0.841471,1.84147,3.84147]
> rk_x
 [0,0.01,0.05,0.21,0.538729,0.939512,1]
> rk_y
   [[0,1,3],[0.00999983,1.01,3.01],[0.0499792,1.04998,3.04998],[0.20846,1.20846,
  3.20846],[0.513046,1.51305,3.51305],[0.80727,1.80727,3.80727],[0.841471,
1.84147,3.84147]]
```

この例で示すように rk_nstore に 3 以上を指定すると rk_nstore を長さの上限とする配列 rk_y と rk_x に解曲線の値が割当てられますが刻幅が可変のために大域変数 rk_nstore で指定した大きさになるとは限りません.
なお, この rkutta 函数は Numerical Recipes からの odeint 函数に基づくもので, 滑らかな函数を扱うのであれば bstore 函数のほうが上手く動作します.

rk_integrate: 〈 導函数名 〉で与えられた函数 $f(y,x)$ の x による積分を rkutta 函数を用いて行う函数です. 第 3 引数の 〈 配列$_{\mathrm{X}}$〉 は単調増加, あるいは単調減少のいずれかの配列とし, この配列が $dy/dx = f(y,x)$ の解曲線の X 座標を与えます. そして, 第 1 引数の配列 〈 配列$_{\mathrm{Y}_0}$〉 が解曲線の X 座標が x_0 の場合の Y 座標を与えます. したがって返却値の配列の大きさは 〈 配列$_{\mathrm{Y}_0}$〉 と 〈 配列$_{\mathrm{X}}$〉 の大きさをそれぞれ m, n とすれば $m \times n$ の大きさになります.

```
include,"rkutta.i"
x=span(0,6,1001);
func neko(y,x){return cos(x);};
test=rk_integrate(neko ,[0,1,2], x,
                    1.0e−5,1.0e−2);
fma;plg,test (3,),x,color="red";
plg,test (2,),x,color="blue";
plg,test (1,),x,color="green";
plg,cos(x),x,color="black";
```

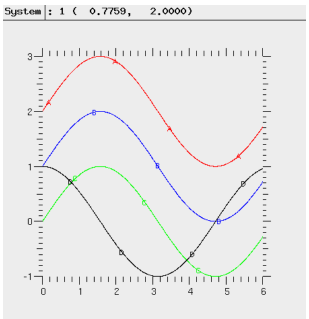


図 6.5: rk_integ の例

rk_integrate 函数内部で大域変数 rk_nstore に-1 を割当てれば大域変数 rk_y と rk_x に配列を割当てません.

bstoer: Gragg-Bulirsch-Stoer 積分器を用いている他は Runge-Kutta 法と同じです. 十分に滑らかでない函数に対しては bstoer 函数よりも rkutta 函数の方が良いでしょう.

bs_integrate: Bulirsch-Stoer 積分器を使っている他は rk_integrate 函数と同様です.

6.9 信号処理に関連する函数

6.9.1 filter.i ライブラリ

"filter.i" ライブラリにはアナログ信号処理に関連する函数が収録されています. このライブラリに収録された函数を利用するためには 'include,"filter.i"' を実行しておく必要があります. ここでは filter 函数, fil_make 函数と fil_normalize 函数の概要紹介に留めるので, 詳細はこれらの函数の help を参照して下さい.

フィルタの操作に関連する函数

フィルタの操作に関連する函数

構文 (filter, fil_make, fil_normalize)
filter($\langle$ 配列$_{\rm FILT}\rangle, \langle$ 実数$_{\rm s}\rangle, \langle$ 配列$_{\rm sig}\rangle$, pad =, shift =)
fil_make($\langle$ 配列$_{\rm POLE}\rangle, \langle$ 配列$_{\rm ZERO}\rangle$)
fil_normalize($\langle$ 配列$_{\rm FILT}\rangle, \langle$ 配列$_{\omega}\rangle, \langle$ 配列$_{\rm db}\rangle$)

filter 函数: 第 3 引数の $\langle$ 配列$_{\rm SIG}\rangle$ で与えられる入力信号に対し, 第 2 引数の $\langle$ 実数$_{\rm s}\rangle$ を標本化時間とし, 第 1 引数の $\langle$ 配列$_{\rm FILT}\rangle$ で指定した伝達函数を作用させる函数です. $\langle$ 実数$_{\rm s}\rangle > 0$ であれば $\langle$ 配列$_{\rm FILT}\rangle$ は角度周波数 (rad/sec) で正規化され, そうでなければ, 周波数 (Hz) で正規化されます. fliter 函数の結果は $\langle$ 配列$_{\rm sign}\rangle$ と同じ長さになります.
ここで第 1 引数の $\langle$ 配列$_{\rm FILT}\rangle$ は伝達函数を表現するために次の書式を持ちます:

伝達函数の表現

配列	値	概要
FILT(1)	n_p	極の個数 (正整数値)
FILT(2)	n_z	零点の個数 (正整数値)
FILT(3)	予約	
FILT(4:4+nz)	$[a_1, \ldots, a_{nz}]$	伝達函数の分子になる多項式の係数
FILT(5+nz:4+nz+np)	$[b_1, \ldots, b_{np}]$	伝達函数の分母になる多項式の係数

ここで「伝達函数」は入力信号と出力信号の Laplace 変換を I, O としたときに $O = H(s)I$ の関係を満す有理式 $H(s) = N(s)/D(s)$ のことです．ここで分子多項式 $N(s)$ は $\sum_{i=0}^{n_z} a_i s^i$, 分母多項式 $D(s)$ は $\sum_{i=0}^{n_p} b_i s^i$ となり, 分子多項式の次数が伝達函数の零点の個数, 分母多項式の次数が伝達函数の極の個数と一致します.
キーワードには pad キーワードと shift キーワードがあります.

fil_make 函数:　指定した極と零点に対応する伝達函数を返却する函数です．この伝達函数の書式は filter 函数の $\langle$ 配列$_{\mathrm{FILT}}\rangle$ の書式と一致しますが, DC 利得を 1 にしたものになります.

fil_normalize 函数:　フィルタの正規化を行う函数です.

第7章　システムに関連する事柄

Hamlet

There are more things in heaven and earth,
Horatio,Than are dreamt of in your philosophy.

ハムレット

天地の下には沢山あるぞ、ホレーショ、
お前の哲学では想像さえもせぬことが。

Hamlet: 第一幕, 第五場

7.1　検索経路について

Yorick にはライブラリや函数の読込用に三種類の検索経路があります．これらは次の大域変数に登録されています:

検索経路について

Y_LAUNCH	OS 上で実行可能ファイルが置かれるディレクトリ．このディレクトリには “yorick” や “gist” といった実行プログラムが置かれています.
Y_SITE	ライブラリが置かれたディレクトリ．このディレクトリには “i0” ディレクトリや “i” ディレクトリ, ライブラリの初期読込用のファイルが収録された “i-start” ディレクトリが置かれています.
Y_HOME	パッケージのコンパイルで用いられるファイルを収録したディレクトリ.

これらの大域変数の書換を利用者が行うことができますが, その場合は, 当然, これらの大域変数を利用する函数全てに影響が及ぶことを考慮しなければなりません.

7.2　ライブラリの読込について

Yorick では, そのライブラリの修飾子として “.i” を用います．このライブラリファイルは Yorick の大域変数, 配列, 構造体や函数といった各対象の定義, Yorick 言語によるプログラムや注釈を含んだテキストファイルです.
Yorick には, このライブラリの読込を行う函数として, include 函数や require 函数, あるいは include_all 函数があり, これらの函数でライブラリの読込を行う際にファイル名だけを指定すると, Yorick は次の順序で指示されたファイルを探します:

ファイルの検索順序

1	`.`	現行の作業ディレクトリ
2	`~/.yorick`	ホームディレクトリ上の.yorick ディレクトリ
3	`Y_SITE/i`	標準ライブラリディレクトリ
4	`Y_SITE/contribi`	
5	`Y_SITE/i0`	基本ライブラリディレクトリ
6	`Y_HOME/lib`	計算機環境依存のライブラリディレクトリ

この検索順序は set_path 函数で指定し, get_path 函数で確認できます:

ファイルの検索順序に関連する函数
構文 (set_path)
set_path,"⟨ 経路$_1$⟩;. . . : ⟨ 経路$_n$⟩"
get_path()

set_path 函数: 検索経路を指定します. 検索経路はコロン " : " で結合した文字列として与えます. ここでコロン " : " の左側の経路が検索順序の上位になります.

get_path 函数: 検索経路をコロン " : " で区切った文字列の形式で返却します. この函数は引数を必要とせず, 検索順序は返却文字列でコロン " : " の左側が右側よりも上位になります:

```
> get_path()
"./:/home/yokota/.yorick/:/usr/local/yorick-2.1/i/:/usr/local/yorick-2.1/contr\
ib/:/usr/local/yorick-2.1/i0/:/usr/local/yorick-2.1/Linux-x86_64/lib/"
```

なお, Yorick は起動時に大域変数 Y_SITE で指定されたディレクトリ下にある "i0" ディレクトリに収録されたライブラリを読込みます. この "i0" ディレクトリは最も基本的な函数を含むライブラリを包含しています. たとえば KNOPPIX/Math 2009 の Yorick の場合, 次のライブラリが収録されています:

i0 に収録されているライブラリ

ライブラリ	概要
curses.i	curses パッケージ
drat.i	
fft.i	Swarztrauber FFT ライブラリ
graph.i	グラフ描画ライブラリ
hdf5.i	HDF5 ライブラリ (hdf5 プラグイン)
hex.i	
jpeg.i	JPEG 画像ファイル利用ライブラリ (yorick-z プラグイン)
imutil.i	imutil プラグイン
matrix.i	連立方程式解法ライブラリ
paths.i	検索経路に関連するライブラリ
png.i	PNG 画像ファイル利用ライブラリ (yorick-z プラグイン)
soy.i	疎行列を扱うためのライブラリ (SOY プラグイン)
std.i	標準出力に関連する函数のライブラリ
stdx.i	標準出力に関連する函数のライブラリ
zlib.i	zlib に関連するライブラリ (yorick-z プラグイン)

なお, Yorick のソースファイルには “fft.i”, “graph.i”, “matrix.i”, “paths.i”, “std.i” と “stdx.i” のみが包含され, その他は Yorick のプラグインのライブラリです. この他の重要な函数は “i” ディレクトリに置かれています. この i ディレクトリには Yorick のデモ用ファイル “demo1.i” から “demo5.i” も収録されています.

7.3　ライブラリの読込に関連する函数

Yorick では大域変数 Y_SITE で指定されたディレクトリ上, およびカレントディレクトリ上のライブラリは経路を指定せずに include 函数や require 函数を使って読込むことができます:

プログラムやライブラリの読込に関連する函数

構文 (include, require, autoload, library, plug_in)
include, ⟨ ファイル名 ⟩
include, ⟨ ファイル名 ⟩, NOW =⟨ 整数値 ⟩
include_all, ⟨ ディレクトリ$_1$⟩, . . . , ⟨ ディレクトリ$_n$⟩
require, ⟨ ファイル名 ⟩
autoload,⟨ ファイル名 ⟩, ⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩
library
plug_in,⟨ パッケージ名 ⟩

include 函数: ライブラリの読込を行う函数です. 読込を行うライブラリのファイル名を文字列で指定します. ここで第 2 引数を指定しなければ指定したファイルを一行毎に解釈します. この第 2 引数の指定による動作を纏めておきます:

第 2 引数による動作の指定

-1	ファイル名をスタックに収め, 読み終えた時点で, 構文解析を行います.
0	一行毎に構文解析を行います. 第 2 引数を指定しなかったときと同じ働きになります.
1	構文解析を直ちに行い, 読込んだのちにプログラムの解釈を実行します. この指定により require 函数と同じ働きになります.
2	0 の場合と同じ働きになりますが, 指定したライブラリファイルが存在しなければエラー出力を行いません.
3	1 の場合と同じ働きになりますが, 指定したライブラリファイルが存在しなければエラー出力を行いません.

include_all 函数: 指定したディレクトリに含まれるライブラリの読込をアルファベット順で行います. この函数を利用した例として “Y_SITE/i0” にあるライブラリ “stdx.i” があります.

require 函数: ライブラリの読込を行う函数です. include 函数の第 2 引数に ‘1’ を指定したものと同じ働きになります.

autoload 函数: Y_SITE にある “i-start” ディレクトリに収録するファイルで用いる函数で, 指定した函数や変数といった対象をどのファイルから読込むかを指定するために用います. ここで ⟨ ファイル名 ⟩ は文字列で指定し, 各対象はそのファイルで指定された名前 (文字列ではありません) を指定します.

library 函数:　“Y_SITE/i” にある README ファイルの内容を端末に表示するだけの函数です.

plug_ind 函数:　プラグインを Yorick に動的に結合させるために用いる函数です.

7.4　shell に関連する函数

Yorick からシステムを操作したり, あるいは作業ディレクトリを変更させることが多少できます:

shell に関連する函数

構文	概要
cd(⟨ 経路 ⟩)	ディレクトリの変更
cd, ⟨ 経路 ⟩	ディレクトリの変更
pwd	作業ディレクトリの表示
mkdir,⟨ 文字列 ⟩	ディレクトリを生成する
mkdirp,⟨ 文字列 ⟩	ディレクトリを生成する (mkdir とは別)
rmdir,⟨ 文字列 ⟩	ディレクトリを消去する
lsdir(⟨ 文字列 ⟩)	ディレクトリに含まれるファイル名をリストで返却
lsdir(⟨ 文字列$_1$⟩,⟨ 配列 ⟩)	pointer を介した返還を行う
lsdir,⟨ 文字列$_1$⟩,⟨ 配列 ⟩	pointer を介した返還を行う
rename,⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩	ファイル名の変更
rename,⟨ 文字列 1⟩	ファイルの削除
get_cwd()	Yorick の作業ディレクトリを表示
get_home()	ホームディレクトリを表示
get_env(”⟨ 環境変数 ⟩”)	環境変数の値を表示
get_argv()	起動している Yorick を経路付きで表示

cd 函数:　Yorick の作業ディレクトリを変更する函数で, ⟨ 経路 ⟩ は文字列で与えます. この経路の表記は MS-Windows 版でも UNIX 風に記号 “ / ” をディレクトリの階層の区切記号として用います. また, cd 函数の構文は括弧 “( )” を省略した表記も許容します.

pwd 函数:　Yorick の作業ディレクトリを表示する函数で, 引数を必要としません.

mkdir 函数: 文字列で指定したディレクトリを生成する函数です. なお, 与える文字列ではディレクトリの階層を記号 “ / ” で区切ります. ただし, この函数でディレクトリを生成するためには, 目的のディレクトリよりも上の階層, すなわち, 親ディレクトリ全てが存在していなければなりません. 確実にディレクトリを生成するためには mkdirp 函数を用いると良いでしょう.

mkdirp 函数: UNIX の mkdir 命令に “-p オプション” を付けたものに対応する函数で, 文字列で指定したディレクトリをそのまま生成する函数です.

rmdir 函数: 文字列で指定したディレクトリを消去する函数です. ここで与える文字列はディレクトリの階層を記号 “ / ” で区切ります. なお, この函数は指定したディレクトリが空でなくても容赦なく削除するので注意が必要です.

lsdir 函数: 第 1 引数の文字列で指定したディレクトリ下のディレクトリやファイルの名前で構成された文字列ベクトルを返還する函数です. もし, 第 2 引数に変数を与えると, この第 2 引数に文字列ベクトルを代入します.

rename 函数: 第 1 引数の文字列で指定したファイル名を第 2 引数で指定したファイル名に変更します.

remove 函数: 文字列で指定したファイルの削除を行います.

get_cwd 函数: 引数を必要としない函数で, Yorick の作業ディレクトリを表示する函数です. この作業ディレクトリはファイルの open でファイルの経路を省略した場合にファイルが置かれる場所です. このディレクトリは cd 函数で変更できます.

get_home 函数: 引数を必要としない函数で, 利用者のホームディレクトリを表示します. なお, 作業ディレクトリとホームディレクトリは通常異なります.

get_env 函数: システムの環境変数の値を取出すために用いる函数です. 引数は文字列でなければなりません.

get_argv() 函数: 引数を必要としない函数で, 起動している Yorick の所在を表示します.

7.5　時間に関連する函数

時間に関連する函数

構文 (timer, timer_print, timestamp)
timer,⟨ 配列 ⟩
timer,⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩
timer_print,⟨ 文字列$_1$⟩, ⟨ 配列$_1$⟩, . . . , ⟨ 文字列$_n$⟩, ⟨ 配列$_n$⟩
timer_print,⟨ 文字列 ⟩
timer_print
timestamp()
timestamp(⟨ 変数 ⟩)
timestamp,⟨ 変数 ⟩

timer 函数:　double 型の 3 成分, 1 次元配列を引数とする函数です. この函数では値の引渡しが pointer を介して行われるために, あらかじめ引数として double 型の配列割当てた変数を容易しておく必要があります.

timer_print 函数:　timer 函数で得られた配列を整形して表示する函数で, 文字列と 3 成分のベクトルを対で引数とします.

timestamp 函数:　日時を表示する函数です. 引数に変数を指定したときに, その変数に対して整数値の代入・割当が実行されます. ここで代入される整数値は世界時で 1970 年 1 月 1 日からの秒数になります.

7.6　例外処理

Yorick にも例外処理の函数があります. ここで示す函数を用いることで, エラー時の処理が定められます:

例外処理に関連する函数

構文 (catch, error, exit)
catch(⟨ 範疇 ⟩)
error,⟨ 文字列 ⟩
exit,⟨ 文字列 ⟩

catch 函数: 16 進数表現の ⟨ 範疇 ⟩ に対応する処理を行うために if 文と併用します:

範疇	
0x01	数学演算に関するエラー (SIGFPE)
0x02	入出力に関するエラー
0x04	キーボードからの中断 (Ctrl-C)
0x08	翻訳エラー (YError)
0x10	解釈エラー (error)
-1	全てのエラー

簡単な例を次に示しておきます:

```
func is_10array(a){
 if (catch(−1)) return 0;
 if ( is_array (a)){
     if (a(10)==1)return 1;
     else return 0;}
 else return 0;};
```

この函数は配列に対して 10 番目の成分が '1' であれば '1' を返し, そうでなければ '0' を返すというものです. この函数では配列に対して 10 番目の成分を参照しているために通常の処理ではエラーになりますが, catch 函数があるために処理が行われて長さが足りない配列に対しても問題なく処理が行えます:

```
> y=array(2,3)
> is_10array(y)
0
```

error 函数: エラーを人為的に発生させて引数の文字列を表示します. なお, エラーを発生させた時点で Return キーや Enter キーを押すと虫取りモードに入ります.

exit 函数: 函数内部で利用する場合, 引数の文字列を表示して函数を終了させる return 函数に似た働きをします. ただし return 函数は値の返却を伴いますが, exit 函数は引数の文字列を表示するだけです.

大域変数 after_error: この大域変数に引数を必要としない函数名を代入します. エラーが発生した時点で大域変数 after_error に指定された函数が実行されます:

```
> func dame{write,format="%s\n","もうダメだ！";}
> after_error=dame
> error
ERROR (*main*) <interpreted error function called>
WARNING source code unavailable (try dbdis function)
now at pc= 1 (of 8), failed at pc= 3
もうダメだ！
>
```

このように「もうダメだ！」を表示する函数 dame を大域変数 after_error に指定していれば, error が生じた時点で指定した函数が起動されていることが判ります.

7.7 表示函数

ストリームとは無関係で, 標準出力にのみに出力する函数を纏めておきます:

表示函数

構文 (print, pr1, print_format)
print(⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩)
print,⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩
pr1(⟨ 対象 ⟩)
print_format,⟨ 正整数$_1$⟩, char =⟨ 文字列$_1$⟩, short =⟨ 文字列$_2$⟩, int =⟨ 文字列$_3$⟩, float =⟨ 文字列$_4$⟩, double =⟨ 文字列$_5$⟩, complex =⟨ 文字列$_6$⟩, pointer =⟨ 文字列$_7$⟩

print 函数:　複数の対象を引数として取り, それらの対象の表示を行います. この函数は引数の列を括弧で括った場合と, そうでない場合で結果が異なります:

```
> print("1",2,"3",4)
["\"1\"  2  \"3\"  4"]
> print,"1",2,"3",4
"1"  2  "3"  4
```

このように引数全体を括弧で括った場合, 引数全体を 1 つの文字列とし, その文字列を成分とする文字列ベクトルとして返却します. それに対して引数全体を括らない場合は引数をそのまま表示します. つまり, print 函数は単に表示を行う函数ではなく, むしろ, 与えられた対象で構成された文字列を生成する函数です:

```
> a=print("1",2,"3",4)
> a
```

["\"1\"　2　\"3\"　4"]

この函数の数値の表示書式は print_format 函数で設定されます.

pr1 函数:　1 個の引数のみを取って print(⟨ 対象 ⟩)(1) と同値の結果を返す函数です.

```
> pr1("1")
"\"1\""
> pr1(1)
"1"
```

print_format 函数:　print 函数や pr1 函数の数値表示の書式を定め, ここでの書式は write 函数の format キーワードで指定する書式に対応します. ここで第 1 引数の正整数は 1 行に表示される文字数を指定し, この値が未設定の場合や 0 以下の値が設定されると自動的に 79, すなわち, 79 文字が設定されます. また 39 よりも小さければ 39 に固定され, 256 を越える数値を指定すると 256 に固定されます. なお最大文字数は 5000 文字で, これが print 函数の上限です. ここで各キワードが無指定の場合, char 型の書式は "0x%02x", short 型と int 型の書式は "%d", long 型の書式は "%ld",float 型と double 型の書式は "%g", 最後に complex 型の書式は "%g%+gi" で指定されます.

7.8　システムに関連する函数

システムに関連する函数

構文 (batch, help, legal, progress_argv)
batch,⟨ 正整数 ⟩
batch()
help, ⟨ 項目 ⟩
help
legal
progress_argv(⟨ 文字列 ⟩)
progress_argv,⟨ 文字列 ⟩

batch 函数　バッチモードへの切替と情報を表示する函数で, 引数は 0 か 1, あるいは無引数です. ここで引数が 1 でバッチモードに切換えられ, 引数が 0 でバッチモードから抜けます. ここでバッチモードであれば Yorick からの端末への結果出力が切られてしまうために何も表示されません. ただし, print 函数や write 函数による表示

は切られておらず, また batch() を入力すると, 現時点でのバッチモードの切替の様子が '0' か '1' で表示されます.

help 函数: 函数や大域変数等の解説を表示する函数です. 引数として函数や大域変数等の名前を入力すると, あらかじめ include 函数や require 函数で読込まれたライブラリに対象が含まれていれば, そのライブラリ中の解説を表示し, そのような解説がなければ info 函数と同様の表示を行います. また函数名や大域変数名ではなく, long 等の与件型を入力すると, "struct 〈与件型〉 { }" といった出力を返します.
ここで利用者が構築した函数や大域変数といった対象に解説を記入するときには func 文による函数宣言や extern 文による大域変数宣言のあとに "/* DOCUMENT" で開始する注釈行に対象の解説を注釈として記述します. この仕様は MATLAB や Octave と似ています. 直接端末で対象を入力して定義してもファイルで対象が同様に定義されていなければ help 函数で対象の解説は表示されません.
help 函数の特殊な項目に copyright と warranty があります. copyright の場合は版権に関する情報が warranty に関しては無保証であることが表示されます.

legal 函数: Yorick のライセンス形態の BSDL の条文を表示する函数です.

progress_argv 函数: 引数なしで起動時に現われる文字列を表示させる函数で, 引数として文字列を与えれば, その文字列を表示する函数です. この函数はプログラム起動時での文字の表示に用いると良いでしょう.

```
> process_argv
 Copyright (c) 2005.  The Regents of the University of  California.
 All  rights  reserved.  Yorick 2.1.05  ready.  For help type 'help'
> process_argv,"三毛猫だぞー"
三毛猫だぞー
```

7.9 プログラムの起動, 中断や停止に関連する函数

Yorick には外部プログラムを起動させたり, Yorick の処理を一時的に止めたりする函数があります. これらの函数を用いることで Yorick を計算専用のエンジンとするシステムを構築することができます:

プログラムの起動に関連する函数

system(⟨ 文字列 ⟩)	shell に命令を遂行させる
spawn(⟨ 文字列 ⟩, ⟨ 函数名 ⟩)	process を生成
spawn(⟨ 文字列 ⟩, ⟨ 函数名$_1$⟩, ⟨ 函数名$_2$⟩)	

system 函数: システム側の shell に実行させる命令を文字列で引渡す函数です. popen 函数と似た函数ですが popen 函数がパイプを用いるために Yorick と並列して処理が行えるのに対し, system 函数の場合は呼出した命令が終了するまで Yorick で次の処理が行えません.

spawn 函数: process を生成する函数で, 返却値は生成した process-process 型の与件になります. popen 函数と同様に, system 函数のように起動した process が終了するまで Yorick が束縛されることはありません. ここで第 1 引数の ⟨ 文字列 ⟩ は 0 次元か 1 次元の文字列配列で, 1 次元配列の場合は第 1 成分が spawn 函数が起動する process になります. 第 2 引数は起動した process からの標準出力を処理する函数名, 第 3 引数は起動した process のエラーを受ける函数名を指定します:

```
> func test(x){print,x;print,"end!";};
test=spawn(["ls","/home/yokota/Works/Yorick/Book/TEST"],test)
> (nil)
"end!"
"test1\ntest2\n"
"end!"

> test
spawned process: ls
> typeof(test)
"spawn-process"
```

この例では Linux 環境で ls 命令を実行させています. 第 1 引数の文字列ベクトルの第 1 成分が起動すべき process であり, その引数を第 2 引数以降に並べます. 第 2 引数の函数 test は入力を単に表示したのちに "end!" と表示するだけの函数です.
この spawn 函数を使った例として Python+GTK で Yorick の GUI を構築する pyk.i ライブラリが挙げられます. 実例は Yorick GUI Tutorial[1] を参照して下さい:

[1] http://www.maumae.net/yorick/doc/gui_tutorial.php

Yorick の停止に関連する函数

構文	概要
pause(⟨ 正整数 ⟩)	正整数で指定したミリ秒間システムを停止．他のキー入力により指定したミリ秒に到達していなくても動作を開始します．時間とは無関係に Enter キーを入力するまで停止させたい場合は rdline 函数を用います．
after,⟨ 数値 ⟩, ⟨ 函数名 ⟩	第 1 引数の数値で指定した時間後に第 2 引数の函数を実行させます．第 2 引数の函数は引数を必要としない函数でなければなりません．
suspend	処理を一時停止する函数．resume 函数を on_stdout や on_stderr といったストリームに送り込むことや, 単純に Ctrl-C で起動します．
resume	suspend 函数で停止した Yorick を復帰させる函数．
quit	引数不要の Yorick を終了する函数．

これらの函数の他に, Yorick の処理を中断させる信号を発生するキー入力 Ctrl-C もあります:

Ctrl-C:　Yorick のプログラムが無限反復に陥いった場合や計算が長くて処理を中断させたい場合に Ctrl-C で処理を止められます．

7.10　虫取りモード

Yorick では構文エラー等でプログラムが異常終了した場合, 虫取りモードで問題点を調べることができます．たとえば次の函数を実行させてみましょう:

```
> func test(x){return sin(x,y);}
> test(1)
ERROR (test) expecting exactly one argument
WARNING source code unavailable (try dbdis function)
now at pc= 2 (of 12),  failed  at  pc= 8
 To enter debug mode, type <RETURN> now (then dbexit to get out)
>
```

この例では函数 test の定義で函数 sin の引数を間違えています．そのために函数 test を実行した時点で ERROR が発生しています．ここで Yorick が通常のプロンプト “>”

を出していますが, `Return キー`, あるいは`Enter キー`だけを押してみましょう. するとプロンプトが "`dbug>`" に切替わります. つまり Yorick の虫取りモードに入ったことを示します. このモードでは通常のプロンプト "`>`" が出ているときと同様の処理ができますが, 虫取りモードでは函数内部変数の変遷を調べることができます:

```
>
dbug> x
1
dbug> y
[]
dbug>
```

この例では函数内部の変数の値を参照しています. ここで虫取りモードから抜けたければ, `dbexit`と入力すれば虫取りモードから抜けられます:

```
dbug> dbexit
> x
[]
> y
[]
```

一旦抜けてしまうと問題を起した函数内部の局所変数は削除され, 上記の変数 x の値は 'nil' になっています.

虫取りに関連する函数

虫取りに関連する函数等を纏めておきます:

虫取りに関連する函数

構文 (yorick_stats, dbauto)
yorick_stats()
dbauto
dbauto,⟨ 正整数 ⟩

yorick_stats 函数　引数を必要としない函数で, Yorick のメモリに関する情報を含む 14 個の成分で構成されたベクトルを返却します.

dbauto 函数:　これらの函数は虫取りモードに自動的に入るかどうかを定める函数で, 通常のプロンプトが出ている状態でも使えます. 引数として 1 を取る場合は問題が生じると自動的に虫取りモードに入る設定とし, 引数として 0 を取る場合には自動

的に虫取りモードに入らない設定にします. また引数を取らない場合には, その時点の設定を別の設定に切換えます.

虫取りモードで有効な函数

さて, 虫取りモードに入ると, その中だけで使える函数があります. ここでは虫取りモードで有効な函数を簡単に纏めておきます:

虫取りモードで有効な函数

構文	概要
dbexit,⟨ 正整数 ⟩	虫取りモードから抜けるために用いる函数です. 引数として正整数を指定した場合, 正整数で指定した段階を抜けることになります.
dbexit	
dbcont	虫取りモードから抜けて, 問題のプログラムの継続実行を行います.
dbret,⟨ 値 ⟩	虫取りモードで, 障害となっている函数の値を dbret 函数の引数で置換えて, 処理を継続します.
dbskip,⟨ 正整数 ⟩	虫取りモードで問題個所のとばしを行います.
dbskip	
dbup	問題を起している函数を捨てます. 注意して利用しなければなりません.
dbinfo	虫取りモードの段階等の情報を表示します.
dbdis	虫取りモードで局所変数の変遷や函数の呼出等の表示を行います.

7.11　パッケージとプラグイン

Yorick には機能を拡張するものとしてパッケージやプラグインがあります. ここでパッケージは Yorick 言語で記述されたライブラリ集であり, 修飾子が".i"の文書ファイルです. このパッケージを利用するときは include 函数を使って各ライブラリの読込を行っておくか, Y_SITE/i-start ディレクトリ に自動的に読込を行うためのファイルを置く必要があります.
この自動読込を行うためのファイルでは autoloat 函数を用いて所定のライブラリから必要とする函数や変数を読込むように指定します.

それに対してプラグインはCやC++, あるいはFORTRANでコンパイルされた対象を含み, Yorickの機能を拡張する働きを持つファイル集です．このプラグインをYorickに動的に結合させるための函数がplug_in です.
パッケージやプラグインのインストールでは, まずパッケージはその性格上, ライブラリファイルを所定のディレクトリ/フォルダに置くだけで済みますが, プラグイン場合は, コンパイルを行う必要があり, その多くがMacOSXやUNIX環境に限定される傾向があります.

7.11.1 パッケージの概要

yutils: 何かと便利な函数が揃っているパッケージです．インストールしておくことを強く薦めます．なお, KNOPPIX/Math 2009には収録されていますが, MS-Windows版のYorickには収録されていません．また画像処理の函数ではPNM形式の画像を経由して読込や書込を扱うためにNetBPMパッケージのインストールが別途必要です. yutilsのインストールはUNIX環境であれば管理者権限を持つ利用者になって, 最初に`yorick -batch make.i`を実行し, それから`make install`を実行するだけです. MS-Windows 環境であれば, Yorick の Y_SITE の i フォルダの下に全てのファイルを移動し, “yutils_start.i” ファイルを大域変数 Y_SITE で指定されたフォルダの中にある “i-start” フォルダに移動させるだけです．この yutils_start.i ファイルの内容はautoload函数によるパッケージに収録された函数の読込指定となっています.

Spydr: このパッケージはKNOPPIX/Math 2009にもMS-Windows版のYorickにも含まれていません．UNIX環境であれば`yorick -batch make.i`を実行したのちに, `make; make install`を実行するとインストールされます.

7.11.2 プラグインの概要

Yorickで利用可能なプラグインを纏めておきます:

プラグイン概要

プラグイン	概要
yeti	ハッシュテーブル, 疎行列の処理, 正規表現, GSL の特殊函数を用いるための函数, TIF 画像の処理, メモリ操作のための函数等を収録.
yorick-z	JPEG, PNG 形式の画像入出力, zlib ライブラリの利用や ffmpeg パッケージとの連動による MPEG1 形式のファイルが生成可能.
yorick-gl	Yorick で OpenGL を使うためのプラグイン.
imutil	画像処理を行うためのプラグイン. yutils パッケージが必須.
SOY	疎行列を扱うためのプラグインです.
yao	光学シミュレーションが行えるプラグイン.
hdf5	HDF5(Hierarchical Data Format 5) を扱うためのプラグイン. HDF5 ライブラリの Ver.1.6.4 以上が必要.
ycatools	EPIC CATOOLS を利用するためのプラグイン.

ここで yorick-z プラグインは KNOPPIX/Math 2009 と MS-Windows 版の Yorick には最初から含まれているために手軽に遊ぶことができます. また yorick-z プラグインを用いた画像処理の話は §12 を参照して下さい. その他のプラグインに関しては Yorick の「非公式サイト」[23] を参照して下さい.

第8章　分岐と反復

悠々たる哉天壌、
遼々たる哉古今、
五尺の小躯を以って此大をはからむとす、
ホレーショの哲學境に何等のオーソリティを價するものぞ、
萬有の眞相は唯だ一言にして悉す、
曰く、「不可解」。

藤村操：巌頭之感より

8.1　分岐

if 文:　Yorick の条件分岐は if 文のみで, 構文は C と同様です:

if 文の構文

☆	if (⟨ 条件文 ⟩) ⟨ 文 ⟩	⟨ 条件文 ⟩ が真ならば ⟨ 文 ⟩ を実行
☆	if (⟨ 条件 ⟩) ⟨ 文$_1$⟩ else ⟨ 文$_2$⟩	⟨ 条件文 ⟩ が真ならば ⟨ 文$_1$⟩ を実行, それ以外は ⟨ 文$_2$⟩ を実行
☆	if (⟨ 条件文$_1$⟩) ⟨ 文$_1$⟩ else if(⟨ 条件文$_2$⟩) ⟨ 文$_2$⟩ . . . else if(⟨ 条件$_{n-1}$⟩) ⟨ 文$_{n-1}$⟩ else ⟨ 文$_n$⟩	⟨ 条件文$_1$⟩ が真ならば ⟨ 文$_1$⟩ を 実行,⟨ 条件文$_1$⟩ が偽で ⟨ 条件文$_2$⟩ が真ならば ⟨ 文$_2$⟩ を実行,. . ., 最後 に ⟨ 条件文$_1$⟩, . . . , ⟨ 条件文$_{n-1}$⟩ の 何れも偽であれば ⟨ 文$_n$⟩ を実行

if 文で用いられる条件文は偽であれば '0' を返却し, 真であれば '0' 以外の値を返却する文が使えます. 条件文の真理値に対応して処理される文を纏めるために括弧 "{ }" を用います. この括弧 "{ }" は文が 1 つだけのときに後続の else 文や別の if 文との混同が生じなければ省略できます[1]. また Yorick では if-then-else 文は次の省略形で表記可能です:

if 文の略記

☆	⟨ 条件文 ⟩　?　⟨ 文$_1$⟩　:　⟨ 文$_2$⟩

この構文では ⟨ 条件文 ⟩ の評価が 0 以外であれば ⟨ 文$_1$⟩ が評価され, そうでなければ ⟨ 文$_2$⟩ が評価されます:

```
> x=10;
> x<0? 1:−1
−1
> x>0? (x>5?2:1):−1
2
```

この例では `x<0?  1:-1` で `if(x<0)1;-1` と同じ意味になります. この構文は括弧 "( )" を用いて二番目の例に示すような入れ子にできます.
この構文では直接, 値を返却できるために函数の引数に記述できます. 実際, ライブラリ "std.i" の中で openb 函数の定義で利用されています.

[1]else 文の混同の問題を「ぶら下りの else の問題」と呼びます.

8.2 反復

反復処理に関連する構文を次にまとめておきます:

反復処理

構文 (while, for, goto, break)	
☆ while(⟨ 条件文 ⟩) ⟨ 文 ⟩	⟨ 条件文 ⟩ を評価した値が偽, i.e., 0 でなければ ⟨ 文 ⟩ を処理
☆ do ⟨ 文 ⟩ while (⟨ 条件文 ⟩)	⟨ 条件文 ⟩ を評価した値が偽, i.e., 0 でなければ ⟨ 文 ⟩ を処理
☆ for(⟨ 変数 ⟩ = ⟨ 初期値 ⟩; ⟨ 評価式 ⟩;⟨ 変数の増分式 ⟩) ⟨ 文 ⟩	⟨ 変数 ⟩ に対して与えた ⟨ 変数の増分式 ⟩ で ⟨ 変数 ⟩ を動かし, ⟨ 評価式 ⟩ が偽でない場合, i.e.,0 でない場合に ⟨ 文 ⟩ を実行
☆ goto ⟨ ラベル ⟩	指定したラベルに移動
☆ break	反復処理から離脱

while 文: 最初に条件文の判断を行い, 条件を満す場合に処理を実行します.

do 文: while 文を引っくり返したような構文になります. そのために条件文の判断が while 文とは逆で, 反復処理を一通り処理してから条件判断を行います.

for 文: Yorick では複雑な処理はできません. よく用いられる処理のみです.

goto 文: ラベルと併用して反復処理を行います. このラベルと break 文を組合せることで原始的な反復処理が構築できます. ここで Yorick のラベルとしては Yorick の演算子, 各種の括弧や与件を構成する上で用いられる文字を除外したアルファベットや記号 “ _ ” が利用できます. またラベルの先頭以外であれば “0” から “9” までの数字も使えます.
文中ではラベルであることを明示するためにラベル名とは別に, ラベル名の末尾に記号 “ : ” を置きますが, この記号はラベル名に含めてはいけません:

```
> for(i=1;i<128;i++){print,i;
cont> if(i%10==0)goto test;};test: print,”test”;
1
2
3
4
```

```
5
6
7
8
9
10
"test"
```

この例ではラベル名が “test” なので, 文の先頭では “test:”, goto 文ではラベル名 “test” が指示されています.

break 文:　反復処理の中断で用います:

```
> for(i=1;i<128;i++){print,i;if(i%10==0)break;};
1
2
3
4
5
6
7
8
9
10
```

break 文が処理された時点で反復処理が中断されていますが, break 文が置かれた反復処理を停止させるだけで, それよりも上の階層, すなわち break 文を含む反復処理を内包する反復処理には影響が及びません:

```
> for(i=1;i<4;i++){for(j=1;i<4;j++){k=i+j;
cont> if(k%3==0)break;write,format="k=%d,i=%d,j=%d\n",
cont> k,i,j ;};};
k=2,i=1,j=1
k=4,i=3,j=1
k=5,i=3,j=2
```

この例では変数 k が 3 の倍数になると break 文による中断が入りますが, break 文で停止させられるのは break 文が置かれた変数 j の反復のみで, 上の階層の変数 i の反復には影響が及びません.

continue 文:　反復処理の中断ではなく continue 文以降の文の処理を省きます:

```
> for(i=1;i<10;i++){if(i==5)continue;print,i;};
1
2
```

```
3
4
6
7
8
9
```

変数 i が 5 の場合の処理が抜けていることに注意して下さい．continue 文が実行されることで, あとの文の処理が省略されます．continue 文も break 文と同様に continue 文が置かれている反復にのみ影響し, 上の階層の反復処理には影響しません．

8.3 反復処理を行う上での注意事項

反復処理は一歩間違えると誤差の累積によって通常では考えられない事態に陥ります．まず一見自明な式 $11k - 10k$ の結果を k に代入させて再度計算という反復計算をやってみましょう．ここで $11k - 10k = k$ なので値は不変です．「何故, こんな下らないことを 」と思わないで私に付き合って下さい．手始めに ‘2’ に対して 1000000 回反復処理して下さい:

```
> k=2;for(i=0;i<1000000;i++){k=11*k−10*k;};k
2
```

あたりまえ？ そのとおりでしたね．では k に浮動小数点数 ‘0.2’ を代入するとどうでしょうか？

```
> k=0.2;for(i=0;i<1000000;i++){k=11*k−10*k;}k;
0.2
```

問題ありませんね…．$k = 0.2$ なら式 $11k - 10k$ の $10k$ は 2 です！では $11k - 2$ に $k = 0.2$ として今度は 100 回計算させてみましょう:

```
> k=0.2;for(i=0;i<100;i++){k=11*k−2.;};k;
2.22539e+87
```

あれあれ？ 「やっぱり無料だから仕方がない ！」．いいえ違います．たとえば, MATLAB や Octave ではどうなるでしょうか？ KNOPPIX/Math 2010 であれば画面下のツールバー左側にあるアイコン を押して仮想端末の LXTerminal を起動し, `octave` と入力すれば Octave が立上がります．もし KNOPPIX/Math 2009 以前であれば メニューから Octave (3.0) を選べば Octave が起動します．さて Octave が起動したところで `k=0.2;for i=[1:100] k=k*11-2;end;k` と入力してみましょう．すると次の結果が得られる筈です:

```
octave−3.0.1:1> k=0.2;for i=[1:100] k=k*11−2;end;k
k =  2.2254e+87
```

このように Yorick と同じ結果になります…
ならば表計算ソフトの MS-Excel や OpenOffice の Calc では？ そこで A 列の最初のセルに 0.2 を, それから 2 行目のセルに式 ‘=A1*11-2’ を記述し, その式を評価してから以降は二行目の複写と貼付を繰り返すだけです:

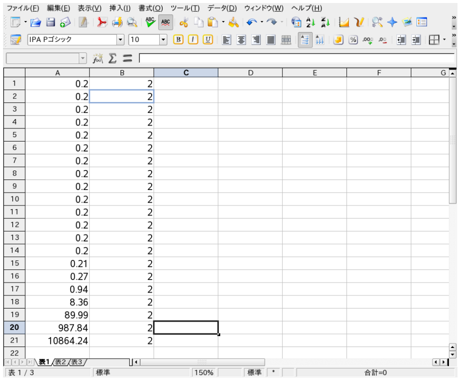
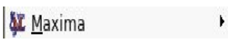


図 8.1: Calc での処理

図 8.1 には Calc での処理を示していますが, MS-Excel でも全く状況は同様に A 列の 15 行目から怪しくなるものの B 列は問題がありません. 「いや, 数式処理で計算すれば OK！」, これも残念ながら違います. たとえば数式処理の Maxima[2] を使って確認しましょう. Maxima の起動は KNOPPIX/Math 2010 であれば §13.7 で解説している KnxmLauncher  から wxMaxima を起動して下さい. また KNOPPIX/Math 2009 以前であれば画面左下側の  メニューから 

の中の何れかを選択して下さい. Maxima が立上がると `k1:0.2;for i:1 thru 20 do k1:11*k1-2;` を入力し, それから `k1;` と入力してみましょう:

```
(%i1) k1:0.2; for  i:1  thru 20 do k1:11*k1−2;
(%o1)                              0.2
(%o2)                             done
(%i3) k1;
(%o3)                    10864.23686095397
```

[2]Maxima の詳細は「はじめての Maxima」[12] や附属の KNOPPIX/Math の DVD の文献を参照して下さい. 附属 DVD の使い方は §13 を参照して下さい.

ここでは`k1:0.2;`で変数 k1 に値 0.2 を束縛させ, そのうしろの for 文で 11*k1-2 を k1 に代入させるという反復処理を 20 回実行し, 変数名`k1;`を入力するという処理です. この Maxima でも最初は '0.2' だったのに 20 回の反復処理で '10864.23686095397' に膨れ上っています. これは何故なのでしょうか?
まず最初の '2' で問題がなかった理由は, 倍精度の浮動小数点数で 2^{53} よりも小さな整数は誤差なしで計算機内部で表現され, それらの整数の演算も 2^{53} の範囲内であれば誤差が生じないからです. 次に $11k - 10k$ で問題が生じなかったのは $11k - 10k = k$ が浮動小数点数の切捨と丸めの性質から保証されるからです. 最後の $11k - 2$ で問題が生じた理由は, 実数 0.2 が 2 進数で循環小数になるために実数 0.2 に近い浮動小数点数で表現されるからです. そこで循環小数になることを先ず確認しておきましょう. ここで $0.2 = 15$ ですが, 5 は $2^2 + 1$, 2 進数で 101 となります. 10 進数と混同しないように 5 の 2 進数表記を $101_{(2)}$ と表記し, この $101_{(2)}$ で 1 を割ってみます:

$$
\begin{array}{ccc l cccccc}
 & & & & 0 & . & 0 & 0 & 1 & 1 \\
\cline{5-10}
1 & 0 & 1_{(2)} & \Big) & 1 & . & 0 & 0 & 0 & 0 \\
 & & & & & & 1 & 0 & 1 & \\
\cline{5-10}
 & & & & & & & 1 & 1 & 0 \\
 & & & & & & & 1 & 0 & 1 \\
\cline{5-10}
 & & & & & & & & & 1
\end{array}
$$

このように 0.2 の 2 進数表記は 0011 が繰り返して出現する循環小数 $0.\dot{0}01\dot{1}_{(2)}$ になります. ここで浮動小数点数は 2 の冪の和として表現されるので計算機内部で 0.2 は「微小な数 ε」を加えた浮動小数点数 $0.2 + \varepsilon$ で表現されます. ここで点 0.2 は函数 $f(x) = 11 * x - 2$ の不動点, つまり $f(a) = a$ となる点ですが, この不動点から ε 程外れると $f(0.2 + \varepsilon) = 0.2 + 11\varepsilon$ と本来の誤差 ε よりも 11 倍に膨れることが判ります. ここで「微小な数 ε」の大きさは ulp(0.2) で抑えられるために 2, 3 回程度の四則演算で顕在化しませんが, 反復処理を行うことで顕現したわけです. なお, 函数 ulp や浮動小数点数の詳細は §2.2.3 を参照して下さい.
では $x = 2.0$ が函数 $f(x) = 11x - 2$ の不動点だったから失敗したのでしょうか?たとえば $h(x) = 0.1x + 0.18$ を考えてみましょう. この $h(x)$ も $x = 0.2$ を不動点としますが $0.2 + \varepsilon$ に対しては $h(0.2 + \varepsilon) = 0.2 + 0.1\varepsilon$ となって誤差が 1/10 になります. この場合は反復処理を何度行っても結果がおかしくなることはありません:

```
> L=0.2;for(i=0;i<10000;i++){L=0.1*L+0.18;};L
0.2
```

これらのことを Maxima の力学系パッケージ dynamics を使って可視化してみましょう．この力学系パッケージの詳細は MaximaBook.pdf[13] を参照して下さい．力学系パッケージの Maxima への読込みは `load(dynamics)$` で行います．それから次を入力しましょう:

```
staircase(11*x-2,1/5+1/1000000000000,15,[x,0,4],[y,0,4]);
```

すると, 図 8.2 の示すグラフが描画される筈です．それから, 次の式を入力すると今度は図 8.3 に示すグラフが表示される筈です:

```
staircase(1/10*x+9/50,1/5+2,15,[x,0.1,3],[y ,0.1,3]);
```

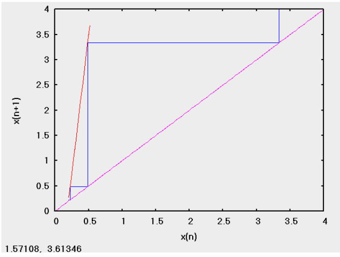


図 8.2: $11x - 2$ の不動点周辺の挙動

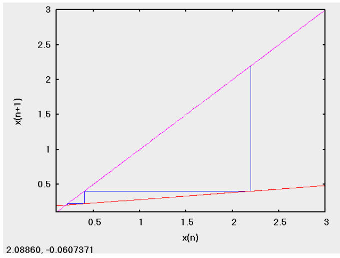


図 8.3: $x/10 + 9/50$ の不動点周辺での挙動

ここで, 図 8.2 で, 上側のグラフが函数 $f(x) = 11x - 2$ のグラフ, その下にあるグラフが $y = x$ のグラフです．そして, これらのグラフの間にある階段状のグラフが点列 $\{f(x_i)\}_{i=0}^{15}$ を繋ぐグラフです．この階段状のグラフの意味を解説しておきましょう．この staircase 函数では, 初期値 x_0 が与えられると, 函数 f による像 $f(x_0)$ を計算し, 最初に点 (x_0, x_0) と点 $(x_0, f(x_0))$ を結ぶ線分を描きますが, これがこのグラフ描画の一つの単位となります．反復を行う場合は初期値を新たに $(f(x_0), f(x_0))$ として処理を行うので, 点 $(x_0, f(x_0))$ から点 $(f(x_0), f(x_0))$ への線分を描いて, 点 $(f(x_0), f(x_0))$ と点 $((fx_0), f^2(x_0))$ を結ぶ線分を描くのです．この最初の入力では初期値 x_0 を $x_0 = 1/5 + 10^{-11}$ としていますが, 反復処理によって徐々に不動点から遠ざかる様子が見られます．ここで Maxima の函数や数値を有理数 (分数) にした理由は浮動小数点数とは異なり, 整数や有理数演算では精度が確実に保証されるからです．これに対して図 8.3 では, 上側が $y = x$ のグラフ, 下側が函数 $g(x) = x/10 + 9/50$ のグラフで初期値 x_0a を $x_0 = 1/5 + 2$ としています．点列 $\{g(x_i)\}_{i=0}^{15}$ が徐々に不動点に吸収されてゆく様子が伺えますね．

この函数 f の不動点のように僅かに不動点から外れた点が, ころがり落るように不動点から離れてゆく性質を持った不動点を「不安定な不動点」, 函数 h の不動点のように, 不動点から外れた点もやがて不動点に吸収されてしまう不動点のことを「安定な不動点」と呼びます. このように反復処理を行う際に函数の性格を把握した上で計算を行わなければ僅かな反復でも非常識な結果を得ることになるのです.

ではどのような函数なら安心できるでしょうか. たとえば函数 f を作用させたときに, 任意の異なる二点 P_0 と P_1 に対して, 点 $f(P_0)$ と点 $f(P_1)$ の距離が本来の点 P_0 と点 P_1 の距離よりも小さくなってくれればどうでしょう? この場合は函数 f を反復して作用させても異なる点同士の距離はどんどん小さくなりますね. このことは仮に誤差が生じていても, やがては本来の位置に引き戻されることを意味します. このような函数のことを「縮小写像」と呼びますが, このような縮小写像を使った反復であれば, 安全性は保証できるのです.

なお, プログラムやシステムの開発では「KISS!」(=Keep It Simple, Stupid!) という標語があります. ここでの問題は $11k - 10k$ を k で置換えれば良い処理を, $k = 0.2$ だから, わざわざ, $11k - 2$ とするような半端に計算量を減らしたために生じています. このように闇雲の「単純」にするのではなく, 式の「意義」や「意味」, さらには「影響」も考えた上で「単純」にしなければならないのです.

8.4 反復処理速度の大雑把な比較

反復処理についてはもう 1 つ注意しなければならないことがあります. それは Yorick や MATLAB 系の言語で for 文等の反復処理によって処理速度が極端に低下することです.

まず Yorick や MATLAB 系の言語では数値行列の計算はライブラリを用いるために処理速度は高速です. しかし, これらの言語の代入文や反復処理で用いられる文は各言語が担当し, 数値行列計算ライブラリとは無縁です. ここでの処理が非効率であれば全体の処理の低下に繋がりますが, これらの言語は対話処理言語のために然程速いものではありません.

このことを Octave を使って確認してみましょう. ここで Octave の反復処理は for 文で行いますが, この for 文は次の書式になります:

MATLAB や Octave の for 文の構文

for	⟨ 変数 ⟩=[⟨ ベクトル ⟩]
	⟨ 文$_1$⟩;
	...
	⟨ 文$_n$⟩;
end;	

ここでのベクトルは Yorick では range 型となる '[1:100]' のような式のことです.
では Octave で 1000×1000 の大きさの行列の総和を通常の sum 函数を用いた方法と for 文を用いた方法で比較した例を次に示しておきましょう:

```
octave:4> V=[1:1000];
octave:5> x=rand(1000);
octave:6> k1=0;
octave:7> A=cputime;for i2=V for i1=V k1=k1+x(i1,i2);end;end;cputime-A
ans =  5.6924
octave:8> A=cputime;sum(x);cputime-A
ans =  0.0080010
```

ここで時間に CPUTIME を用いています. この結果からも判るように, for 文を用いない通常の函数 sum を用いた処理が段違いに高速です. この例では for 文を用いた処理と sum 函数を用いた処理の比は 711.5 にもなり, 極端な処理速度の低下が生じていることが判りますね.
Yorick の場合はではどうでしょうか？

```
> neko1=neko2=array(0.0,3);
> x=random(1000,1000)
> k=0;
> timer,neko1;for(h=1;h<1001;h++){for(i=1;i<1001;i++){k=k+x(i,h);};};timer,neko2;
> neko2-neko1
[0.344022,0,0.365325]
> timer,neko1;x(*)(sum);timer,neko2;neko2-neko1
[0.012001,0.004001,0.0174038]
> timer,neko1;sum(x(*));timer,neko2;neko2-neko1
499635
[0.008001,0,0.00739408]
```

返却されたベクトルの第 1 成分が CPUTIME で, この値からも反復処理の不利さ加減が判りますが, 反復処理と添字を使った場合の比は 28.7, 反復処理と sum 函数の比ならば 43 になります. このように for 文による反復処理による処理速度の低下は Yorick でも生じていますが, Octave 程の極端な低下ではありません. また, sum 函数による処理の比較から, Yorick と Octave の処理速度はほぼ同等なので, ここで得られた速

度比はそのまま Octave と Yorick で比較できます. 以上から反復処理によって速度が低下するものの, Yorick は Octave と比べて極端な速度の低下がないと結論付けられます.
このことから Octave で処理速度に悩んでいる方は, Yorick を使うともっと幸せになれるかもしれません[3].

---

[3] MATLAB®が使えればより高速な処理が可能になりますが, Octave や Yorick のような処理言語のオーバーヘッドがやはり存在するために, 1 つの函数で処理できることを for 文で処理させれば Octave 同様に遅延が目立つ結果になります.

第9章　入出力について

武右衛門君は悄然として薩摩下駄を引きずって門を出た。可哀想に。
打ちゃっておくと巌頭の吟でも書いて華厳滝から飛び込むかも知れない。

夏目漱石：我輩は猫である (十) より

9.1 ストリームについて

Yorick ではファイルやアプリケーションへの標準入出力の経路としてストリームが用いられ, その場合は次の 2 つの与件型があります:

- text_stream 型: , このストリームは基本的に標準入出力向けに用いられ, 通常のテキスト形式のファイル操作やパイプを使った外部アプリケーション操作で用いられます.
- stream 型: このストリームはバイナリ形式のファイル操作に限定されます. ここで Yorick のバイナリ形式のファイルは PDB ファイルと呼ばれ, さまざまな計算機環境に対応したバイナリファイルへの出力が行えます. このストリームには構造体のような構造を持っています. たとえば, バイナリファイルへのストリームを F, ファイルに含まれる対象を 'a' とするときに 'F.a' で対象 a の値が参照できます.

ストリームが割当てられた変数名を Yorick 上で入力すれば, 指示したストリームがどのようなものであるかが表示されます:

```
> fb=create("test");
> f=open("tanuki")
> fb
read-write binary stream: test
  In directory: /home/yokota/Works/Yorick/Book/
> f
read-only text stream at:
  LINE: 1  FILE: /home/yokota/Works/Yorick/Book/tanuki
```

ここでは最初に test という名前のバイナリファイルを createb 函数で生成し, "tanuki" という既存のテキストファイルを open 函数 で開いています. ここで stream 型のストリーム 'fb' を入力すれば, それが読み書きができるバイナリファイルのストリームで, 対応するファイル名と在処が表示されています. 同様に text_stream 型のストリーム 'f' を入力すると, 読込だけができるファイルで "`LINE:`" のうしろの整数値 1 によってファイルポインタの位置がファイルの先頭に置かれていることが判り, そのうしろの情報からファイルの在処とファイル名が判ります.

9.2 ストリームの開閉処理

ストリームの開閉処理を行う函数を次に纏めておきます:

ストリームの開閉処理に関連する函数

構文 (open, popen, create, openb, createb, updateb, close)
⟨ 変数 ⟩ = open(⟨ 文字列 ⟩)
⟨ 変数 ⟩ = open(⟨ 文字列$_1$⟩, ⟨ 文字列$_2$⟩)
⟨ 変数 ⟩ = open(⟨ 文字列$_1$⟩, ⟨ 文字列$_2$⟩, ⟨ 整数値 ⟩)
⟨ 変数 ⟩ = popen(⟨ 命令 ⟩, ⟨ 設定 ⟩)
⟨ 変数 ⟩ = create(⟨ 文字列 ⟩)
⟨ 変数 ⟩ = openb(⟨ 文字列 ⟩
⟨ 変数 ⟩ = openb(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩
⟨ 変数 ⟩ = createb(⟨ 文字列 ⟩)
⟨ 変数 ⟩ = createb(⟨ 文字列 ⟩, ⟨ 計算機環境 ⟩)
⟨ 変数 ⟩ = updateb(⟨ 文字列 ⟩)
⟨ 変数 ⟩ = updateb(⟨ 文字列 ⟩, ⟨ 計算機環境 ⟩)
close, ⟨ 変数 ⟩

ここでストリームの与件型には text_stream 型か stream 型の二種類があります. また Yorick では複数のストリームを同時に開くことができますが, 各ストリームの名前は全て異ったものでなければなりません.

open 函数: open 函数は第 1 引数の ⟨ 文字列 ⟩ でファイル名を指定して返却値としてストリームを生成します. ここで生成したストリームは close 函数を使って閉じます. この第 2 引数では次の読込・書込設定一覧に示すような設定が行えます:

読込・書込設定一覧

指定	概要
r	読取専用.
w	書込専用. 既存のファイルは上書きされます.
a	書込専用. 既存のファイルの末尾から書込を開始します.
r+	読込・書込の双方が可能. 既存のファイルは保護されます.
w+	読込・書込の双方が可能. 既存のファイルは上書きされます.
a+	読込・書込の双方が可能. 既存のファイルの末尾に追加されます.

ここでの読込・書込設定だけを指示すると open 函数の返却型は text_stream 型となっ

てテキストファイルが扱えます．バイナリ型ファイルを扱う必要があれば上記の指定に加えて指示 “b” を追加します．つまり設定が stream 型になるのは “rb”, “wb”, “ab”, “r+b”, “rb+”, “w+b”, “wb+”, “a+b” と “ab+” を指定した場合に限定されます．また第 1 引数のみの場合は自動的に読取専用 “r” が指定されます．

第 3 引数の ⟨ 整数値 ⟩ はファイルが存在しないときに用います．ここに ‘0’ 以外の整数値を与えたときに指定したファイルが存在しなければ ‘nil’ が返却され, ‘0’ を設定した場合には第 3 引数を与えなかった場合と同様にエラーを返却します:

```
> open("三毛猫","r",0)
ERROR (*main*) cannot open file 三毛猫 (mode r)
WARNING source code unavailable (try dbdis function)
now at pc= 1 (of 16),  failed  at  pc= 9
 To enter debug mode, type <RETURN> now (then dbexit to get out)
> open("三毛猫","r",1)
[]
> open("test","r")
read-only text stream at:
  LINE: 1  FILE: /home/yokota/Works/Yorick/Book/test
```

この例では存在しないファイル名 “三毛猫” を open 函数を用いて開こうとしていますが, 第 3 引数が ‘0’ 以外の整数であれば返却値が ‘nil’ になり, 既存のファイルを指定していればストリームの内容が表示されます．

popen 函数:　外部アプリケーションとの連絡を行うためのパイプとして texst_stream 型のストリームを開く函数です．第 1 引数の ⟨ 文字列 ⟩ にはシステムに実行させる命令や外部アプリケーション名を文字列で与えます．第 2 引数は ‘0’ か ‘1’ を指定しますが, この意味を次に纏めておきます:

第 2 引数の指定

0 の場合	外部アプリケーションへの標準入力向けにストリームを開く
1 の場合	外部アプリケーションからの標準出力向けにストリームを開く

すなわち ‘0’ を指定すると外部アプリケーションを終了するまで Yorick へのキーボード入力は外部アプリケーションに転送され, ‘1’ を指定すれば逆に Yorick に外部アプリケーション側の標準出力が出力されます．また外部アプリケーションへの命令等の転送は write 函数等による text_stream 型のストリームへの書込で行われますが, この書込はバッファに蓄えられ, ストリームを閉じるとアプリケーションに送付されます．ストリームを閉じずに処理させるためには fflush 函数を用いて強制的にバッファの内容をアプリケーションに送付させる必要があります．

外部アプリケーションを起動させる函数として, system 函数もありますが, この system 函数で外部アプリケーションを起動すると, そのアプリケーションを終了するまで Yorick 側で処理が行えません. これに対し popen 函数で第 2 引数を '1' と指定すると Yorick 側の操作は通常通り行えるだけではなく, さらに開いたパイプを通じて外部アプリケーションの操作が行えるという長所があります. しかし, このパイプは MS-Windows 環境では使えません. そのために外部アプリケーションを起動させても Yorick が同時に使えるものの連携はできません. ここで実際に gnuplot を使って描画させてみましょう:

```
> x=span(−1,1,1001)+0.000001
> y=sin(6∗x)/(6∗x)
> fp=popen(”gnuplot −”,1)
> gnuplot> f=open(”tmp”,”r+w”)
> write,f,format=”%f %f'\n”,x,y;
> fflush(f);
read−write text stream at:
  LINE: 1  FILE: /home/yokota/tmp
> write,fp,format=”plot '%s'\n”,”tmp”;
> fflush(fp);
write−only text stream at:
  LINE: 1  FILE: gnuplot −
> gnuplot>
```

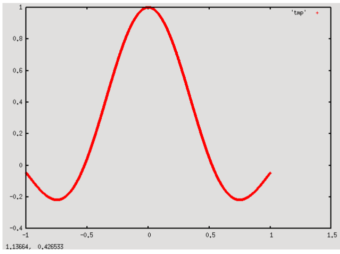


図 9.1: gnuplot による描画

popen を使って gnuplot を起動した時点と gnuplot に命令を fflush 函数で送込んだ時点で “`gnuplot>`” という文字列が出現していますが, これは命令を受け取った gnuplot 側の標準出力へのプロンプト出力が Yorick の入力画面に出力されたものです. この例では中間ファイルを用いていますが描画命令を直接送り込むこともできます:

```
> fp=popen("gnuplot",1)
> write,fp,format="plot %s;\n","sin(x)/x";
> fflush(fp);
write-only text stream at:
  LINE: 1  FILE: gnuplot -
>
```

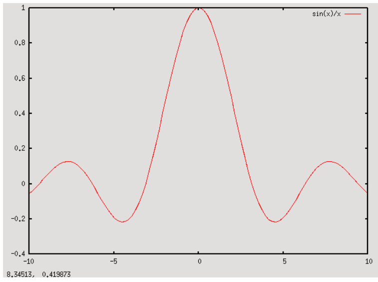


図 9.2: gnuplot との連携

この例に示すように fflush 函数によってバッファから gnuplot に命令を強制的に送付させて描画が実行されます.

create 函数:　引数として与えた文字列から指定されるファイルを生成する函数です. そのために既存のファイルは上書きされます.
ここで ‘f=create("file")’ と ‘f=open("file","w")’ は同じ意味です.

openb 函数:　Yorick 言語で記述されたバイナリ形式のファイルを読込専用のために開く函数で, 実体は open 函数を用いてファイルを “r+b” で開く函数です. もし書込が必要であれば createb 函数や updateb 函数を用いるべきです.
第 1 引数の文字列に Yorick で開くファイル名を指定します. なお第 2 引数に指定する文字列は「内容記録 (Context Log)」と呼ばれるストリーム出力の詳細を記録するためのテキストファイルを指定します. このファイルは dump_clog 函数を用いて生成されるテキストファイルで, openb 函数内部にて read_clog 函数を使ってこのファイルの内容が表示されます.

createb 函数 と updateb 函数　createb 函数はバイナリ形式のファイルを生成するための函数で, updateb 函数は内容の更新を行うために既存のハイナリ形式のファイルを開く函数です. これらの函数は共に Yorick 言語で記述されており, createb 函数はファイルを open 函数を用いて “w+b” で開き, updateb 函数はファイルが存在していなければ createb 函数で , 既存のファイルであれば open 函数から “r+b” で開く函数です.
ここで第 2 引数の ⟨ 計算機環境 ⟩ は各種計算機に適したバイナリ形式のファイルを指示します. 指定可能な値を次に纏めておきます:

計算機環境

指定値	概要
sun_primitives	Sun, HP, IBM 等の大半の EWS に適しています.
dec_primitives	古い Sun-2 や Sun-3 向け
alpha_primitives	DEC の Alpha EWS 向け
sgi64_primitives	SGI の 64bit EWS 向け
cray_primitives	Cray 1, XMP, YMP 向け
mac_primitives	Macintosh 向け
macl_primitives	Macintosh 12 バイトの倍精度向け
i86_primitives	x86 の Linux 向け
pc_primitives	IBM の PC 互換機向け
vax_primitives	VAX のみ (H 倍精度)
vaxg_primitives	VAX のみ (G 倍精度)
xdr_primitives	XDR ファイル向け

これらの計算機環境は set_primitives 函数を通して行われます. バイナリファイル操作の詳細は §9.4.4 を参照して下さい.

close 函数: open 函数で開いたストリームを閉じる函数です. 引数としてストリーム ⟨ 変数 ⟩ を 1 つ取ります. この函数は返却値を持たない函数で, 引数を括弧 “( )” で括るとエラーになります. ここでストリームを参照するためだけに開いているのであれば ‘close,f’ と ‘f=[]’ は同値ですが, それ以外であればファイルが閉じられます.

9.3 text_stream 型のストリーム入出力処理

ここで解説する函数は open 函数等で開かれた text_stream 型のストリームに対して読込や書込を行う函数です.

9.3.1 format による書式の指定

text_stream 型のストリーム処理を行う函数で, その書式の指定は “format=” の右辺に文字列を指定することで行います. ここで format に指定する書式は C の sscanf に似た書式ですが Yorick 独特の指定もあります. ここで format で指定する文字列の基本

形は読込であれば `"%*WSC"`, 書出であれば `"%FW.PSC"` となります. ここでの “F”,“W”, “P” と “S” は読込や書込の際の文字位置の指定であって “C” が対象の型に対応します.

読込の場合

***:**　この “*” が出現した位置から右側の変換は全て無視され, 対応する対象の値は ‘(nil)’ が与えられることになります.
たとえばストリームから文字列”12345” が与えられたときに format が “%3s%*s%s”, 対応する対象が a, b, c であれば, 対象 a に文字列”123” が, 対象 b と c には ‘(nil)’ が代入されます.

W:　正整数を指定, あるいは無指定にします. ここで指定する整数が対象の文字幅, すなわち文字数になります. 無指定の場合は format の文字列に沿った区切で対象が区切られます.
たとえばストリームから文字列”12345” が与えられたときに format が “%3s%2s”, 対応する対象が a と b であれば, 対象 a には文字列”123”, 対象 b には文字列”45” がそれぞれ代入されます.

S:　h, l か L の何れか一つを指定, あるいは無指定にします. この引数は C との互換性のために用いられるだけで無視されます.

書込の場合

F:　“+” か “-” の何れかを指定しますが, “+” が既定値となるために省略しても構いません. ここで “+” が右寄で “-” が左寄となります.

W:　数値が代入された対象に対して正整数を指定, あるいは無指定にします. ここで指定する整数が表示の際の全体の文字幅, すなわち対象を表示する際に用いられる文字数になります.

P:　数値が代入された対象に対して正整数を指定, あるいは無指定にします. ここで指定する整数が数値の表示桁数を決定します. なお “W” で指定される数値よりも “P” で指定される数値が超過する場合は “W” のみに “P” の数値を設定すれば良いので無意味です. 一例として read 函数による表示を示しておきます:

```
> write,format="%20.10d\n",1
          0000000001
> write,format="%-20.10d\n",1
0000000001
```

ここで最初の例は F="+", W=20, P=10 で, 最後の例は, F="-", W=20, P=10 の例になっています. ここで P は整数に対しては左側に 0 を配置することを意味し, 浮動小数点数については実部の桁数を決定します:

```
> write,format="%20.10e\n",1.0
    1.0000000000e+00
> write,format="%20.10g\n",1.0
                   1
> write,format="%20.10f\n",1.0
        1.0000000000
```

型の変換に関する指定

C: C の sscanf での型変換の指定が行えます. ここで指定可能なものを次に纏めておきます:

format で指定する C 風の書式

指定	概要
d	10 進数の整数
i	10 進数, 8 進数, 16 進数の整数
o	10 進数の整数
u	非負 10 進数の整数
x,X	16 進数の整数
e, f, g, E,G	実浮動小数点数
s	文字列

キーワード format で指定できる書式は函数によって多少異なります. ストリームから読込を行う函数, たとえば read 函数では {%c,%p,%n} に含まれる変換は使えません. そして特殊な書式として "[des]" のように変換を "d","e","s" の何れかに限定することができます. 一方でストリームへの書込を行う函数, たとえば write 函数は右寄せといった表記 "%c,%p" は読込・書込の双方で C 風の書式の "%n" に対応していません. この format の詳細については個別に関連する函数で述べることとします.

実浮動小数点数に関しては “f” が通常の小数点を伴う表記で “%10.0f” とすることで小数点以下の桁を表示しない, つまり 10 桁の整数として表示し, “%10.4f” で小数点以下 4 桁までを表示します. ここで Yorick の write 函数は小数点から 16 桁以上の数の表示はできません. そして “e” と “E” は浮動小数点数の指数表現となり, “g” と “G” は P で指定した表示桁数で整数と違いがなければ整数表示を行います:

```
> write,format="%20.2f\n",1.01
                1.01
> write,format="%20.2e\n",1.01
            1.01e+00
> write,format="%20.2g\n",1.01
                   1
> write,format="%20.2g\n",100.1
               1e+02
```

この例では write 函数に指定した書式で浮動小数点数 ‘1.01’ を表示をさせています. 最初の “f” は小数点を使った表示, “e” で指数を用いた表示で, 次の “g” で P=2 としたために表示桁は 2 桁になり, その範囲では整数の 1 と違いがないために 1 と表示されます. 同様に ‘100.1’ の場合は指数表現で ‘1e+02’, すなわち ‘100.’ として表示されていますが実体は ‘100.1’ です.

9.3.2 text_stream 型ストリームからの入力処理を行う函数

text_stream 型ストリームからの取込に関連する函数

構文 (rdline, rdfile, read_n, read, sread)
rdline(⟨ 変数 ⟩, ⟨ 整数 ⟩, prompt=⟨ 文字列 ⟩)
rdline(⟨ 変数 ⟩)
rdline(prompt=⟨ 文字列 ⟩)
rdfile(⟨ 変数 ⟩)
rdfile(⟨ 変数 ⟩,⟨ 正整数 ⟩)
read_n,⟨ 変数 ⟩, ⟨ 対象$_1$⟩, . . ., ⟨ 対象$_n$⟩
read(⟨ 変数 ⟩, format=⟨ 書式設定 ⟩, ⟨ 対象$_1$⟩, . . ., ⟨ 対象$_n$⟩)
read(prompt=⟨ 文字列 ⟩, format=⟨ 書式設定 ⟩, ⟨ 対象$_1$⟩, . . ., ⟨ 対象$_n$⟩)
sread,⟨ 変数 ⟩,format=⟨ 文字列 ⟩, ⟨ 対象$_1$⟩, . . ., ⟨ 対象$_n$⟩

これらの函数で ⟨ 変数 ⟩ で指定されたストリームを ‘nil’ に指定すれば, 自動的に標準入力に切り換えられるので, キーボードからの直接入力が行えます. このときプロン

プトの変更ができない read_n 函数や, その他の函数でキーワード prompt の値を指定していなければプロンプトとして “`read>`” が出力されます.
入力値は rdline 函数以外の函数で函数の引数の対象に対してポインタを用いた代入が行われるため, これらの対象を配列として大きさも含めて定義をしておく必要がありますが, ここでの配列の大きさは 1 次元配列に限定されません. 実際, 多次元配列の場合は 1 次元配列として入力が行われるだけで配列の大きさが変更されることもありません.
read_n 函数, read 函数や sread 函数のように複数の対象にポインタを介した代入を行う函数で引数の変数に割当てられる対象は, 基本的に同じ大きさでなければなりません. 大きさが異なっていると配列の添字が本来の大きさを超過した時点でエラーになるからです. ただし引数が 0 次元の配列である場合のみはエラーになりませんが, この場合は最後に代入された値が変数値になります.

rdline 函数: ⟨ 変数 ⟩ で指定したストリームから文字列として行単位で読込を行います. 第 2 引数として正整数を指定すると正整数で指定された大きさの文字列配列を生成し, 読込順で行を文字列とした配列を返します:

```
> rdline( nil ,2)
read> 1
read> 2
["1","2"]
```

このように rdline 函数の返却値は与件型が string 型のベクトルになります. この rdline 函数はその性格上, 引数の変数定義は不要です. ここで文字列として読込まれた与件は C の scanf 函数に相当する sread 函数を使って型の変換が行えますが, 2*3+4^2 のような数式の変換ができないことに注意して下さい.
rdline 函数はストリームの EOF を検出した時点で ‘(nil)’ を返します. そのために引数で指定する整数値が入力行の総数よりも大きければ, 余白は ‘(nil)’ で埋められます:

```
> f=open("uum")
> x=rdline(f,7)
> x
["a","b","c","d","e","d,g,h",( nil )]
```

またストリームが ‘nil’ で prompt が無指定であれば “`read>`” を表示し, prompt で指定した文字列が ‘nil’ でなければ, その文字列を表示して入力待ちになります:

```
> x=rdline(nil,3,prompt="INPUT <-:")
INPUT <-:1
INPUT <-:2
```

```
INPUT <-:3
> x
["1","2","3"]
```

この例ではストリームに直接 ‘nil’ を指定したためにストリームが標準入力になっています．そこでキーボードから “1”, “2”, “3” といった数値の入力ができているのです．rdline 函数の引数として prompt キーワードのみを指定すると, `Enter` キーや `Return` キーが入力されるまで処理を一時的に停止させることに使えます:

```
> rdline,prompt="Return キーか Enter キーを押しましょう"
Return キーか Enter キーを押しましょう
```

この場合には返却値が不要のために括弧 “()” による引数の括りは不要です．

rdfile 函数: Yorick 言語で記述された函数で, その返却値は string 型のベクトルになります．この函数は内部で rdline 函数を用いていており, ⟨ 変数 ⟩ で指定したストリームの EOF を検出するか, ⟨ 正整数 ⟩ で指定した行数に到達するまで, ストリームからの読込を行います．もしも, ⟨ 正整数 ⟩ を指定しなければ, 最大行数として 2^{20} 行が指定されています．

```
> system("cat tanuki")
1
2
3
4
5
6
7
"cat tanuki"
> f=open("tanuki");
> rdfile(f)
["1","2","3","4","5","6","7"]
```

この例では, ファイル tanuki は 1 から 7 迄の数字が各行に記述されており, それを rdfile 函数で読込んでいますが, 返却値は string 型のベクトルになっています．

read_n 函数: n 個の空白文字, タブ, コンマ, セミコロンやコロンで区切られた対象を ⟨ 変数 ⟩ で指定したストリームから取り込むための函数で, 型変換を伴いません．ここで ⟨ 対象$_1$⟩, ..., ⟨ 対象$_n$⟩ は配列として予め定義しておくか, 値が割当てられていなければなりません．そして, ⟨ 対象$_1$⟩, ..., ⟨ 対象$_n$⟩ の全ての配列の大きさが等しい場合, ストリームからの入力は ⟨ 対象$_1$⟩ から順番に ⟨ 対象$_n$⟩ へと入力され, この入力を配列の大きさ分, 繰返されます．

```
> x=array(string,3)
> y=array(string,3)
> read_n,nil,x,y,w
read> 1
read> 2
read> 3
read> 4
read> 5
read> 6
> 7
7
> x
["1","3","5"]
> y
["2","4","6"]
> w
[]
```

この例では, x と y が同じ大きさの配列ですが, w は未定義です．そのために w への代入は実行されません.

```
> z=1;
> read_n,nil,x,y,z
read> 1
read> 2
read> 3
read> 4
read> 5
read> 6
read> 7
read> 8
read> 9
> x
["1","4","7"]
> y
["2","5","8"]
> z
9
```

この例では x と y が 3 成分のベクトルですが z は 0 次元配列です．そのために, 配列の演算と同様のことが生じ, z は x と y に合せることができます．ただし, 配列としては 0 次元のために最後の代入のみが残されます.
もし引数の配列の大きさが等しくなく, 等しくないものの何れかが '0' でなければ, 配列の大きさを越える代入が発生した時点でエラーが発生して終了します.

read 函数: ⟨ 変数 ⟩ で指定したストリームから指定した書式で対象の読込を行います. この函数では引数として与えた変数に対して代入を行おうとするため, あらかじめ, 変数に対して割当を行っておく必要があります:

```
>  x=array(string,3);
> y="1"
> read(nil,prompt="<-: ",format="%s\n",x)
<-: 1
<-: 2
<-: 3
3
> read(nil,prompt="<-: ",format="%s\n",y)
<-: 1
1
> x
["1","2","3"]
> y
"1"
```

ここで示すように引数として与えられた対象の配列の大きさで入力量が異なります. このように read 函数は予め大きさが決った配列に対して用いられ, 大きさが不定のものには向いていません.

sread 函数: ⟨ 変数 ⟩ で指定した入力に対して, format で指定した書式に従って対象を取出す函数です. ここで, $\langle$ 対象$_1\rangle, \ldots, \langle$ 対象$_n\rangle$ は read 函数と同様にポインタを介した代入が行われるために, あらかじめ定義された配列でなければなりません. ここで read 函数と違う点は, ⟨ 変数 ⟩ で指定できる与件型がストリーム以外に文字列型も利用可能な点です. このときに文字列として数式を入れても, sread 函数による変換では演算子を除いた format に該当する先頭の一箇所のみが切り出されることに注意が必要です:

```
> a1=array(long,4);
> test="12^2+3*4-6/2+4"
> sread,test,format="%d",a1
> a1
 [12,0,0,0]
```

このように文字列変換は文字列総体が持つ意義ではなく, 個々の文字そのものに対して行われることに注意が必要です.

9.3.3 text_stream 型のストリームへの出力に関連する函数

text_stream 型のストリームへの出力では format で指定できる書式がより細かなものになります. まず型の指定の前に数字を記述すると, その数字が空白の数となります. そして, 数字の頭に "-" を置けば左寄, "+" を置けば右寄になります:

```
> write,format="%+10s%+10s\n","1","2"
         1         2
> write,format="%-10s%+10s\n","1","2"
1                  2
> write,format="%-10s%-10s\n","1","2"
1         2
```

ここで text_stream 型のストリーム出力に関連する函数を纏めておきましょう:

text_stream 型のストリーム出力に関連する函数

構文 (write, swrite, fflush)
write(⟨ 変数 ⟩, format =⟨ 書式設定 ⟩, linesize =⟨ 正整数 ⟩, ⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩)
write(⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩)
swrite(format =⟨ 書式設定 ⟩, linesize =⟨ 正整数 ⟩, ⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩)
fflush,⟨ 変数 ⟩

これらの函数は ⟨ 変数 ⟩ で指定したストリームに対して文字列の出力を行います. そして, ⟨ 変数 ⟩ の値が 'nil' であれば出力先が標準出力になります.

write 函数: ⟨ 変数 ⟩ で指定されたストリームに対して format で指定した書式に従ってテキスト出力を行い, 引数を括弧 "( )" で括った場合は出力文字数を返却する函数です.
この函数には format と linesize の 2 つのキーワードがあります. ここでキーワード format が出力の書式を定め, キーワード linesize が表示列数を定めます. ここで linesize の既定値は 80, すなわち 80 文字です.

swrite 函数: sread 函数が read 函数に対応するように swrite 函数は write 函数に対応する函数です.

fflush 函数: text_stream 型のストリームへの書出しを強制的に行う函数で, popen 函数によるパイプ処理で特に重要です. 実際, 引数の ⟨ 変数 ⟩ で指定されたストリームへのテキスト出力はバッファに蓄えられ, close 函数でストリームを閉じるか, fflush

函数で強制的に書込を行わせなければ, そのままバッファに蓄えられたままになります. なお, バイナリ出力はテキスト出力のようにバッファに蓄えられません.

9.3.4 栞に関連する函数

Yorick には C の rewind に対応する仕組があります. この仕組は「栞 (bookmark)」と呼ばれ, この栞の利用では次の 2 つの函数を必要とします:

栞に関連する函数

構文 (bookmark, backup)
bookmark(⟨ 変数 ⟩)
backup,⟨ 変数$_1$⟩,⟨ 変数$_2$⟩

bookmark 函数: ⟨ 変数 ⟩ で指定された text_stream 型のストリームに対して栞 (bookmark) と呼ばれる対象を生成します. この対象の与件型は bookmark 型となります.

backup 函数: backup 函数はその第 1 引数の ⟨ 変数$_1$⟩ で指定された text_stream 型のストーリムに対し, bookmark 函数で設定された栞を第 2 引数 ⟨ 変数$_2$⟩ とする函数で, C の rewind 函数のようにストリーム上の位置を栞で指定した位置に戻す働きを行う函数です.
ここで bookmark 函数で生成した栞と backup 函数の働きを確認しましょう. ファイル tanuki の内容は次のような '1' から '7' までの数字とします:

```
> system("cat tanuki")
1
2
3
4
5
6
7
"cat tanuki"
```

この例では最初にファイル"tanuki"を cat 命令[1]を system 函数で実行し, その結果を Yorick 側に表示させています. では, このファイルを open 函数で開き, このストリーム f に対して bookmark 函数で栞を生成し, rdfile 函数で読込をさせます:

[1]cat 命令は MS-Windows 版では type 命令で置換えて下さい.

```
> f=open("tanuki")
> g=bookmark(f);
> typeof(g)
"bookmark"
> texts=rdfile(f)
> texts
["1","2","3","4","5","6","7"]
> backup,f,g
> rdline(f)
"1"
```

rdfile 函数でファイルの読込を行うことで, ファイルポインタはファイル末尾に移動します. そして, backup 函数を使ってストリーム f の栞 g を呼出すと, ファイルポインタは栞を生成した時点のファイルポインタの位置, つまり, ファイル先頭に戻ります. これば rdline 函数による行の読込で確認できます.

9.4 stream 型のストリーム入出力処理

9.4.1 stream 型のストリームについて

Yorick ではテキスト形式のファイルだけではなく, バイナリ形式のファイルも扱えます. ここでバイナリ形式のファイルは stream 型のストリームに対応付けられますが, stream 型のストリームは text_stream 型のストリームと幾分性格を異にし, 構造体としての構造を持つだけでなく, 計算機環境に強く依存する与件になります. そのため, stream 型のストリームから直接対象の値を参照できたり, 計算機環境の設定が行えます.

9.4.2 対象の参照について

バイナリファイルに含まれる対象の参照は容易に行えます. たとえば F を stream 型のストリーム, a を対象とするとき, 表記 'F.a' で値が参照できます:

```
> fb=createb("testb")
> a=[1,2,3,4,5]
> b=a(,-)(..,-)(..,-);
> save,fb,a,b
> show,fb
 2 non-record variables:
      a      b
> fb.b
```

```
  [[[[1,2,3,4,5]]]]
> b=1
> fb.b
  [[[[1,2,3,4,5]]]]
```

この例ではバイナリファイル test を createb 函数で生成し, 配列 a, b を保存します. このとき, 対象 b の値はファイルに対応するストリーム fb から `fb.b` で参照できます. なお, ‘fb.b’ の値は Yorick の記憶上の変数 b に割当てられた対象とは別物で, stream 型のストリームに対応するバイナリファイルに保存された対象の値で, Yorick 側で `b=1` としたあとでも, ストリームに出力しない限り `fb.b` の値は同じです.

9.4.3 バイナリファイルが簡単に扱える函数

ここではバイナリ形式のファイルに収録された対象の情報を得たり, 対象そのものをバイナリファイルに保存したり, 読込んだりする函数で, 予備知識を必要とせず, 扱いが簡単な函数について解説します:

バイナリファイルが簡単に扱える函数

構文 (show, save, restore)
show,⟨ 変数 ⟩
show,⟨ 変数 ⟩,⟨ 文字列 ⟩
show,⟨ 変数 ⟩,1
save, ⟨ 変数 ⟩, ⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩
save, ⟨ 変数 ⟩
restore, ⟨ 変数 ⟩, ⟨ 対象$_1$⟩, . . . , ⟨ 対象$_n$⟩

show 函数； ⟨ 変数 ⟩ で指定されたバイナリファイルのストリームに対し, ファイルに含まれている対象の概要を示す函数です. ここで ⟨ 文字列 ⟩ を与えると, バイナリファイルに含まれている対象名で, ⟨ 文字列 ⟩ で指定した文字列から開始する対象の情報を返します:

```
> fb=createb("neko")
> x="12345"
> y="54321"
> save,x,y
> show,fb
 2 non−record variables:
      x       y
> x1234=128*sin(128)
> save,fb,x1234
```

```
> show,fb
 3 non−record variables:
          x       x1234           y
> show,fb,"x"
 2 non−record variables:
          x       x1234
```

この例ではファイル"neko"を生成し, そこに対象 x, y を保存します. `show,fb` によって 2 つの対象が保存されていることが判ります. つぎに変数 x1234 の値を保存してから `show,fb,"x"` と入力すると, 文字列"x"から開始する対象名の情報が表示されます.

save 函数: Yorick の対象をそのまま ⟨ 変数 ⟩ で指定したバイナリファイルに出力する函数です. ここでテキストファイルとは異なり, バイナリファイルの場合には fflush 函数でバッファの掃出しを行わなくても, 適宜, 対象はファイルに書込まれます:

```
> a=[1,2,3,4]
> c=a(,−)(..,−)
> fb=createb("neko")
> save,fb,a,c
> show,fb
 2 non−record variables:
      a       c
```

ここで引数としてストリームのみが指定された場合, Yorick 内部の全ての 1 次元以上の大きさの配列の保存が実行されます.

restore 函数: この函数は ⟨ 変数 ⟩ で指定したバイナリファイルのストリームから第 2 引数以降で指定した対象の読込を行い, save 函数とは逆の働きをします. save 函数の例に続けて説明しましょう:

```
> a=2*a;
> c=c(*)
> print,a,c
 [2,4,6,8]     [1,2,3,4]
> restore,fb,a,c
> print,a,c
 [1,2,3,4]      [[[1,2,3,4]]]
```

このように書換を行った a と c の値が restore 函数の実行後に元に戻されていますね.

9.4.4　より高度な stream 型のストリーム処理

上述の save 函数や restore 函数, あるいは show 函数では, 単純に対象をバイナリファイルへの書込みや読込み, あるいはファイルに含まれている対象を表示する程度ですが, より詳細な処理も行えます.

バイナリファイルの設定

計算機環境によってバイナリデータの扱いは異なります. これは俗に「**Big-Endian**」と「**Little-Endian**」と呼ばれるものです. すなわち, 与えられたデータを上位のバイトから記憶に格納する方式を「**Big-Endian**」, 下位のバイトから先に格納する方式を「**Litttle-Endian**」と呼びます. これらの言葉の由来は, Swift の「ガリバー旅行記」に出てくる「小人国」で, 二国間の長期の戦争の原因が鶏卵の「大きな端」から割るか, 「小さな端」から割るかということに由来したものです. 現在の主要な計算機は「Big-Endian」か「Little-Endian」の 2 種類に分類可能で, Big-Endian は IBM の汎用機や Motrola の CPU, Sun の SPARC, 「Little-Endian」は Intel の x86 系の CPU が該当します.
Yorick では, これらの Endian に加えて計算機の特性に合せたバイナリファイルの指定が行えます:

バイナリファイルの設定に関連する函数

構文 (set_primities, set_blocksize, set_filesize)
set_primitives(⟨ 変数 ⟩, ⟨ 機種 ⟩)
get_primitives,⟨ 変数 ⟩
set_blocksize,⟨ 変数 ⟩, ⟨ 整数 ⟩
set_filesize,⟨ 変数 ⟩, ⟨ 整数 ⟩

これらの函数が必要とされる理由は, Endian の問題だけではなく, 浮動小数点数の表現が計算機環境によって微妙に異なることがあるからです.

set_primitives 函数:　⟨ 変数 ⟩ で指定されたストリームに対して, ⟨ 機種 ⟩ に機種毎の所定の値を与えることで計算機環境に適したバイナリファイルを提供するための機種設定が行なえる函数です. ここでの設定は 32 個の long 型の整数配列で行なわれますが, その書式は次のようになっています:

数値配列の書式

添字	並び	用いられる型
1-18	[size,align, order]×6	char 型,shot 型,int 型,long 型, float 型,double 型
19-32	[sign_address,exponent_address, exponent_bits,mantissa_address, mantissa_bits,mantissa_normalization, exponent_bias]×2	float 型,double 型

つまり, 先頭の 18 個の数値は「大きさ (size)」,「整列 (align)」と「順序 (order)」の 3 個の並びで, char 型から double 型までの指定が 6 回繰返され, 次の 14 個の数値は float 型と double 型に対し,「符号部番地 (sign_address)」,「指数部番地 (exponent_address)」,「指数部ビット」,「仮数部番地 (mantissa_address)」,「仮数部ビット (mantissa_bits」,「仮数部正規化 (mantissa_normalization)」と「指数部の下駄履き表現 (exponent_bias) の指定」が 2 度反復されます. この配列は次に纏め計算機環境一覧から計算機環境に対応する変数を指定することで行えます:

計算機環境一覧

Endian	計算機環境						
little endian	__i86	__ibmpc	__alpha	__dec	__vax	__vaxg	
big endian	__xdr	__sun	__sun3	__sgi64	__mac	__macl	__cray

なお, set_primitives 函数で指定した設定は get_primitives 函数で調べられます:

```
> fb=createb(”testb”)
> get_primitives(fb);
 [1,1,0,2,2,−1,4,4,−1,8,8,−1,4,4,−1,8,8,−1,0,1,8,9,23,0,127,
 0,1,11,12,52,0,1023]
```

この例から得られた配列を纏めておきましょう. まず配列の添字 1-18 です:

与件型	大きさ (size)	整列 (align)	順序 (order)
☆ char 型:	1	1	0
☆ short 型:	2	2	-1
☆ int 型:	4	4	-1
☆ long 型:	8	8	-1
☆ float 型:	4	4	-1
☆ double 型:	8	8	-1

ここで「大きさ」は対象が費すバイト数,「整列」もバイト数が単位で, 順序は Endian の指定になります. この例では x86_64 環境のために Little-Endian となるので ‘-1’ となりますが, Big-Endian の場合は ‘1’ になります.

それから, 浮動小数点数に対しては次の設定が行われています:

内容	float 型	double 型
符号部番地 (sign_address)	0	0
指数部番地 (exponent_address)	1	1
指数部ビット (exponent_bit)	8	11
仮数部番地 (mantissa_address)	9	12
仮数部ビット (mantissa_bit)	23	52
仮数部正規化 (mantissa_normalization)	0	0
指数部下駄履き表示 (mantissa_bias)	127	1023

浮動小数点数の表現は配列の添字 19 から 32 が対応します. ここでの数値の単位はビットで, 浮動小数点数の表現が記述されています. たとえば, 倍精度の浮動小数点数の場合, 大きさが $8 \times 8 = 64$ ビット で, 符号部の番地が ‘0’, 指数部の番地が ‘1’ にあることが判ります. そして, 指数部のビットが 11 ビットなので, 仮数部番地は ‘12’ になります. それから残りの 52 ビットが仮数に使われます. ここで, 指数部下駄履き表示は指数の表現が 1023 加えられた整数値が用いられていることを意味しますが, 実際の利用では, この整数値から 1023 を引いた値を指数として用いることになります.

get_primitives 函数:　set_primitives 函数で $\langle$ 変数 $\rangle$ で指定されるストリームに設定した値を調べることのできる函数で, 返却値は 32 成分の long 型のベクトルになります.

set_blocksize 函数;　$\langle$ 変数 $\rangle$ で指定されたバイナリストリームに対し, キャッシュで用いられる最小区画の大きさを定める函数で, 実際に設定される大きさは, 与えた数値の 4096×2^n に最も近い値に丸められます. ここで, この区画の大きさの既定値は 0x4000(=16kB) です.

set_filesize 函数;　$\langle$ 変数 $\rangle$ で指定されたバイナリストリームに対し, ファイルの大きさを指定します. 既定値は ‘0x800000’(=8MB) です.

データの整列に関連する函数

データの整列に関連する函数

構文 (data_align, struct_align)
data_align,⟨ 変数 ⟩,⟨ 整数 ⟩
struct_align,⟨ 変数 ⟩,⟨ 整数 ⟩

data_align 函数: ⟨ 変数 ⟩ で指定した stream 型のストリームに対し, add_variable 函数を使って変数を追加する際に, これらの新しい変数の整列を開始する番地を ⟨ 整数 ⟩ で定めます. ⟨ 整数 ⟩ の値が '0' 以下であれば '0' が指定されます.

struct_align 函数: ⟨ 変数 ⟩ で指定した stream 型のストリームに対し, add_member 函数を使って構造体を追加する際に, これらの新しい構造体の整列を開始する番地を ⟨ 整数 ⟩ で定めます. ⟨ 整数 ⟩ の値が '0' 以下であれば '0' が指定されます.

変数や構造体の追加に関連する函数

変数や構造体の追加に関連する函数

構文 (add_variable, add_member, get_member, install_struct)
add_variable, ⟨ 変数 ⟩,⟨ 整数 ⟩,⟨ 文字列 ⟩,⟨ 型 ⟩,⟨ 整数ベクトル ⟩
add_member,⟨ 変数 ⟩,⟨ 文字列$_1$⟩,⟨ 整数 ⟩,⟨ 文字列$_2$⟩,⟨ 型 ⟩,⟨ 整数ベクトル ⟩
get_member(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
install_struct,⟨ 変数 ⟩,⟨ 文字列 ⟩
install_struct,⟨ 変数 ⟩,⟨ 文字列 ⟩,⟨ 整数$_1$⟩,⟨ 整数$_2$⟩,⟨ 整数$_3$⟩
install_struct, ⟨ 変数 ⟩,⟨ 文字列 ⟩,⟨ 整数$_1$⟩,⟨ 整数$_2$⟩,⟨ 整数$_3$⟩,⟨ 整数$_4$⟩

add_variables 函数: この函数はストリームを構造体として見たときに, 構造体の構成員を追加する働きをもつものと譬えることができるでしょう. つまり, ⟨ 変数 ⟩ で指定された stream 型のストリームに対し, ⟨ 文字列 ⟩ で指定した名前, ⟨ 型 ⟩ で指示した与件型で, ⟨ 整数ベクトル ⟩ で指定される大きさの配列の対象を追加する函数です:

```
> fb=createb("test")
> a=[1,2,3]
> b=a(,-)(..,-)(-,..)
> b
  [[[[1],[2],[3]]]]
> save,fb,a,b
> c=b
```

```
> add_variable,fb,−1,"c",long,dimsof(c)
> show,fb
 3 non−record variables:
      a      b      c
```

この処理によってストリームに変数 (構造体で譬えるなら構成員) が新たに追加されます．ただし, ここまでの処理では c とよぶ変数名の箱ができただけで, 肝心の c の中身は初期値のままです．この変数の具体的な値は演算子 “=” で与えなければなりません:

```
> fb.c
ERROR (*main*) encountered end−of−file before read completed
WARNING source code unavailable (try dbdis function)
now at pc= 1 (of 10),  failed  at  pc= 5
 To enter debug mode, type <RETURN> now (then dbexit to get out)
> fb.c=c
> close,fb
> fb=openb("test")
> fb.c
  [[[[1],[2],[3]]]]
```

この例では, 変数 c がファイルに追加されていますが, あくまでも, 指定された型と大きさの配列が具現化しただけで, 肝心の実体までは入っていません．そのために `fb.c` と入力するとエラーが返されています．そこで, fb.c=c で変数 c にその実体を与えてやれば, ファイルにも反映されるのです．

なお, 構造体に対しては, add_member 函数と install_struct 函数で構造体の定義を行ったのちに, add_variable 函数で対象を追加します．

ここで ⟨ 型 ⟩ は型名, 文字列, structof 函数からの返却値の何れも使えます．

add_member 函数: ⟨ 変数 ⟩ で指定された stream 型のストリームに対し, ⟨ 変数$_2$⟩ で指定された Yorick の構造体を出力する函数です．

get_member 函数: ⟨ 変数 ⟩ で指定された stream 型のストリームや構造体に対し, ⟨ 文字列 ⟩ で指定された名前の対象を取出す函数です．最初に, stream 型の対象から変数名"y"の値を取出す例を示しておきましょう:

```
> show,fb
 2 non−record variables:
      x      y
> get_member(fb,"y")
 [1,2,3,4]
```

構造体の場合も同様です:

```
> struct PET{string cat,dog;}
> MyPET=PET(cat="Mike",dog="Pochi")
> get_member(MyPET,"dog")
"Pochi"
```

この例からも判るように stream 型の対象は構造体のような構造を持っています.

install_struct 函数: 第 1 引数の ⟨ 変数 ⟩ で指定される stream 型のストリームに対し, 第 2 引数の ⟨ 文字列 ⟩ で指定される構造体を追加する函数です. 第 2 引数の ⟨ 文字列 ⟩ は add_member 函数で生成されたものでなければなりません. 具体的な例で見ることにしましょう:

```
> fb=createb("test")
> a=[1,2,3,4]
> b=a(,-)(..,-)(-,..)
> save,fb,a,b
> show,fb
 2 non-record variables:
      a       b
```

ここでは幾つかの対象も生成し, ファイル test に予め追加しておきます. それから構造体 PET を定義し, 与件型が構造体 PET となる対象 mike を生成します:

```
> struct PET{int age,weight;}
> mike=PET(age=10,weight=15)
```

次に構造体を追加します:

```
> add_member,fb,"PET",-1,"age","int";
> add_member,fb,"PET",-1,"weight","int";
>  install_struct ,fb,"PET"
```

このようにして構造体を追加するための前処理が終ります. 実際の追加は add_variable 変数を用いますが, add_variable 函数では実体を含めた追加ではありません. 実体を入れられる箱を用意したものと考えると良いでしょう. 最終的には, 他の対象と同様に `fb.mike=mike` のような代入で追加処理が行われます:

```
> add_variable,fb,-1,"mike",PET;
> show,fb
 3 non-record variables:
         a          b       mike

> fb.mike=mike
> close,fb
```

```
> fb=openb("test")
> fb.mike
PET(age=10,weight=15)
```

なお, add_variable 函数を add_member 函数や install_struct 函数の前に実行すると, ストリームに包含される対象が定義され, そのあとで実行する add_member 函数や install_struct 函数による構造体の定義が二重定義になると判断されてエラーになります.

9.4.5 内容記録ファイルに関連する函数

「内容記録ファイル」(Context Log) は「clog」と略記され, その性格上, 所定のバイナリファイルと対になって, 対応するバイナリファイルに保存された対象がどのような書式で保存されているかが記述されています. 具体的な例で解説しましょう:

```
> fb=createb("test");
> a=[1,2,3,4]
> b=a(,−)(..,−)(−,..)
> save,fb,a,b
> dump_clog,fb,"test.clog"
> close,fb
```

この例では最初に対象 a, b を createb 函数 で生成したバイナリファイル test に save 函数 で保存します. それから, ストリームを閉じる前に dump_clog 函数で"test.clog" ファイルに内容記録を保存しています. そこで, この内容記録ファイルの中身を観察しましょう:

リスト 9.1: 内容記録の例

```
"Contents Log"
+align variable [0]
+align struct [1]
+define char [1][1][0]
+define short [2][2][−1]
+define int [4][4][−1]
+define long [8][8][−1]
+define float [4][4][−1] {0 1 8 9 23 0 127}
+define double [8][8][−1] {0 1 11 12 52 0 1023}
+define string standard
```

```
+define pointer standard
+define "char *" [8][8][ pdbpointer]
+define "char*" [0][1][ pdbpointer]
long a[4]@192
long b [1][1][4][1] @224
+eod @256
```

ここで示すように, 先頭に set_primitives 函数による設定があり, 以降, 対象の情報が続きます. このファイルは openb 函数でバイナリファイルを開く際に用いられます:

内容記録ファイルに関連する函数

構文 (read_clog, _init_clog, dump_clog)
⟨ 変数 ⟩ = read_clog(⟨ 変数 ⟩, ⟨ 文字列 ⟩)
_init_clog,⟨ 変数 ⟩
dump_clog,⟨ 変数 ⟩, ⟨ 文字列 ⟩

read_clog 函数: ⟨ 文字列 ⟩ で指定したファイル名の「内容記録」を開きます.

dump_clog 函数: ⟨変数⟩ で指定されたストリームの「内容記録] を ⟨ 文字列 ⟩ で指定した名前のテキストファイルに上書きします. この内容記録は openb 函数や read_clog 函数で用いられます.

_init_clog 函数: 「内容記録」ファイルを初期化する函数です. 新しいバイナリファイルを生成し, 計算機環境の設定を行ったのちに用います.

記録の設定

save 函数による対象の保存では, save 函数を実行するたびに対象の値は書換えられます:

```
> f1=createb("test1")
> a=1
> save,f1,a
> a=2
> save,f1,a
> a=3
> save,f1,a
> f1.a
3
```

これでは, 常微分方程式の時刻 t に於ける解 $[X_t, Y_t, Z_t]$ のような対象の保存は, 計算後に全結果を配列として纏めたものに対してのみ行わなければならなくなります. この方法は時間刻幅が小さな大きな物理モデルの計算では配列が莫大なものとなるために不利です. そこで Yorick では「記録」(record) を導入します. 「記録」には「時刻」や「記録番号」といった「目印」に対応付けられ, これらの「目印」に対して関連する各種変数の値を保存します. 結果の取出は「目印」に対応する記録参照で行います.

```
> f2=createb("test2")
> a=1;
> add_record,f2,1.0,1
> save,f2,a
> a=2;
> add_record,f2,2.0,2
> save,f2,a
> a=3;
> add_record,f2,3.0,3
> save,f2,a
> jt,f2,double(1)
> f2.a
1
> jt,f2,double(3)
> f2.a
3
```

この例では, add_record 函数 を用いて「時刻」1.0, 2.0, 3.0 に対応する変数 a の値を保存させています. 値の参照では, jt 函数を使って指定した時刻の記録に移動し, その「記録」に対応する変数 a の値を参照します. なお, この例では「記録番号」も使っているので, 「時刻」と同様に「記録番号」による参照も可能です.

ここでは最初に記録設定に関連する函数を纏めておきましょう:

記録設定に関連する函数

構文 (set_filesize, add_record, get_times, get_ncycs, edit_times, collect)
set_filesize,⟨ 変数 ⟩,⟨ 整数 ⟩
add_record,⟨ 変数 ⟩,⟨ 浮動小数点数 ⟩,⟨ 整数 ⟩
add_record,⟨ 変数 ⟩,⟨ 浮動小数点数 ⟩,⟨ 整数$_1$⟩,⟨ 整数$_2$⟩
add_record,⟨ 変数 ⟩
get_times(⟨ 変数 ⟩)
get_ncycs(⟨ 変数 ⟩)
edit_times, ⟨ 変数 ⟩,⟨ ベクトル$_1$⟩,⟨ ベクトル$_2$⟩,⟨ ベクトル$_3$⟩
edit_times, ⟨ 変数 ⟩,⟨ ベクトル ⟩
edit_times, ⟨ 変数 ⟩
collect(⟨ 変数 ⟩,⟨ 文字列 ⟩)

set_filesize 函数: Yorick では, 対象を追加すると 1 つの記録が消費されます. この記録の大きさを指定する函数が set_filesize 函数で, このファイルの記録の大きさは 0x800000(8 MB) となっています. この函数は, add_record 函数の最初の呼出のあとで用いなければなりません.

add_record 函数: ⟨変数⟩ で指定したバイナリファイルのストリームに対し, set_filesize 函数で指定された大きさの領域を付加し, この領域に浮動小数点数で指定した「時刻」と整数値で指定した NCYC と呼ぶ「記録番号」を付与します.
ここで「時刻」を一定にして「記録番号」を動かしても, 逆に, 「記録番号」を一定にして「時刻」を動かしても構いません. ここで与えた「時刻」は get_times 函数で,「記録番号」は get_ncycs 函数でベクトルとして取出せます.
ここで, 簡単な例を次に示しておきましょう:

```
>  f3=createb("test")
> for(i=1;i<5;i++){
cont> a=i;
cont> add_record,f3,double(a),2^(a−1);
cont> save,f3,a;
cont> };
> get_times(f3)
 [1,2,3,4]
> get_ncycs(f3)
 [1,2,4,8]
> jt,f3,double(3);f3.a
3
> jc,f3,8;f3.a
```

4

この例では, 時刻と記録番号の関係を 記録番号 $= 2^{\text{時刻}}$ として, add_record 函数を使って各記録に時刻と番号を付与しています. ここでストリームの時刻の取出は get_times 函数によってベクトルとして行え, 同様に, 記録番号の取出は get_ncycs 函数で行えます. そして, これらの情報を基に, 「時刻」で該当する「記録」に飛ぶ jt 函数や「記録番号」で該当する「記録」に飛ぶ jc 函数を使って変数 a の値が参照できるのです.

get_times 函数:　⟨変数⟩で指定したバイナリファイルのストリームに対し, add_record 函数で与えた時刻を double 型のベクトルとして返却する函数です.

get_cycs 函数:　⟨変数⟩で指定したバイナリファイルのストリームに対し, add_record 函数で与えた NCYC を long 型のベクトルとして返却する函数です.

edit_times 函数:　「記録」が設定されたストリームに対して, 「時刻」や「記録番号」の変更が行える函数です.

collect 函数:　collect 函数は ⟨ 変数 ⟩ で指示された記録が設定されているストリームに対し, ⟨ 文字列 ⟩ に対応する変数の値を配列として取出す函数です:

```
> fb=createb("test")
> for(i=1;i<10;i++){
cont> add_record,fb,double(i),2^(i-1);
cont> save,fb,i;};
> collect(fb,"i");
  [1,2,3,4,5,6,7,8,9]
```

ここでの例では, 「時刻」で変数 i の値をストリーム fb に出力しています. ここで, ‘fb.i’ ではポインタが置かれた記録に対応する値を返すだけで, 記録全体での ‘fb.i’ の値ではありません. collect 函数は, 各記録での ‘fb.i’ を記録番号順に配列として返却する函数であるため, ‘1’ から ‘9’ までの整数を成分とする配列を返却しています.

記録の移動に関連する函数

text_stream 型のストリームに対しては, bookmark 函数による栞と backup 函数による移動がありましたが, stream 型のストリームに対しては, 「時刻」や「記録番号」, あるいは, 番号による移動が可能です:

記録の移動に関連する函数

構文 (jc, jt, jr)
jc,⟨ 変数 ⟩, ⟨ 整数 ⟩
jt,⟨ 変数 ⟩, ⟨ 浮動小数点数 ⟩
jt,⟨ 変数 ⟩, –
jt,⟨ 変数 ⟩
jr,⟨ 変数 ⟩, ⟨ 整数 ⟩

jc 函数: ⟨ 整数 ⟩ で指定された「記録番号」に最も近い記録に飛び, そこに保存された対象の参照が行えるようになります. なお, _jc 函数は 2 つの引数を取り, jc 函数の本体に相当する函数です.

jt 函数: ⟨ 浮動小数点数 ⟩ で指定された「時刻」に最も近い「記録」に移動し, そこに保存された対象の参照が行えるようになります. なお, _jt 函数は 2 つの引数を取り, jt 函数の本体に相当する函数です.

jr 函数: 「記録」のあるストリームに対し, ⟨ 整数 ⟩ で指定した番号の記録に移動し, そこに保存された対象の参照が行えるようになります. jc 函数との違いは, ここでの ⟨ 整数 ⟩ の意味が, 「記録番号」ではないことです. つまり, ここでの ⟨ 整数 ⟩ は get_times 函数や get_ncycs 函数で取出したベクトルの「添字」に対応します. したがって, ⟨ 整数 ⟩ が 0 であれば末尾の記録に移動, -1 であれば二番目に末尾の記録に移動します. なお, _jr 函数は 2 つの引数を取り, jr 函数の本体に相当する函数です.

ファイル中の対象名に関連する函数

Yorick ではストリームに含まれる変数の名前を変更させたり, ストリームに含まれる変数の値等を参照することが可能です:

ファイル中の対象名に関連する函数

構文 (set_vars, get_vars, get_addrs)
set_vars,⟨ 変数 ⟩, ⟨ ベクトル ⟩)
set_vars,⟨ 変数 ⟩, ⟨ ベクトル$_1$⟩, ⟨ ベクトル$_2$⟩)
get_vars(⟨ 変数 ⟩)
get_addrs(⟨ 変数 ⟩)

set_vars 函数: ⟨ 変数 ⟩ で指定された stream 型の対象に含まれる変数名を string 型の ⟨ ベクトル ⟩ で指定した変数名で置換する函数です:

```
> show,fb
 2 non−record variables:
      a       b

> print,fb.a,fb,b
 [1,2,3]       [[[[1],[2],[3]]]]
> set_vars,fb,["e","f"]
> show,fb
 2 non−record variables:
      e       f

> print,fb.e,fb.f
 [1,2,3]       [[[[1],[2],[3]]]]
```

ここで引数が 3 個の場合, ⟨ ベクトル$_1$⟩ が記録を持たない対象の変数名, ⟨ ベクトル$_2$⟩ が記録を持つ対象の変数名となります.

get_vars 函数: ⟨ 変数$_1$⟩ で指定したバイナリファイルに含まれる対象名と記録情報を返却する函数です. 返却値は 2 成分のベクトルで, 各成分は番地で返却されます:

```
> show,fb
 3 non−record variables:
      a       b       c

> uum=get_vars(fb)
> *uum(1)
["a","b","c"]
```

get_addrs 函数: ⟨ 変数 ⟩ で指定したバイナリファイルのストリームから, 対象が置かれた番地を返す函数です. 返却値は 16 進数表記された pointer 型の対象で構成されたベクトルになります:

第 1 成分	記録を持たない変数の絶対番地
第 2 成分	記録付けられた変数の相対番地
第 3 成分	記録の絶対番地
第 4 成分	第 3 成分に対応する記録の添字
第 5 成分	該当するファイル名

ファイルの復元

ファイルの復元を行う函数

構文 (recover_file)
recover_file,⟨ ファイル ⟩
recover_file,⟨ ファイル ⟩,⟨ 文字列 ⟩

recover_file 函数: 壊れたバイナリファイルの復元を行う函数です. ⟨ ファイル ⟩ には, 文字列でファイル名, あるいは stream 型のストリームが指定可能です. 第 2 引数の ⟨ 文字列 ⟩ は修復の際に得られたバイナリファイルの「内容記録」が保存されるテキストファイル名となります. この第 2 引数を指定しなければ, 第 1 引数で指定される ファイル名+"L" が「内容記録」のファイル名となります. 具体的にはバイナリファイル名が "test" の場合, 「内容記録」ファイルの名前は "testL" になります.

第10章　グラフ処理機能

First Clown	第一の墓掘人
In youth, when I did love, did love, Methought it was very sweet, To contract, O, the time, for, ah, my behove, O, methought, there was nothing meet.	若い時分に恋をした, 恋したさ, そりゃぁ、ええもんと思ってたぜ, それが, おぅ, 今じゃ, あー, 年貢の納時, おぅ, なんにもよくないじゃねぇか.
	Hamlet: 第五幕, 第一場

10.1　概要

Yorick のグラフ表示では PLPlot ライブラリが用いられており, 意外に細かな処理ができます. ここでは Yorick に次の入力をして下さい:

```
> x=span(0,1,1001)+0.0001;
> plg,sin(1/x),x
```

すると”Yorick 0”という名前の Window が開かれて図 10.1 に示すグラフが表示されます:

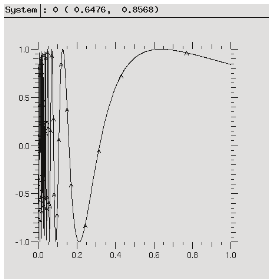


図 10.1: plg 函数による sin(1/x) の描画

描画用の Window は “Yorick ⟨ 番号 ⟩” という名前で生成されます. ここで番号を指定しなければ自動的に生成順に 0 から番号が割当てられ, Yorick から Window に対して処理を行うときは, この番号を使って処理します.
今度はグラフ上でマウスポインタを動かしてみましょう. ここでグラフの上にポインタが移動するとポインタが十字に変化します. この十字を曲線の上に置いてマウスの左ボタンをクリックして下さい. すると図が拡大されます. それから右ボタンをクリックしてみましょう. 今度は図が縮小されます. 次にマウスの右ボタンを押しながら左右に動かしてみましょう. するとグラフが拡大されて左右に移動します. 左ボタンで同様のことを行えば今度は縮小されて左右に移動します. もし 3 ボタンマウスを使っているのであれば中ボタンを押しながら左右に動かすと単にグラフが移動します. それから `limits;` と入力して下さい. すると表示が元に戻る筈です. この limits 函数は引数を与えなければグラフが丁度良く全体に収まるように再描画を行います.
今度は次の入力をして下さい:

```
> fma
> plg,cos(1/x),x
```

fma 函数で Window の初期化を行って plg 函数を実行するために前のグラフが消去されて図 10.2 に示す新しいグラフだけが表示されます:

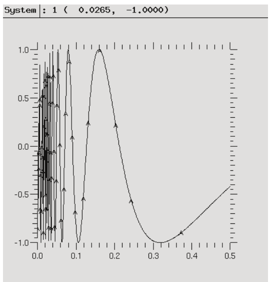


図 10.2: plg 函数による cos(1/x) の描画

ここで左ボタンを一度だけ Window 上でクリックして図を拡大し，次を入力して下さい:

> plg,sin(1/x),x,color=”red”

今度は ‘cos(1/x)’ のグラフの上に ‘sin(1/x)’ のグラフが赤で描かれますが，‘sin(1/x)’ がそのまま描画されるだけで全体は見えないかもしれません．Yorick では fma 函数を使って Window の初期化を行わなければ重ね描きになります．fma 函数は以前の描画を消す作用だけなので，limits 函数も併用しなければ描画函数の性質によっては意図した通りに出力されません．そこで `limits;` と入力しましょう:

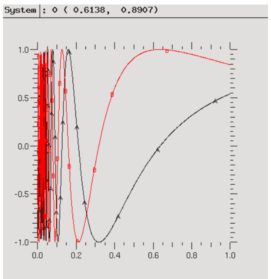


図 10.3: plg 函数による sin(1/x) と cos(1/x) の描画

これで図 10.3 に示すように全体がきちんと表示されます．
グラフ描画の基本として，重ね描きを意図しないのであれば fma 函数を実行し，グラ

フを Window に収まるようにしたければ limits 函数を実行すればよいことを覚えて下さい．さて今度は次を入力して下さい:

```
> limits,0,0.5,−1,1
```

すると図 10.4 に示すように X 座標が 0 から 0.5, Y 座標が-1 から 1 までの領域の表示に切替わります:

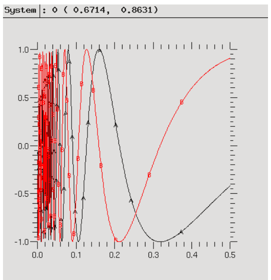


図 10.4: “`limits,0,0.5,-1,1`” の結果

今度はグラフに表題や各軸にラベルを付けてみましょう．ここで表題は pltitle 函数, X 軸と Y 軸のラベルは xytitles 函数を使います:

```
> pltitle,"black:cos(1/x), red:sin(1/x)"
> xytitles,"X−axis","Y−Axis"
```

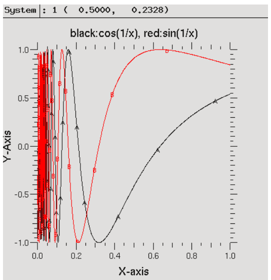


図 10.5: 表題と XY 軸のラベル付きのグラフ

これで図 10.5 に示すように表題やラベルが付きましたが, ただウィンドウは小さ過ぎるかもしれません．そこでもう少し見映えがするように大きな Window 上に表示させましょう．そのために window 函数を使って新しい Window を開きます:

```
> window,1,dpi=120,hcp="test.ps"
```

どうですか？ 今度は前よりも大きな Window が出現します．このように Window の大きさは dpi(Dot Per Inch) で指定します．それから次の操作を行ってください:

```
> plg,cos(1/x),x
> plg,sin(1/x),x,color="red"
> xytitles,"X−axis","Y−Axis"
> pltitle,"black:cos(1/x), red:sin(1/x)"
> hcp
```

これで二つの曲線と表題付きのグラフが得られました．特に最期の hcp 函数は表示画像を window 函数の hcp キーワードで指定ファイルに保存する函数です．ここで ‘hcp="test.ps"’ としたので “test.ps” という名前のファイルがカレントディレクトリ上にできます．ここで hcp キーワードに指定する文字列の修飾子が “.ps” であれば POSTSCRIPT 形式, それ以外ならば CGM 形式のファイルを出力します．ここで CGM 形式の画像ファイルは Yorick に附属の gist を使って表示できますが対話的な処理は行えません．この方法では画像ファイル名は Window に拘束されますが, 他には eps 函数や pdf 函数を用いることで画像をそれぞれ PostScript 形式や PDF 形式に落せます．ここでは図 10.6 に eps 函数による “test.eps” を示しておきましょう:

```
t=span(0,1,1001);
x1=2*cos(2*pi*t);
y1=2*sin(4*pi*t);
plg,y1,x1
eps,"eye.eps"
pdf,"eye.pdf"
```

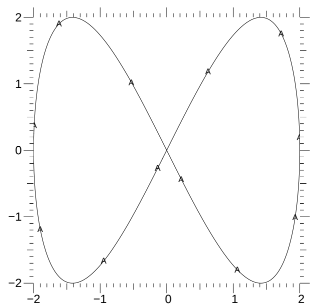


図 10.6: Lissajous 図形

このように eps 函数で “eye.eps” ファイルに PostScript 形式の画像, pdf 函数で “eye.pdf” ファイルに PDF 形式の画像がそれぞれ保存されます．

10.2　グラフの諸設定

10.2.1　表示 Window の設定

ここでは描画用 Window の処理を行う函数について述べます.

表示 Window の設定を行う函数

構文 (fma, window, winkilll, current_window)
fma
window, ⟨ 整数 ⟩, ⟨ キーワード ⟩
winkill,⟨ 整数 ⟩
current_window()

fma 函数:　描画用 Window の初期化函数で引数を必要としません. Window がない状態で fma 函数を実行すれば名前が "Yorick 0" の Window を生成し, そうでなければ利用中の Window の初期化を行います.

window 函数:　描画用 Window の生成・切替を行います. 第 1 引数が Window 番号で 0 から 63 までの整数が指定できます. また Window 番号は current_window 函数で入手できます.
window 函数のキーワードは第 2 引数以降に記号 " , " で区切った式で指定します. ここでの式はキーワードと呼ばれる語句とその値を演算子 "=" で繋いだ 'dpi=75' のような式になります. ここでは window 函数の重要なキーワードについてのみ解説しておきます:

- dpi, width と height:
  画像の dpi(dot per inch) を整数値で指定します. 標準は 75dpi, 450 × 450 の画素の Window で, dpi を 100 に指定すると画像の大きさは 600 × 600 になります. ここで $m \times m$ 画素の大きさの画像を得るためには $m/6$ dpi を指定します. また width と height は Window の大きさを指定するだけで画像の実際の大きさと無関係なので dpi と合せて利用すべきです.

- display:
  Window の表示先を指定します. 表示先の通常の書式は, X11 の環境変数 DISPLAY の書式の「ホスト名:サーバー. 画面」と同じものになりますが, ここで "display=""" を指定すると Yorick は Window を表で見える形で開きません. この状態で hcp にファイルを指定し, グラフ表示の函数を通常と同様に実行さ

せてから hcp 函数を実行すれば POSTSCRIPT ファイルに画像が出力されます．この処理を通じて端末側に画像が一切表示されません．この指定は X が使えないリモート接続で Yorick を利用しているときに画像ファイルの生成が必要な場合，あるいは画像表示 Window を表示せずに画像ファイルのみを生成したい場合には hcp キーワードのファイル指定と一緒にに「"""」を display キーワードに指定しましょう．

- hcp:
  hcp 函数で画像保存するファイルを指定します．ここでファイル名の修飾子が ".ps" の場合に限って PostScript 形式で画像が保存され，それ以外は CGM 形式で保存されます．この CGM 形式の画像は Yorick に付属の gist で表示することができます．

- style:
  Window の外観を決定します．この外観は Yorick のホームディレクトリの g ディレクトリにある修飾子が ".gs" のファイルで設定されています．現在，axes.gs, boxed.gs, boxed2.gs, nobox.gs, l_nobox.gs, vbox,gs, spydr.gs, spydr2.gs, work.gs, work2.gs, vg.gs があります．ここでは幾つか代表的なものを示しておきましょう：

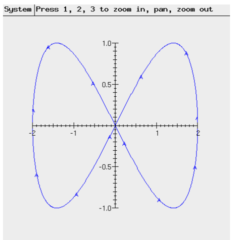


図 10.7: style=axes 場合

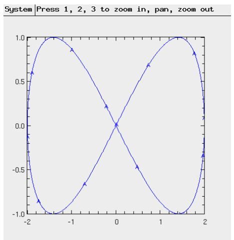


図 10.8: style=boxed の場合

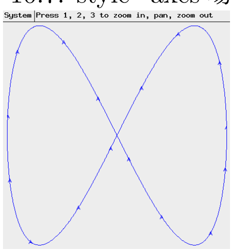


図 10.9: style=nobox の場合

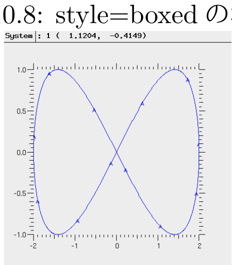


図 10.10: style=work の場合

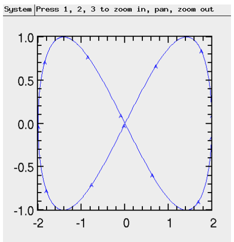


図 10.11: style=vgbox の場合

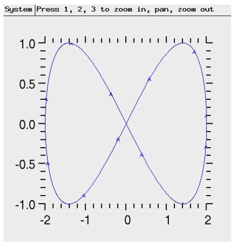


図 10.12: style=vg の場合

- legends:
  legend=1 が標準であり, legend=0 の場合に凡例はハードコピーファイルに保存されません.
- wait:
  wait=1 で Window を表示するまで Yorick 本体側の動作を止めます.

winkill 函数:　不要となった Window を削除します. 引数は Window 番号を 1 つだけ取ります.

current_window 函数:　引数を取らない函数で, “current_window()” で用います. この函数はその時点の描画用 Window 番号を返す函数で, 描画用 Window が開かれていなければ ‘-1’ を返します:

```
> current_window()
−1
> window,2
> window,3
> current_window()
3
> window,2
> current_window()
2
```

ここで示すように, Window がない状態では current_window 函数は-1 を返しています. それから, Window 番号 2 と 3 番を window 函数で生成しています. ここで, の window 函数で Window を生成すると描画 Window は新規に生成した Window になります. そして, window 函数で既存の Window 番号を指定することで, 指定した番号の Window が描画 Window になります.

10.2.2 表示領域の指定

表示 Window の設定を行う函数

構文 (logxy, limits, range)
logxy,⟨0 または 1⟩,⟨0 または 1⟩
limits, ⟨ 最小値$_X$⟩,⟨ 最大値$_X$⟩, ⟨ 最小値$_Y$⟩,⟨ 最大値$_Y$⟩, ⟨ キーワード ⟩
limits, ⟨ 最小値$_X$⟩,⟨ 最大値$_X$⟩, ⟨ キーワード ⟩
limits,⟨ キーワード ⟩
range, ⟨ 最小値$_Y$⟩,⟨ 最大値$_Y$⟩

logxy: 引数として 2 変数を取り, 引数は 0 か 1 の何れかになります. 第 1 引数が 1 の場合に X 軸が対数目盛, 第 2 引数が 1 の場合に Y 軸が対数目盛になります.

```
t = span(0,1,1001);
logxy,0,1
fma;plg,exp(t),t
```

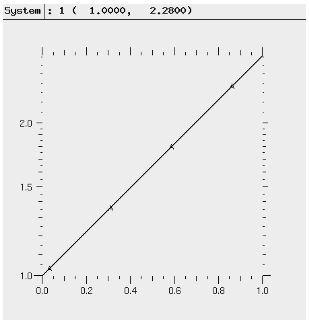


図 10.13: Y 軸のみを対数目盛にした例

limits 函数: グラフの大きさの自動調整やグラフを指定した領域で表示させる作用があります. limits 函数で数値引数を 2 つだけ指定すれば X 軸側のみの領域指定となって Y 軸側がグラフが Window に収まるように設定されます. ここで range 函数は Y 軸側の指定可能で, この limits 函数の引数 2 つの場合に対応します.
引数を指定しなければグラフが Window にきちんと収まるように再描画させるために使えます. この limits 函数のキーワードには次のものがあります:

- square
  画面の縦横比を等しくするときに 'square=1', そうでない場合に 'square=0' とします. 既定値は 'square=0' です.

- nice
  キーワード square の影響を受け, ‘nice=0’ であればグラフを Window 一杯に表示させ, ‘nice=1’ であれば多少隙間を空けます.

- restrict
  ‘0’ か ‘1’ の値を取ります.

range 函数:　Y 軸側の描画領域を指定して再描画を行う函数です. limits 函数の引数 2 つの場合のものに対応する函数ですが, limits 函数と違いキーワードを持ちません.

10.2.3　階調を指定する函数

階調を設定する函数として palette 函数があります:

palette 函数の構文

palette,⟨ 文字列 ⟩
palette,⟨ 正整数 ⟩
palette,⟨ ベクトル$_1$⟩,⟨ ベクトル$_2$⟩,⟨ ベクトル$_3$⟩,ntsc=1/0
palette,⟨ ベクトル$_1$⟩,⟨ ベクトル$_2$⟩,⟨ ベクトル$_3$⟩,query=1
palette,⟨ ベクトル$_1$⟩,⟨ ベクトル$_2$⟩,⟨ ベクトル$_3$⟩,⟨ ベクトル$_4$⟩
palette,⟨ ベクトル$_1$⟩,⟨ ベクトル$_2$⟩,⟨ ベクトル$_3$⟩,⟨ ベクトル$_4$⟩,query=1

palette 函数:　Yorick には幾つかの階調が予め用意されています. これらの階調はファイルで “Y_SITE/g” で指示されるディレクトリに収録されており, これらの階調を利用する場合はファイル名を文字列として palette 函数の引数として引渡します. ここで指定可能な階調ファイルを次に纏めておきます:

階調ファイルの概要

ファイル名	概要
earth.gp	標準で用いられる地図に似せた階調で, 黒→青→緑→黄色→白と階調が変化.
gray.gp	黒→白と線形に変化.
heat.gp	熱した鉄棒の温度を表現し, 赤→橙→黄色→白と変化.
ncar.gp	白→紫→赤→黄色→緑→青とちょっと不思議な階調.
rainbow.gp	赤→橙→緑→青→紫と変化.
stern.gp	IDL 由来の Stern Special
yarg.gp	gray.gp の逆で, 白→黒と線形に変化.

この階調は簡単な次のスクリプトで検証しするとよいでしょう:

```
x=span(-1,1,1001)(,-:1:1001,)
y=transpose(span(-1,1,1001)(,-:1:1001,))
z=255*cos(2*pi*(x^2+y^2))
p=["gray.gp","heat.gp","ncar.gp","rainbow.gp","stern.gp","yarg.gp"];
fma;
for(i=1;i<7;i++){palette,p(i);pli,z;
rdline,prompt=p(i)+". Please press any key."};
```

このスクリプトでは $255\cos 2\pi(x^2+y^2)$ のグラフを領域 $[-1,1]\times[-1,1]$ で描きます. ここの函数は原点で 255, 半径 $1/\sqrt{2}$ の円で-255 と, これらが最高点と最低点で, 階調もこの最高点と最低点に合せられます. 上述のスクリプトを入力すれば階調ファイル名と「Please press any key.」が表示され, そこで適当なキーを押せば次の階調表示を行います.
ここで階調ファイルの構造は次に示す形式となっています:

$$
\begin{array}{lll}
\texttt{ncolors} = \langle n \rangle & & \\
\langle \texttt{ntsc = 1} \rangle & & \\
\langle \mathrm{Red}_1 \rangle & \langle \mathrm{Green}_1 \rangle & \langle \mathrm{Blue}_1 \rangle \\
\vdots & \vdots & \vdots \\
\langle \mathrm{Red}_{\langle n \rangle} \rangle & \langle \mathrm{Green}_{\langle n \rangle} \rangle & \langle \mathrm{Blue}_{\langle n \rangle} \rangle \\
\langle \mathrm{Gray}_{\langle n \rangle} \rangle & &
\end{array}
$$

ここで $\langle n \rangle$ は正整数, $\langle \mathrm{Red}_i \rangle$, $\langle \mathrm{Green}_i \rangle$, $\langle \mathrm{Blue}_i \rangle$, $\langle \mathrm{Gray}_i \rangle$ は $\langle n \rangle$ 個の成分の整数ベクトルで, 各々, 赤, 青, 緑と灰色の 0 から 255 までの整数値を階調として取ります. 赤, 青, 緑は通常のカラー出力が行える環境向けの出力で省略はできません. 灰色は白黒プリンタ等の白黒機器への出力で用いられる階調で, こちらは省略しても構いません. またキーワードの ntsc の設定も行えます. このキーワード ntsc は白黒テレビ出力向けの設定で, 'ntsc=0' であれば階調は赤, 緑, 青の平均を取り, 'ntsc=1' であれば '0.30*赤+0.59*緑+0.11*青' で与えられます. そのために 'ntsc=1' とする場合のみ書込みます. そして, ファイルの注釈行は記号 "#" で開始し, 改行で終了します.
自分で階調を定義する場合, 赤, 青, 緑の階調, あるいは赤, 青, 緑, 灰色の階調を数値ベクトルで指定します. このとき各ベクトルは同じ長さでなければなりません.

```
> x=span(-1,1,1001)(,-:1:1001)
> y=transpose(span(-1,1,1001)(,-:1:1001))
> z=cos(2*pi*(x^2+y^2))
```

```
> window,1;pli,z;
> r=b=span(1,255,255)
> g=span(1,255,255)(::−1)
> palette,r,g,b
> window,2;pli,z;palette,1
```

この例では階調を赤と青を 1 から 255, 緑を 255 から 1 に設定しています．この結果,「紫→緑」となる階調が得られますが階調は Window 単位で独立しています．そこで他の Window で別の Window で定義した階調を利用したければ Window 番号を palette 関数の引数とします．この例では番号 1 の Window で定義した階調を番号 2 の Window 上の描画に適用させています.

10.2.4　3 次元表示に関連する函数

Yorick では 3 次元表示を行う上で便利な函数を幾つか持っています．ただし, 2 次元表示を基本としているので, ここで紹介する函数も大本は前述の函数を母体としています:

3 次元表示に関連する函数

構文 (orient3, window3, cage3, limit3)
orient3,$\langle\phi\rangle$, $\langle\theta\rangle$
orient3,$\langle\phi\rangle$
orient3, ,$\langle\theta\rangle$
orient3
window3, ⟨ 整数 ⟩
window3
cage3,on/off
cage3
limit3,$\langle X_{min}\rangle$,$\langle X_{max}\rangle$, $\langle Y_{min}\rangle$,$\langle Y_{max}\rangle$
limit3,$\langle X_{min}\rangle$,$\langle X_{max}\rangle$, $\langle Y_{min}\rangle$,$\langle Y_{max}\rangle$, $\langle Z_{min}\rangle$,$\langle Z_{max}\rangle$

orient3 函数:　視点を Z 軸回りの ϕ と Y 軸回りの θ で制御します．角度は弧度法で与えます.

window3 函数:　3 次元表示のために ‘style=”nobox.gs”’ で Window の初期化を行う函数です．引数の意味は window 函数と同様です.

cage3 函数: 3 次元グラフ用の目盛を表示するかどうかを指定する函数です. 引数がなければ, その時点の状態の逆に切換えます. つまり cage3 で目盛の表示・非表示が切換えられます.

limit3 函数: 3 次元版の limits 函数で表示領域を指定する函数です. ただし limits 函数のような引数なしで Window の初期化は行いません. 2 次元と同様に 3 次元表示でも Window の初期化は引数なしの limits 函数で行えます.

10.3 描画函数のキーワード

Yorick にはさまざまな描画函数がありますが, 描画函数のキーワードには共通するものがあります. ここでは代表的なキーワードについて述べます.

- color と ecolor: color で描画する曲線の色を指定し, ecolor で曲面の稜線の色を指定します. 名前で指定可能な色は:

  名前で指定可能な色

名前	black	white	red	green	blue	cyan	magenta	yellow
番号	-3	-4	-5	-6	-7	-8	-9	-10

  color が利用可能な函数は plg, plm, plc, pldj と plt で, ecolor が利用可能な函数は plf と plwf です.

- width と ewidth: width で描画する曲線の太さを数値で指定し, ewidth で曲面の稜線の太さを指定します. 標準で共に 1.0 であり, 大きな数値を指定するとより太い曲線が得られます. width が利用可能な函数は plg, pldj, plm, plv, plc で, ewidth が利用可能な函数は plf と plwf です.

- type: 曲線の種類を指定します. 指定可能な種類を纏めて置きましょう:

  名前で指定可能な曲線の種類

名前	solid	dash	dot	dashdot	dashdotdot	none
番号	0	1	2	3	4	5

  solid が通常の実線, dash が破線, dot が点線, none が非表示となります. 曲線の種類は上の名前でも下の番号でも指定ができます.

このキーワードが利用可能な函数は, plg, plm, plc, pldj と plv ですが, ここで plv 函数は hollow=1 の場合に限定されます.

- legend: 凡例には ‘legend=”test”’ のように文字列を指定します. ただし hcp で生成したグラフファイルのみに追加され, X-Window System 側のグラフには表示されません. このキーワードが利用可能な函数は plg, plm, plc, plv, plf, pli, plt と pldj です.
- hide: ‘hide=1’ で表示を隠します. 通常は ‘hide=0’ で曲線が表示されています.
- closed と smooth: ‘closed=1’ とすると, 曲線と始点と終点を直線で結びます. ‘closed=0’ であれば始点と終点が一致しなければ閉曲線になりません. ‘smooth=0’ であれば点の間を線分で結びます. このキーワードが利用可能な函数は plg, plm, plc, plv, plf, pli, plt と pldj です.
- rays, arrow, arrowl,rsoace と rphase: ‘rays=1’ のときに線上に矢印を描きます. ここで矢印の頭の幅は arroww, 矢印の長さは arrowl で指定します. また矢印の配置は rspace で定めます. mspace と同様に小さくするに従い矢印が増えます. 図 10.14 に ‘rays=1, arrowl=2, arroww=2, rspace=1’ を指定したグラフを示しておきます:

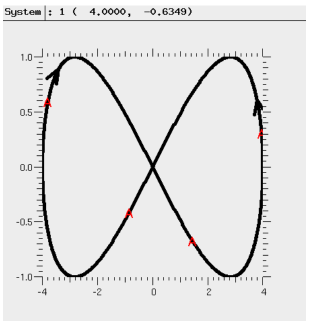


図 10.14: plg,y1,x1,rays=1,arrowl=2,arroww=2,rspace=1,width=10

これらのキーワードが利用可能な函数は plg と plc です.

- marks,marker,mcolor, msize, mspace と mphase: ‘marks=1’ で曲線の印字を表示, ‘marks=0’ で消します. marker には ASCII 文字や数字を指定し, marker に指定した対象を曲線の印字として表示します. mcolor で曲線の印字の色を指定

し, その色の指定は color と同様です. また msize で印字の大きさを指定します. ここでの大きさは拡大率です. mspace は印字の数に関係し, 'mspace=1' で 1 つだけ印字が表示され, それよりも大きくすると印字は消えます. これらのキーワードが利用可能な函数は plg と plc です.

- shade と edges: shade は曲面の描画を行うかどうかを指定し, shade=1 で曲面の描画を行い, shade=0 で曲面の描画を行いません. edges は曲面の稜線を描くかどうかを指定し, edges=1 で稜線の描画を行い, edges=1 で稜線の描画を行います. これらのキーワードが使える函数では, 2 次元的な可視化に加えて 3 次元的な可視化が可能です. edges=1 は描画点数が多い与件の 2 次元的な可視化では稜線で曲面が埋め尽され欠点があります. ただし, 図 10.16 のような 3 次元的な可視化では "shade=0, edges=1" としても 2 次元の場合程の問題にはならないでしょう. ただし, 一般的には図 10.15 の方が余計な線がない分, 好まれるでしょう.

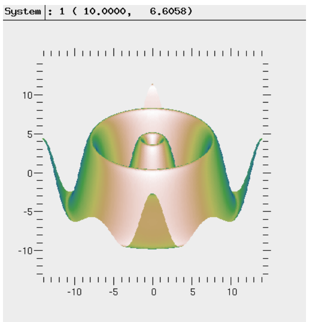


図 10.15: plwf,shade=1,edges=0

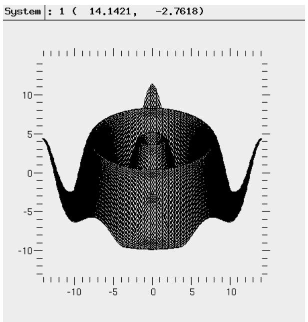


図 10.16: plwd,shade=0,edges=1

これらのキーワードが利用可能な函数は plf と plwf 函数です.

- triangle: 等高線表示での配列の三角形分割の指定を行います. このキーワードが利用可能な函数は plc 函数のみです.
- region: 網目の領域を選択します. このキーワードが利用可能な函数は plm, plc, plv と plf です.
- scale: 2 次や 3 次元配列を 3 次元表示する際の Z 軸方向の倍率を与えます. 標準で "scale=1.0" となっており, scale に与えた数値倍の値でグラフ表示されま

す．そのため, Z 座標も読み取る必要があれば, Z 軸側の目盛を調整しなければなりません．このキーワードが利用可能な函数は plwf のみです．

10.4　グラフ表示を行う函数

10.4.1　plg 函数

通常の函数グラフは plg 函数を用いて行います:

plg 函数の構文

plg,⟨ リスト$_{\mathtt{Y}}$⟩,⟨ リスト$_{\mathtt{X}}$⟩, ⟨ キーワード ⟩
plg,⟨ リスト$_{\mathtt{Y}}$⟩,⟨ キーワード ⟩

与えられたリストの描画を行います．第 1 引数が Y 座標を与え, 第 2 引数が X 座標を与えます．第 2 引数は第 1 引数と同じ大きさでなければなりませんが, この第 2 引数は省略可能で, 省略した場合の X 座標は 1 から第 2 引数の個数までの整数列で与えられます．plg 函数は描画用 Window が開かれていなければ, “window,0” を実行して描画を行います．複数のグラフの描画を行う場合, fma 函数で Window の初期化を行わずに, そのまま plg 函数を使って重ね描きします．この plg 函数のキーワードとしては次のものがあります:

plg 函数のキーワード

legend	hide	type	width	color	closed	smooth	marks
marker	mspace	mphase	rays	arrowl	arroww	rspace	rphase

10.4.2　plc 函数

2 次元配列 X と 2 次元配列 Y に対応する 2 次元配列 Z で与えられる高さに対して等高線を描きます:

plc 函数の構文

plc,⟨ 配列$_{\mathtt{z}}$⟩,⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩, ireg, levs=⟨ 数値$_{z}$⟩, ⟨ キーワード ⟩
plc,⟨ 配列$_{\mathtt{z}}$⟩,⟨ 配列$_{\mathrm{Y}}$⟩, ⟨ 配列$_{\mathrm{X}}$⟩,levs=⟨ 数値$_{z}$⟩, ⟨ キーワード ⟩
plc,⟨ 配列$_{z}$⟩, levs=⟨ リスト ⟩, ⟨ キーワード ⟩

ここで ⟨ 配列$_{\mathtt{Z}}$⟩ は ⟨ 配列$_{\mathtt{Y}}$⟩ や ⟨ 配列$_{\mathtt{X}}$⟩ と同じ大きさの配列でなければなりません．そして, ⟨ 配列$_{\mathtt{Z}}$⟩(i,j) は ⟨ 配列$_{\mathtt{X}}$⟩(i,j) と ⟨ 配列$_{\mathtt{Y}}$⟩(i,j) から得られる函数値となります．

そして等高線は描くべき高さを ⟨ リスト ⟩ として与えます. たとえば, “levs=[1,2,3]” とすると, 等高線は高さ 1, 2, 3 に描かれ, 左から順に印字付けられます.
簡単な例として, $[-10,10] \times [-10,10]$ の領域で $\sin\left(\sqrt{x^2+y^2}\right)$ を描いてみましょう.

```
> X1 = span(−10,10,201)(,−:−100:100);
> Y1 = transpose(span(−10,10,201)(,−:−100:100));
> fma;plc,sin(sqrt(X1ˆ2+Y1ˆ2)),Y1,X1;
> fma;plc,sin(sqrt(X1ˆ2+Y1ˆ2)),Y1,X1,levs=[0.1,0.9],\
cont> color=”red”,width=5,msize=2
```

この例では最期の行入力が長くなるために入力の継続を示す記号 “\” を行末尾に入力したために入力が継続していることを示すプロンプロ “`cont>`” が現われています. ここで X1 は span(-10,10,201) を $(:,i)_{i=1..201}$ 成分とする 2 次配列, Y1 は span(-10,10,201) を $(i,:)_{i=1..201}$ 成分とする 2 次配列とし, 図 10.17 はキーワードを指定しない場合, 図 10.18 は等高線の高さ, 太さと印字の大きさを指定した場合を示しています:

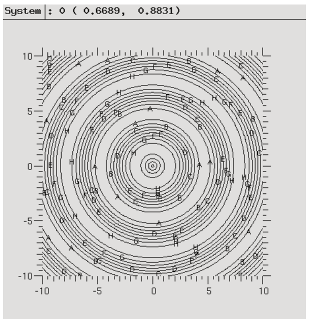


図 10.17: plc のグラフ

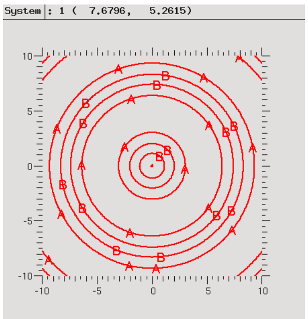


図 10.18: plc で levs の指定を行った場合

plc 函数のキーワードを次に示しておきます:

plc 函数のキーワード

legend	hide	color	smooth	marks	marker	mspace	mphase
triangle	region						

plg 函数と共通するものを多く含んでいますが triangle と region は plg 函数にはありません.

10.4.3　plfc 函数

2 次元配列 X と 2 次元配列 Y に対応する 2 次元配列 Z で与えられる高さに対して等高線を描く函数で, plc 函数に pli 函数風の味付けが加味されています:

plfc 函数の構文

plfc,⟨ 配列$_Z$⟩,⟨ 配列$_Y$⟩, ⟨ 配列$_X$⟩, ireg, levs=⟨ 数値リスト ⟩, ⟨ キーワード ⟩
plfc,⟨ 配列$_Z$⟩,⟨ 配列$_Y$⟩, ⟨ 配列$_X$⟩,levs=⟨ 数値リスト ⟩, ⟨ キーワード ⟩

ここで ⟨ 配列$_Z$⟩ は ⟨ 配列$_Y$⟩ や ⟨ 配列$_X$⟩ と同じ大きさの配列でなければなりません. そして, ⟨ 配列$_Z$⟩(i,j) は ⟨ 配列$_X$⟩(i,j) と ⟨ 配列$_Y$⟩(i,j) から得られる函数値となります. そして, 描くべき高さを ⟨ リスト ⟩ として与えます. たとえば, “levs=[1,2,3]” とすると, 等高線は高さ 1, 2, 3 に描かれ, 左から順に印字付けられます.
簡単な例として, $[-10, 10] \times [-10, 10]$ の領域で $\sin\left(\sqrt{x^2 + y^2}\right)$ を描いてみましょう.

```
> X1 = span(−10,10,201)(,−:−100:100);
> Y1 = transpose(span(−10,10,201)(,−:−100:100));
> fma;plfc,sin(sqrt(X1^2+Y1^2)),Y1,X1;
> fma;plfc,sin(sqrt(X1^2+Y1^2)),Y1,X1,levs=[0.1,0.9],\
cont> color="red",width=5,msize=2
```

この例でも最期の行入力が長くなるために入力の継続を示す記号 “\” を行末尾に入力したために入力が継続していることを示すプロンプロ “`cont>`” が現われています. ここで, X1 は span(-10,10,201) を $(:, i)_{i=1..201}$ 成分とする 2 次配列, Y1 は span(-10,10,201) を $(i, :)_{i=1..201}$ 成分とする 2 次配列で, 図 10.19 ではキーワードを指定しない場合, 図 10.20 では等高線の高さ, 太さと印字の大きさを指定した場合になります:

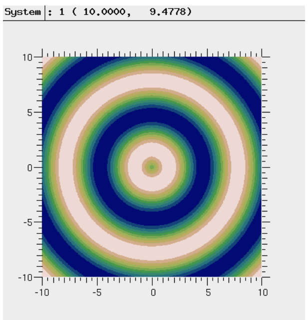


図 10.19: plfc のグラフ

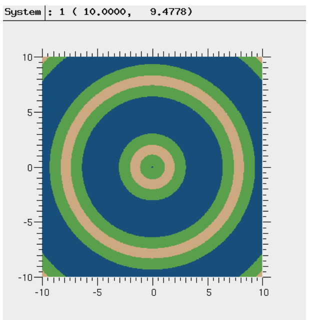


図 10.20: plfc で levs の指定を行った場合

plfc 函数のキーワードを次に示しておきます:

plfc 函数のキーワード

triangle	region

10.4.4 pli 函数

2 次元配列をそのまま表示する函数です. この函数は Yorick で読込んだ画像の表示に使えます:

pli 函数の構文

pli,⟨ 配列$_{\mathtt{Z}}$⟩, ⟨$\mathtt{X_0}$⟩, ⟨$\mathtt{Y_0}$⟩, ⟨$\mathtt{X_1}$⟩, ⟨$\mathtt{Y_1}$⟩, ⟨ キーワード ⟩
pli,⟨ 配列$_{\mathtt{Z}}$⟩, ⟨$\mathtt{X_1}$⟩, ⟨$\mathtt{Y_1}$⟩, ⟨ キーワード ⟩
pli,⟨ 配列$_{\mathtt{Z}}$⟩, ⟨ キーワード ⟩

表示する 2 次元配列を ⟨ 配列$_{\mathtt{Z}}$⟩ とし, キーワード以外の引数を与えなければ X 座標と Y 座標は配列の添字が利用されます. つまり配列の各成分が画素に 1 対 1 で対応します. ここで第 2 引数と第 3 引数の ⟨$\mathtt{X_0}$⟩ と ⟨$\mathtt{Y_0}$⟩ が図の左下の座標を与え, 第 4 引数と第 5 引数の ⟨$\mathtt{X_1}$⟩ と ⟨$\mathtt{Y_1}$⟩ が図右上の座標を与えます. 引数として対 (X_0, Y_0) のみを与えたときは図右上の座標は自動的に $(0,0)$ が与えられます. ここでは領域 $[-10,10]\times[-10,10]$ で $\sin\left(\sqrt{x^2+y^2}\right)$ を pli 函数を使って描いてみましょう:

```
> X1 = span(−10,10,201)(,−:−100:100);
> Y1 = transpose(span(−10,10,201)(,−:−100:100));
> fma;pli,sin(sqrt(X1^2+Y1^2))
> fma;pli,sin(sqrt(X1^2+Y1^2)),cmax=0.5,cmin=−0.8
```

キーワードなしを図 10.21, キーワード付きを図 10.22 に示しておきます:

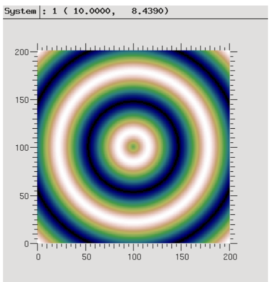


図 10.21: pli のグラフ

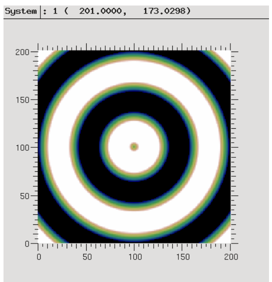


図 10.22: pli で cmax と cmin を指定

図の階調は palette で変更できますが, この階調に関連するキーワードに top があり, この top が最も明い地点の階調を定めます. この階調は 1 から 199 までの整数値が有効に働きます. 図 10.22 は ‘cmax=0.5’, ‘cmin=-0.5’ を指定した結果です. 何も指定していない図 10.21 と比較して cmax で指定した値以上が白くなり, cmin で指定した値以上が黒くなっています.

この函数は plc 函数と併用することも勿論可能です. この場合, 図の配置に注意する必要があります. 上の例の場合, 領域は $[-10, 10] \times [-10, 10]$ なので図の左下と右上の座標を指定しなければなりません:

```
fma;
pli,sin(sqrt(X1^2+Y1^2)),
      −10,−10,10,10,
      cmax=0.5,cmin=−0.5
plc,sin(sqrt(X1^2+Y1^2)),Y1,X1,
      levs=[−0.5,0.5],color=”red”,
      width=5,msize=2
```

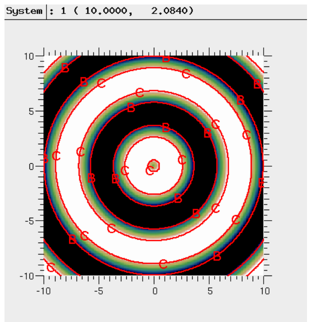


図 10.23: pli 函数と plc 函数によるグラフ

この例では pli 函数で描いたグラフの上に plc 函数で等高線を重ね描きをしています. この重ね描きで pli 函数で規準点 (X_0, Y_0) や (X_1, Y_1) を指定しなければ座標が配列の添字となり, 領域 $[-10, 10] \times [-10, 10]$ から外れてしまいます. そこで基準点として, $(X_0, Y_0) = (-10, -10)$, $(X_1, Y_1) = (10, 10)$ となるように pli 函数で指定していることに注意して下さい. plc 函数では等高線は ‘Z=-0.5’ と ‘Z=0.5’ の位置で描くように levs で指定しています. この pli 函数のキーワードを次に纏めておきます:

pli 函数のキーワード

legend	hide	top	cmin	cmax

10.4.5 plwf 函数

plwf 函数の構文

plwf,⟨ 配列$_{\mathtt{Z}}$⟩,⟨ 配列$_{\mathtt{Y}}$⟩, ⟨ 配列$_{\mathtt{X}}$⟩,⟨ キーワード ⟩
plwf,⟨ 配列$_{\mathtt{Z}}$⟩, ⟨ キーワード ⟩

引数が 1 つの配列となる場合は配列が 3 次元配列の場合に限定されます．2 次元配列であれば引数は高さ, Y 座標, X 座標をそれぞれ表現する同じ大きさの 2 次元配列を必要とし, この plwf 函数で ‘shade=1’ としたときの色彩は palette 函数で与えられます．簡単な例として, $[-10, 10] \times [-10, 10]$ の領域で $\sin\left(\sqrt{x^2+y^2}\right)$ を描いてみましょう:

```
> X1=span(−10,10,201)(,−:−100:100);
> Y1=transpose(span(−10,10,201)(,−:−100:100));
> plwf,sin(sqrt(X1^2+Y1^2)),Y1,X1,shade=1,edges=0;
> orient3;
```

この例では ‘edgs=0, shade=1’ としています．この理由は稜線を描くと点数の多いグラフは稜線だけが描かれて 2 次元表示では不明瞭になるためです．3 次元表示では ‘edge=1’ としても構いませんが, ‘shade=1’ にした方が綺麗なグラフになります．また描画用 Window を標準の設定のままであれば図 10.24 に示すように pli 函数に似た等高線のグラフが表示されます．orient3 函数を使って視点を指定すると図 10.25 に示すような 3 次元的表示に切替わります:

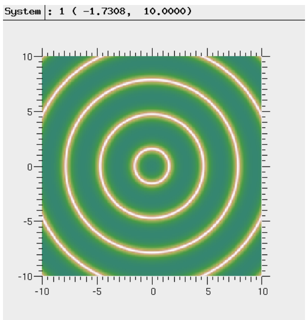


図 10.24: plwf のグラフ

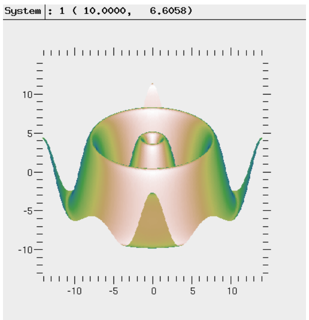


図 10.25: plwf の描画後に orient3 を実行

plwf 函数のキーワードを次に纏めておきます:

pli 函数のキーワード

fill	shade	edges	ecolor	ewidth	cull	scale	cmax

10.4.6 plm 函数

plm 函数の構文

plm,$\langle$ 配列$_{\mathtt{Y}}\rangle$,$\langle$ 配列$_{\mathtt{X}}\rangle$, boundary=0/1,inihibit=0/1/2, $\langle$ キーワード $\rangle$
plm,$\langle$ 配列$_{\mathtt{Y}}\rangle$,$\langle$ 配列$_{\mathtt{X}}\rangle$, ireg, boundary=0/1, inihibit=0/1/2, $\langle$ キーワード $\rangle$
plm, ireg, boundary=0/1, inihibit=0/1/2, $\langle$ キーワード $\rangle$

$\langle$ 配列$_{\mathtt{Y}}\rangle$ と $\langle$ 配列$_{\mathtt{X}}\rangle$ は同じ大きさの 2 次元配列となり, $\langle$ 配列$_{\mathtt{X}}\rangle$ に対する $\langle$ 配列$_{\mathtt{Y}}\rangle$ を描きます.

```
> X1 = span(−10,10,201)(,−:−100:100);
> Y1 = transpose(span(−10,10,201)(,−:−100:100));
> limits,−10,10,−2,2
> fma;plm,sin(sqrt(X1^2+Y1^2)),Y1,boundary=0
> fma;plm,sin(sqrt(X1^2+Y1^2)),Y1,boundary=1
```

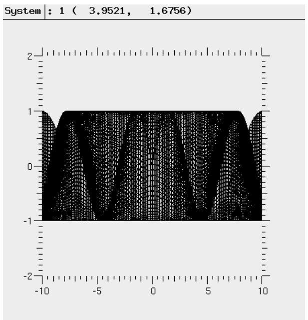


図 10.26: boundary=0 の場合

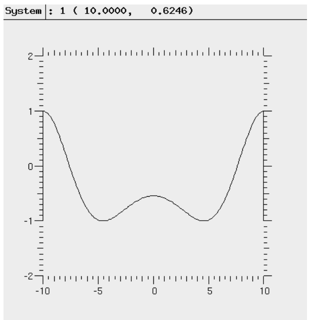


図 10.27: boundary=1 の場合

ここで inihibit についてはもう少し実例を見た方が判り易いでしょう:

- boundary=0 の場合:

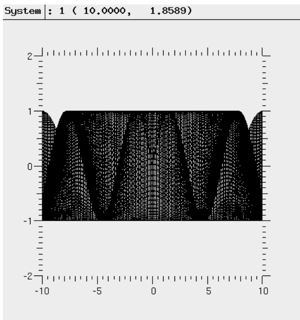


図 10.28: inhibit=0

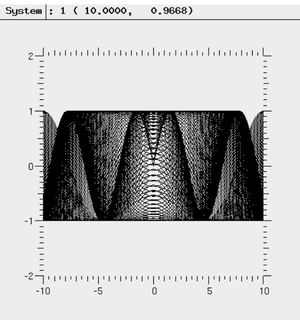


図 10.29: inhibit=1

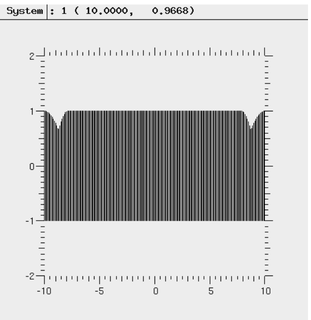


図 10.30: inhibit=2

- boundary=1 の場合:

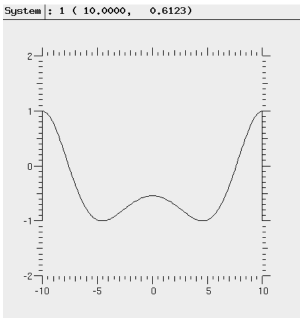


図 10.31: inhibit=0

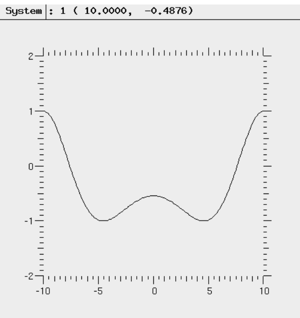


図 10.32: inhibit=1

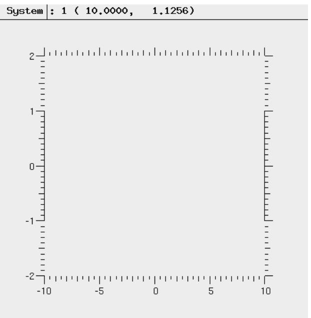


図 10.33: inhibit=2

ここで plm 函数のキーワードを次に示しておきます:

plm 函数のキーワード

legend	hide	type	width	color	region

10.4.7　plf 函数

plf 函数の構文

plf,⟨ 配列$_{\mathtt{z}}$⟩,⟨ 配列$_{\mathtt{Y}}$⟩, ⟨ 配列$_{\mathtt{X}}$⟩,⟨ キーワード ⟩
plf,⟨ 配列$_{\mathtt{z}}$⟩,⟨ 配列$_{\mathtt{Y}}$⟩, ⟨ 配列$_{\mathtt{X}}$⟩,ireg, ⟨ キーワード ⟩
plf,⟨ 配列$_{\mathtt{z}}$⟩, ⟨ キーワード ⟩

次に plf 函数のキーワードを示しておきます:

pli 函数のキーワード

legend	hide	region	top	cmin	cmax	edges	ecolor	ewidth

10.4.8　plv 函数

plv 函数の構文

plv,⟨ 配列$_{\mathtt{V_y}}$⟩,⟨ 配列$_{\mathtt{V_x}}$⟩, ⟨ 配列$_{\mathtt{Y}}$⟩,⟨ 配列$_{\mathtt{X}}$⟩, ireg, scale=⟨ 正数値 ⟩, ⟨ キーワード ⟩
plv,⟨ 配列$_{\mathtt{V_y}}$⟩,⟨ 配列$_{\mathtt{V_x}}$⟩, ⟨ 配列$_{\mathtt{Y}}$⟩,⟨ 配列$_{\mathtt{X}}$⟩, scale=⟨ 正数値 ⟩, ⟨ キーワード ⟩
plv,⟨ 配列$_{\mathtt{V_y}}$⟩,⟨ 配列$_{\mathtt{V_x}}$⟩, scale=⟨ 正数値 ⟩, ⟨ キーワード ⟩

この plv 函数のキーワードとしては次のものがあります:

plg 函数のキーワード

legend	hide	type	color	smooth	marks	marker	mspace
mphase	triangle	region					

10.4.9　pldj 函数

pldj 函数は非常に原始的な函数で, 指定した二点間をで結ぶ線分を描く函数です:

pldj 函数の構文

pldj,⟨$\mathtt{X_0}$⟩, ⟨$\mathtt{Y_0}$⟩, ⟨$\mathtt{X_1}$⟩, ⟨$\mathtt{Y_1}$⟩, ⟨ キーワード ⟩

始点の座標は ⟨$\mathtt{X_0}$⟩, ⟨$\mathtt{Y_0}$⟩, 終点の座標は ⟨$\mathtt{X_1}$⟩, ⟨$\mathtt{Y_1}$⟩ で与えられます. ここで pldj 函数のキーワードを次に示しておきます:

pldj 函数のキーワード

legend	hide	type	width	color

10.4.10　plfp 函数

plfp 函数の構文

plfp,⟨ リスト `Z`⟩, ⟨ リスト `Y`⟩, ⟨ リスト `X`⟩, ⟨ 整数 ⟩, ⟨ キーワード ⟩

ここで plfp 函数のキーワードを次に示しておきます:

plfp 函数のキーワード

legend	hide	top	cmin	cmax	edges	ewidth	ecolor

10.4.11　plt 函数

描画 Window 上の指定した場所に文字を表示させる函数です:

plt 函数の構文

plt,⟨ 文字列 ⟩, ⟨`X`⟩, ⟨`Y`⟩, tosys=1
plt,⟨ 文字列 ⟩, ⟨`X`⟩, ⟨`Y`⟩, tosys=0

この函数は ‘tosys=1’ の場合に限って Window 上の指定した位置に文字を表示します. ここで記号“ ^ ”, 記号“ _ ” と記号“ ! ” は TeX や GNUPLOT の場合と同様の特殊な意味を持ちます. 記号“ ^ ” は後続の文字を上付きに, 記号“ _ ” は後続の文字を逆に下付きにします. なお, TEX と違うのは, その効果が記号の後続の全ての文字に影響する点です. そこで一度上付きにした場合に途中から本来の位置に戻すためには“ __ ” を挿入し, 下付きを本来の位置に戻す場合には“ ^^ ” を挿入します. 一方で“ ! ” は二つの意味があり, 一つが記号“ ^ ” と記号“ _ ” の直前に置いて, これらの記号の作用を無効にすること, もう一つがアルファベットに付けることでフォントを symbol に切換えることです:

```
fma;
plt,”sin^2__!Q_i”,0,0,tosys=1;
plt,”eq!_1=x!^2+y!^2−8*x*y”,3,3,tosys=1;
```

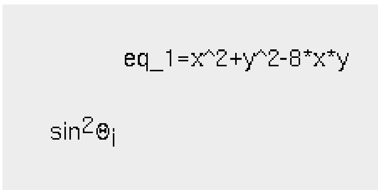


図 10.34: plt による文字列の表示

10.5 動画機能

Yorick では動画機能として原始的な函数を幾つか擁している程度で, demo2,demo3,demo4 を参照することが薦められています. 詳細はこれらのデモファイルを参照して下さい.

第11章　数列あそび

First Clown. [sings]	第一の墓掘人 [歌う]
But age, with his stealing steps, Hath claw'd me in his clutch, And hath shipped me intil the land, As if I had never been such.	それはそうと年波は忍び足, むんずと俺を捕まえて, ここへと補陀落したが, こんな俺じゃあなかったのに.
	Hamlet: 第五幕, 第一場

11.1　Fibonacci 数

ここでは次の有名な数列から始めましょう.

Fibonacci 数

F_0	$=$	0
F_1	$=$	1
F_{n+1}	$=$	$F_n + F_{n-1}$

この数列を計算するためには Yorick でどのようにプログラムを記述すればよいのでしょうか？ そこで帰納的な定義をそのまま利用してプログラムを作ってみましょう;

```
func Fib(n){
 local ans;
  if (n==0)ans=0.;
  if (n==1)ans=1.;
  if (n>1) ans=Fib(n−1)+Fib(n−2);
 return  ans;};
```

このプログラムでは函数 Fib の引数 n の値が 0 の場合, 1 の場合と 1 より大の場合に分けて, Fibonacci 数の帰納的な定義をそのまま実装しています. このプログラムの特徴として, 引数が ‘0’ と ‘1’ の場合の返却値として浮動小数点数の ‘0.’ と ‘1.’ を指定している点と函数内部で自分自身を呼出している点が挙げられます. 前者は Fibonacci 数列が早い段階で Yorick の整数の上限を超過する可能性があるためです. 実際, 64 bit OS の環境であれば $F_{103} = 6334266236422402381$ の計算はできても F_{104} で整数の上限を超過し. 32 bit 環境であれば F_{45} で上限を超過します. また函数内部で自分自身を呼出すという再帰的なプログラムにより Fibonacchi 数列の定義をそのままプログラムに書き写すことができていますが, この難点は効率の悪さです[1]. 実際に函数 Fib で n が ‘10’, ‘20’, ‘30’ と ‘40’ の場合を計算させてみましょう;

```
> A1=array(double,3)
> A2=array(double,3)
> timer,A1;Fib(10);timer,A2;A2−A1
55
 [0,0,6.8903 e−05]
> timer,A1;Fib(20);timer,A2;A2−A1
6765
 [0.004,0,0.0055449]
```

[1]MATLAB や Octave でも Fib と同様のプログラムが構築できますが, この方法の効率は Yorick と同様に良くありません.

```
> timer,A1;Fib(30);timer,A2;A2-A1
832040
[0.684043,0,0.754838]
> timer,A1;Fib(40);timer,A2;A2-A1
1.02334e+08
[86.7374,0.232014,94.1504]
```

‘n=30’ までは我慢できるでしょうが, ‘n=40’ は流石に辛いでしょう. ‘A2-A1’ の第 1 成分の CPUTIME で比較すると明瞭ですが, ‘n=40’ では ‘n=30’ の場合の 124 倍もの CPUTIME を消費しています. どうして遅いのでしょうか？ ここで手計算の手順を思い出して下さい. ‘n=40’ のときの Fibonacci 数を計算するために ‘n=39’ の場合と ‘n=38’ の場合を別々に計算するのではなく, ‘n=0’ と ‘n=1’ の結果から ‘n=2’ を計算し, 以降, 中途の計算結果を積み上げて ‘n=40’ の計算をするでしょう. ところが函数 Fib の方法は ‘Fib(m)’ を計算するために ‘Fib(m-1)’ と ‘Fibm-2’ の計算に分岐し, その上, これらの計算は互いに無関係でさらに分岐するという鼠算的に計算量が増大する処理になっていますね. 今度は手計算の手順を Fib2 に実装してみましょう;

```
func Fib2(n){
 local F0,F1,tmp;
 F0=0.; F1=1.;
  if (n==0) F1=F0;
 for ( i=2;i<=n;i++){tmp=F1; F1=F0+F1; F0=tmp;};
 return(F1);};
```

この Fib2 で処理時間はどれほど改善されているでしょうか？

```
> timer,A1;Fib2(10);timer,A2;A2-A1
55
[0,0,2.09808 e-05]
> timer,A1;Fib2(20);timer,A2;A2-A1
6765
[0,0,2.59876 e-05]
> timer,A1;Fib2(30);timer,A2;A2-A1
832040
[0,0,2.59876 e-05]
> timer,A1;Fib2(40);timer,A2;A2-A1
1.02334e+08
> timer,A1;Fib2(1000);timer,A2;A2-A1
4.34666e+208
[0,0,0.000206947]
```

このように実用的な速度になりました. 一般的に再帰的な定義をそのままプログラム

に用いると極端に速度が低下することが多いために注意が必要です[2].

11.1.1 整数の拡大

Fib2 の返却値の表示は n がある程度大ききなると浮動小数点数の指数表現に切替わり, おおよその値しか判らなくなります. 実際の精度は表示よりもあって, 小数点よりも上の桁が 15 桁よりも小さなければ write 函数を併用することでより正確な値の表示ができます;

```
> Fib2(50);
1.25863e+10
> write,format="%10.0f\n",Fib2(50)
12586269025
```

しかし結果が 15 桁以上になると write 函数による表示も期待できなくなります;

```
> write,format="%16.0f\n",Fib2(73)
 806515533049393
> write,format="%16.0f\n",Fib2(73)*2.0
1000000000000000
> Fib2(1000)
4.34666e+208
> write,format="%210.0f\n",Fib2(1000)


  1000000000000000
```

このように write 函数の書式指定では, 表示する浮動小数点数の小数点よりも上の桁が 15 桁までの数値の表示しかできません. さらに大きな数になると浮動小数点数の固有の問題も出てきます. すなわち, 倍精度浮動小数点数の内部表現で仮数部の最も小さな数値は 2^{-52} です. だから指数部が $2^{52} = 9007199254740992$ 以下であれば浮動小数点数間の距離 (ulp) も 1 以下なので 2^{53} 以下の整数は一意に倍精度浮動小数点数で表現できます. ところが 2^{53} よりも大きな整数では 2 つの浮動小数点数の距離が 1 を越えてしまうので浮動小数点数では一意に表現できません. このことを 64 bit 環境の Yorick で確認してみましょう;

```
> 2^53==2.0^53
1
> 2^53==2^53+1
0
> 2^53-1==2.0^53
```

[2] この現象は言語の仕様に依存します. たとえば Haskell では極端な遅延は生じないようです.

```
0
> 2^53==2.0^53+1.0
1
```

整数 2^{53} と浮動小数点数 '2.0^53' は等しいと判断し, 整数 $2^{53}-1$ と浮動小数点数 '2.0^53' や整数 2^{53} と整数 $2^{53}+1$ が異なることも判断できています. ところが整数 2^{53} と浮動小数点数 '2.^53+1' の区別はできていませんね. この様子を Yorick で見たいものですが表示桁の制約があるので, ここでは Octave を使って確認しましょう. MATLAB や Octave では fprintf 函数の書式設定で浮動小数点数を整数の書式で Yorick よりも自由に表示させることができます;

```
octave:17> fprintf('%16.0f\n',2.0^53-1);
9007199254740991
octave:18> fprintf('%16.0f\n',2.0^53);
9007199254740992
octave:19> fprintf('%16.0f\n',2.0^53+1);
9007199254740992
```

この最後の例で判るように $2^{53} = 9007199254740992$ よりも大きな次の浮動小数点数で表現できる整数が $2^{53}+2 = 9007199254740994$ のために整数 $2^{53}+1 = 9007199254740993$ は $2^{53} = 9007199254740992$ に丸められています. そんな訳で 16 桁以上の Fibonacci 数の正確な値は浮動小数点数を利用する限りは判らないのです. そこで, より大きな整数を扱いたければどうすればよいでしょうか？1 つの方法は 64 bit 環境で作業することです. こうすることで $[-2^{63}, 2^{63}-1]$ の範囲の整数が扱えますが, 任意の桁ではありません. もう 1 つの方法は多倍長整数を持つ数式処理システムを使うことです. そこで数式処理システムの Maxima を使ってみましょう. では Maxima が立上がった時点で次を入力して下さい:

```
fb:F[n-1]+F[n-2];
define(F[n],buildq([u:fb],u));
F[0]:0$F[1]:1$F[2]:1$
```

すると次のような結果が得られる筈です;

```
(%i1) fb:F[n-1]+F[n-2];
(%o1)                          F      + F
                                n - 1    n - 2
(%i2) define(F[n],buildq([u:fb],u));
(%o2)                       F  := F      + F
                             n    n - 1    n - 2
(%i3)  F[0]:0$F[1]:1$F[2]:1$
```

```
(%i6) linel :60$

(%i7) F[100];
(%o7)             354224848179261915075
(%i8) F[1000];
(%o8) 434665576869374564356885276750406258025646605173717804\
024817290895365554179490518904038798400792551692959225930\
803226347752096896232398733224711616429964409065331879382989\
696499285160037044761377951668492288750
```

この Maxima の函数 F[n] は, 再帰的な定義となっているために大きな n を指定して計算に失敗すると, 今度は小刻みに引数を大きくして F の計算を行い, 最終的に F[n] の値を得るといった癖があります. この函数で用いた構文については「はじめての Maxima」[12][13] の「マクロの定義」を参照して下さい.

しかし, これでは面白くはありません. なんとか Yorick を使って正確な値を計算することはできないでしょうか？ ここで最も実用的な方法は整数を複数の配列を用いて表現する方法です. たとえば 1233456789 という金額は `1,234,546,789` 円と表記しますが, これを '[1,234,546,789]' と置換えると如何でしょうか？ 見事に Yorick の配列になっていると同時に数値として読むにも問題がありません. そこで安易ですが n と 10^n を Yorick で扱える整数の範囲内の数として 10^n を越えない n 桁の数に分割して整数を表現しましょう;

表 11.1: 拡大した整数与件の書式

符号	桁数	分割数	a_0	a_1	$\cdots$	$a_{分割数-2}$	$a_{分割数-1}$

この書式で数の符号は正整数なら 0, 負整数であれば 1 を置きます. そして整数 N の復元は

$$N = (-1)^{符号} \sum_{k=0}^{分割数-1} (a_k \cdot 10^{桁数\cdot k})$$

で行います.

次に問題になるのが演算ですが, ここでは Fibonacci 数列の計算に必要な和 "+" の拡大に限定します. そうすると同じ桁数の数の場合は配列の和が使え, あとは格段の桁上りの処理に注意すればよく, 桁数が異なる場合には配列の長さを合せる処理を前に入れればよいことになります.

ここで必要となる函数を幾つか定義しておきますが, これらの函数は "FibMP.i" という名前のファイルに記述しておきましょう;

```
func iceil(a)
/* DOCUMENT iceil
 *
 * ceil 函数の値を long 型の整数で返す函数.
 */
{return long( ceil (a ));};

func ifloor(a)
/* DOCUMENT ifloor --
 *
 * floor 函数の値を long 型の整数で返す函数.
 */
{ return long( floor (a ));};

func num2str(n)
/* DOCUMENT num2str -- long 型の整数を文字列に変換
 *
 * 参照: n19str
 */
{N=0; a="";
  if (numberof(n)){
     if (n!=0) N=ifloor(log(n)/log(10));
     for (i=N;i>=0;i--){
          n1=iceil(n/10^i);
          a=a+n19str(n1);};};
 return a;};

func n19str(n)
/* DOCUMENT n19str -- 1から 9の数を文字列に変換
 *
 * 参照: num2str
 */
{n1=n/10;
  if (n1==0 || n1>0){
```

```
     r=n%10;
     a=string(&char(48+r));}
 else  a="";
 return a;};
```

iceil 函数と ifloor 函数は対応する ceil 函数と floor 函数の結果が必ず long 型の整数となるように long 函数で変換を入れています．それから num2str 函数は整数を文字列に, n19str 函数が '1' から '9' の数値を対応する文字列に変換します．ここで函数の定義の中に "/* DOCUMENT .. */" といった個所がありますが, ここで記載された内容が help 函数で表示される函数の解説になります．
次に大域変数_n を設定する函数 setMPIDigit を定義します．この大域変数_n が多倍長整数を表現する配列の桁数を指示します;

```
func setMPIDigit(digit=)
/* DOCUMENT setMPIDigit -- 多倍長整数の単位桁_n の値を設定する.
 *
 * _n が未設定の場合に引数を指定しない場合,自動的に_n=9が指定される.
 */
{extern _n;
 if ( digit==nil){if(_n==nil) _n=9;}
 else  _n=digit;
 _n;};
```

この函数では変数_n を extern 文を使って明示的に大域変数として定義しています．setMPIDigit 函数は大域変数_n の値が未設定で, 桁数がキーワードで指定されていないときに自動的に変数_n に '9' を割当て, 変数_n に値が設定されているものの, キーワードが未設定のときに変数_n の値を表示する仕様です;

```
> _n
[]
> setMPIDigit;
9
> _n
9
> setMPIDigit,digit=18;
18
> setMPIDigit;
18
```

この例で示すように変数_n が未定義の場合は setMPIDigit 函数を引数なしで実行すると自動的に変数_n に 9 が割当てられ, 変数_n が定義されていると引数なしの setMPIDigit 函数は変数_n の値を表示します. また桁数をキーワード digit=で指定すると, その桁数が指定されています. なお Yorick では if 文の構文で括弧 "{ }" の省略を行う際に, その省略を行ったあとの構文を考えておく必要があります. 具体的に解説しましょう. setMPIDigit 函数の定義で次の箇所に注目しましょう;

```
if ( digit ==nil){if(_n==nil) _n=9;}
else  _n=digit;
```

ここで二番目の if 文を括る括弧 "{ }" を削除するとどうなるでしょうか？ if 文は else 文と対を成すために,

```
if ( digit ==nil){if(_n==nil) _n=9;else _n=digit;}
```

のように勝手に解釈されてしまいます. 括弧 "{ }" が外せるとは言え, 外した結果, Yorick がどう解釈するか考慮しなければならないのです. この難点は「ぶら下りの else の問題」と呼ばれるものです.

さて大域変数_n の値の上限は 64 bit 環境であれば $\log_{10}(2^{62}-1) = 18.96\ldots$ なので '18', 32 bit 環境であれば $\log_{10}(2^{31}-1) = 9.33\ldots$ なので '9' になります. 現時点では 32 bit 環境が多いために函数 setMPIDigit では '9' を既定値としています.

これらの函数を使って Yorick の整数を多倍長整数に変換する函数 L2MPI, 多倍長正整数の和を計算する函数 MPPIp, 多倍長整数の表示を行う函数 printMPI を次で定義します. これらの函数も FibMP.i ファイルに記述します;

```
func L2MPI(a)
/* DOCUMENT L2MPI -- long 型の整数を多倍長整数に変換.
 *
 * 多倍長整数を構成する配列の各成分の桁数は大域変数_n で指定.
 */
{local tmp,ans,i,k,f;
 f=0;
 if(a<0){f=1; a=-a;};
 if(a==0) k=n;
 else k=ifloor(log10(a)/_n)+4;
 ans=array(k-3,k);
 ans(2)=_n;
```

```
 for(i=k;i>3;i--){ans(i)=a/(10^_n)^(i-4); a=a%(10^_n)^(i-4);};
 ans(1)=f;
 return(ans);};

func MPPIp(a,b)
/* DOCUMENT MPPIp -- 2つの多倍長正整数同士の和を計算.
 *
 * 引数の多倍長整数の桁数は大域変数_n で指定されたものに限定
 */
{local c,tmp,d,f,k,i;
 if(a(2)==b(2) && a(2)==_n){
    k = max(a(3),b(3));
    d = abs(a(3)-b(3));
    if (!( d==0)){
       tmp = array(0,d);
       if(k>a(3)) a = _(a,tmp);
       if(k>b(3)) b = _(b,tmp);};
    c = array(long,k+3);
    f = array(long,k+1);
    c(1)=0; c(2)=a(2); c(3)=k;
    for(i=1;i<=k;i++){
        tmp  = a(i+3)+b(i+3)+f(i);
        f(i+1)= tmp/10^_n;
        c(i+3)= tmp%10^_n;};
    if(f(k+1)>0){c(3)=c(3)+1; c=_(c,f(k+1));};
    return(c);};};

func printMPI(a)
/* DOCUMENT printMPI -- 多倍長整数を通常の数として表示
 *
 */
{local k,f,i,j,m,s,t;
 t=a(2); s=num2str(t); j=ifloor(54/t);
 f=a(1); k=a(3);        m=(j-1)*t;
```

```
 write,format="%"+s+"d",linesize=0,a(k+3);
 if(f>0){m=m-t;write,format="$"+s+"s","-";};
 for(i=k+2;i>3;i--){
     m=m-t;
     write,format="%0"+s+"d",linesize=0,a(i);
     if(m<0){m=j*t;write,format="%s\n","\\";};};
 write,format="\n","";};
```

函数 L2MPI で定義した多倍長整数がどのようなものか観察してみましょう;

```
> setMPIDigit,digit=3
3
> L2MPI(1234567890)
 [0,3,4,890,567,234,1]
> setMPIDigit,digit=7
7
> L2MPI(1234567890)
 [0,7,2,4567890,123]
```

今度は函数 Fib2 を多倍長整数向けに書換えますが, 引数は long 型の整数のままで問題ないでしょう. 内部的に初期値 ‘F0’ と ‘F1’ を多倍長整数で, ‘F1=F0+F1’ を多倍長正整数の和 ‘MPPIp(F0,F1)’ で置換えれば十分です. この函数も “FibMP.i” ファイルに追加しましょう；

```
func Fib2MPI(n)
/* DOCUMENT Fib2MPI -- 多倍長整数を使ってFibonacci 数を計算
 *
 * なお, 引数のn は Yorick の通常の long 型整数を利用.
 */
{local F0,F1,tm,ip;
 F0=[0,_n,1,0];  F1=[0,_n,1,1];
 if(n==0) F1=F0;
 for(i=2;i<=n;i++){tmp=F1; F1=MPPIp(F0,F1); F0=tmp;};
 return F1;};
```

では早速試してみましょう;

```
> include,"FibMP.i";
> setMPIDigit;
9
> printMPI(Fib2MPI(10))
```

```
       55

 []
> printMPI(Fib2MPI(100))
       354224848179261915075

 []
> printMPI(Fib2MPI(1000))
 434665576869374564356885276750406258025646605173717804024\
2481729089536555417949051890403879840079255169295922593080\
8032263477520968962323987332247116164299644090653318793829\
2989696499285160037044761377951668492288875

 []
> printMPI(Fib2MPI(10000))

    336447648764317832666216120051075433103021484606800639\
90656476997468008144216666236815559551363373402558206533\
26808361593737347904838652682630408924630564318873545443\
69559827491606602099884183933864652731300088830269235673\
61313511757929743785441375213052050434770160226475831890\
65278908551543661595829872796829875106312005754287834532\
15515103870818298969791613127856265033195487140214287532\
69818796204693609787990035096230229102636813149319527563\
02278376284415403605844025721143349611800230912082870460\
88923962328835461505776583271252546093591128203925285393\
43462090424524892940390170623388899108584106518317336043\
74707379085526317643257339937128719375877468974799263058\
37065742830161637408969178426378624212835258112820516370\
29808933209990570792006436742620238978311147005407499845\
92503606335609338838319233867830561364353518921332797329\
08133732642652633989763922723407882928177953580570993691\
04917547080893184105614632233821746563732124822638309210\
32977016480547262438423748624114530938122065649140327510\
86643394517512161526545361333111314042436854805106765843\
49352383695965342807176877532834823434555736671973139274\
62736291082106792807847180353291311767789246590899386354\
59327894523777674406192240337638674004021330343297496902\
02832814593341882681768389307200363479562311710310129195\
31697946076327375892535307725523759437884345040677155557\
79056450443016640119462580972216729758615026968443146952\
03461493229110597067624326851599283470989128470674086200\
85871350162603120719031720860940812983215810772820763531\
86624611278245537208532365305775956430072517744315051539\
60090516860322034916322264088524885243315805153484962243\
48482993809050704834824493274537326245677558790891871908\
03662058009594743150052402532709746995318770724376825907\
```

```
419939632265984147498193609285223945039707165443156421 32\
815768890805878318340491743455627052022356484649519611 24\
602683139709750693826487066132645076650746115126775227 48\
621598642530711298441182622661057163515069260029861704 94\
542504749137811515413994155067125627119713325276363193 96\
069028956502882686083622410820505624307017949761711212 33\
066073310059947366875

[]
```

ここでは函数 setMPIDigit で大域変数_n に既定値の桁数 9 を設定しています．それでも n=10000 の計算でも速度的に問題はありませんし, n=100000 も計算可能です．しかし仮想端末全体が数で埋まってしまいます．この計算の妥当性ですが, ここで示した n=10, 100, 1000, 10000 に対しては Maxima の結果と容易に比較できます．つまり Maxima で Fibonacci 数を計算させて, その結果と Yorick での結果表示を複写したものを入力として両者の差を計算することで乱暴ですが確認ができます．ここでもし, 函数の使い方が判らなければ help 函数を使ってみましょう;

```
> help,Fib2MPI
 /* DOCUMENT Fib2MPI -- 多倍長整数を使ってFibonacci 数を計算
  *
  * なお, 引数のn は Yorick の通常の long 型整数を利用.
  */
 defined at:  LINE: 129 FILE: /home/yokota/Works/Yorick/Book/FibMP.i
```

日本語表示可能な仮想端末を使っていれば, このように表示される筈です.

11.2 有理数の処理

11.2.1 有理数の定義

多倍長整数を大雑把に定義してみましたが, まだ何か不足しています．実際, あれば良いものの一つに「有理数」があります．ここでは二つの long 型整数を使って有理数を定義してみましょう．

そこで有理数とはどのような数でしょうか？ まず正の有理数は分子と分母の二つの自然数の対 (n, d) で表現される数です．特に分母 d は 1 以上でなければなりません．そして二つの正の有理数を表現する自然数の対 (n_1, d_1) と (n_2, d_2) が与えられたときに, これらの正の有理数が等しくなるのは $n_1 \cdot d_2 - n_2 \cdot d_1 = 0$ を満す場合です．このことは与えられた有理数が等しいものかどうかを字面だけではなく, 関係式を使ってちゃんと判断しなければならないことを意味します．

ではどのように有理数を表現すべきでしょうか？ 一つの考え方は多倍長整数のように配列を使って表現することです．ただし, この方法では多倍長整数を有理数に使おうとすれば多倍長整数が長さが可変な配列であるために処理が大変そうです．そこで, ここでは構造体を使って定義してみましょう．そうすると変数 x に割当てられた有理数の分子は x.numerator, 分母を x.denominator と容易に取出せるだけではなく, 前節の多倍長整数も容易に組込めます．

ここで有理数の正負の表現はどうすればよいでしょうか？ 有理数を単純に整数対とすれば $(-2, 3)$ と $(2, -3)$ は同じ有理数 $-2/3$ の表現になって一意に定まりません．そこで負の有理数はその絶対値に -1 をかけたもの, すなわち $-2/3 \Rightarrow (-1, 2, 3)$ とします．すると符号は 1 か -1 のどちらかになるので真理値で置換えてもよさそうです．そこで正の数であれば整数 ‘0’, 負の数であれば整数 ‘1’ にそれぞれ対応付けておきます．

これらのことを念頭に置いてライブラリ “Ratio.i” を構築しますが, このライブラリの冒頭で有理数を Yorick の構造体を使って次のように定義します;

```
/*
 * 有理数Ratio の定義
 *
 * 有理数は構造体Ratio で定義する.
 *
 */
struct  Ratio{
 int    sgn;
 long  num, den; };
/*
 * 重要なRatio の定義
 *
 */
_Zero=array(Ratio); _Zero.den=1;
_One=array(Ratio); _One.num=1; _One.den=1;

/*
 * 必要なライブラリ
 *
 */
```

```
require,"FibMP.i";
```

Yorick の構造体は C の構造体の定義方法とほぼ同じです．ここで Ratio は符号 sgn と分子 num と分母 den で構成され，分子と分母を自然数とします．分母は基本的に 1 以上の自然数，符号 sgn は 0 の場合に正の有理数，1 の場合に負の有理数を表現することにします．具体的な定義として Ratio で 0 と 1 をそれぞれ_Zero，_One で定義しています．
この "Ratio.i" ライブラリでは前節で構築した "FibMP.i" ライブラリの函数を幾つか利用しますが，include 函数か require 函数の何れかで読込むようにしておきます．

11.2.2 有理数の真理函数

有理数を定義すると次に何が必要でしょうか？ 与えられた対象が有理数であるかどうかを判別する函数が必要ですね．この判別函数は与えられた対象が Ratio 型であれば 1，それ以外は 0 を返す函数であるべきです．このような函数は真理函数，あるいは述語と呼ばれます．この有理数の真理函数を次で定めましょう;

```
func is_Ratio(a)
/* DOCUMENT is_Ratio -- Ratio の真理函数
 *
 * Ratio であれば 1 を返すが，それ以外は全て 0を返す．
 *
 */
{if(catch(-1)) return(0);
 local ans,b;
 ans=1;
 if(typeof(a)=="struct_instance"){b=a.sgn; b=a.num; b=a.den;}
 else  ans=0;
 return ans;};

func is_RI(a)
/* DOCUMENT is_RI -- 有理数の真理函数
 *
 * Yorick の整数であるか，Ratio であれば 1,
 * それ以外は全て 0を返す函数.
```

```
 */
{return is_Ratio(a) || is_integer (a);};
```

is_Ratio 函数が Ratio 型の真理函数です. この函数には catch 函数を使った例外処理があります. つまり変数 a が構造体であれば変数 a から具体的に符号 sgn, 分子 num と分母 den が取出せるかを試し, ここでエラーが出た場合に 0 を返させるというものです. この例外処理を行う catch 函数文は, この is_Ratio 函数の例で示すように函数定義文の本体で頭に置く必要があります. ここで catch 函数の引数が-1 になっていますが, この-1 で全てのエラーに対して catch 函数を含む if 文で処理が行われることを意味します. この例は用心深い方法で, より簡便な処理は nameof 函数と structof 函数を組合せた 'nameof(structof(⟨ 対象 ⟩))' で対象の構造体としての名前を返却させて照合を行うという方法もあります;

```
func is_Ratio(a)
/* DOCUMENT is_Ratio -- Ratio の真理函数
 *
 * Ratio であれば 1 を返すが, それ以外は全て 0を返す.
 *
 */
{if(catch(-1)) return(0);
 if(nameof(structof(a))=="Ratio") return 1;
 else return 0;};
```

ただし最初の例では構造体の中身を見ているのに対し, こちらは名前だけなので構造体の実体が違った場合には問題が生じます. そのために, ここでは用心深い最初の方式を採用することにします.

有理数を long 型等の整数と Ratio で構成するので, 残りは整数の述語になります. この整数の述語は Yorick のパッケージ yutils の utiles.i て定義された is_integer 函数を用います. ただし, yutils パッケージは KNOPPIX/Math には含まれていますが, 自力でコンパイルした場合や MS-Windows 版には含まれてません. このパッケージのインストールは §7.11 を参照して下さい. ただし is_integer 函数の内容は次のように定義されているので, この函数を別途定義しても構いません;

```
func _is_integer (x)
/* DOCUMENT is_integer(x)
     returns true if array X if of integer type.

```

```
   SEE ALSO is_scalar, is_vector, is_matrix, is_array,
             is_integer, is_real, is_complex, is_numerical. */
{ return ((s=structof(x))==long || s==int || s==char || s==short);}
if (! is_integer) is_integer = _is_integer;
```

11.2.3　有理数の変換函数

ここで整数を Ratio に変換したり, 分母が 1 や分子が 0 の Ratio を整数に変換する函数があっても良いですね. これらの函数を次で定義します:

```
func I2Ratio(a)
/* DOCUMENT I2Ratio -- Yorick の整数を Ratio に変換.
 *
 * 引数が整数であれば有理数に変換し,
 * 引数が有理数の場合はそのまま返し,
 * それ以外はnil を返却する.
 */
{local c;
 if( is_integer (a)){
    c=_Zero;
    if(a>0)c.num=a; else if(a<0){c.num=-a; c.sgn=1;};
    return c;}
 else if(is_Ratio(a)) return a;};

func R2Integer(a)
/* DOCUMENT R2Integer --- Ratio から整数に変換.
 *
 * 分子が 0のものと約分によって分母が 1になったもの
 * に対してのみ変換を行う.
 *
 */
{if( is_integer (a)) return a;
 else if(is_Ratio(a)){
    if(a.num==0) return 0;
```

```
    else if(a.den==1) return (1−2∗a.sgn)∗a.num;};};

func R2Double(a)
/∗ DOCUMENT R2Double −− 浮動小数点数に変換する函数.
 ∗
 ∗/
{if(is_Ratio(a)) return (1−2∗sgn)∗a.num∗1.0/a.den;
 else if( is_integer (a) return a;};
```

I2Ratio 函数が Yorick の整数を Ratio に変換する函数です．この函数を使えば容易に有理数が生成できます．一方で R2Integer 函数は Ratio 型の対象で分子が 0 なら 0, 分母が 1 なら符号付きで分子のみを返し, それ以外は nil を返す函数です．そして R2Double 函数は有理数を浮動小数点数に変換する函数です.

11.2.4 有理数の表示函数

ここで Ratio 型の対象は Ratio(sgn=0,num=0,den=1) のような表記であり, 直感的なものではありません．そこで有理数を綺麗に表示させてみましょう．ここでは Maxima のような文字を使って分数表示させるのは如何でしょうか;

```
func printRATIO(a)
/∗ DOCUMENT printRATIO −− Ratio を文字を使って表示.
 ∗
 ∗/
{local n,k1,k2,dv,nd,b,g,br;
 br="      ";
 if(is_Ratio(a)){
    if(a.num==0) write,format=br+"%d\n",linesize=0,0;
    else{
       g=gcd(a.num,a.den); a.num=a.num/g; a.den=a.den/g;
      if(a.sgn!=0) b="− ";
      else b="";
      n=ifloor(log10(max(a.num,a.den))+1+2∗a.sgn);
      k2=num2str(n+1); nd=br+"%"+k2+"hd\n";
      if(a.den==1) write,format=nd,linesize=0,(1−2∗a.sgn)∗a.num;
```

```
       else {
          k1=num2str(n); dv=br+"%"+k1+"hs\n";
          for(i=1;i<n+2;i++) b=b+"−";
          write,format=dv,linesize=0,b;
          write,format=nd,linesize=0,a.den;};};}
else if( is_integer (a)) write,format=br+"%d\n",linesize=0,a;};
```

この printRATIO 函数は昔の数式処理システムで見られるように, 文字だけで有理数を「数式」として表示する函数です. 実際に動作を確認してみましょう;

```
> a1=array(Ratio); a1.num=8172;a1.den=827342;
> a2=array(Ratio); a2.sgn=1;a2.num=872;a2.den=8342;
> a3=I2Ratio(2);   a4=I2Ratio(−5);
> printRATIO(a1); printRATIO(a2);
        4086
     −−−−−−−
      413671
[]
         436
     − −−−−−−−
        4171
[]
> printRATIO(a3); printRATIO(a4);
      2
[]
       −5
[]
```

昔の計算機の表示のように古風ですが実用上問題ありません.

11.2.5 有理数の大小関係

今度は有理数の同値性と 大小関係を処理する函数を定義しましょう:

```
func RSimp(a)
/* DOCUMENT RSimp −− Ratio の約分を実行.
 *
 * 有理数の約分を実行するが, それ以外の対象はそのまま返却する.
 * なお, この函数では分母が 1の場合と分子が 0であっても整数化しない.
 * すなわち, 型を保つ函数である.
 */
```

```
{local g,b;
 if(is_Ratio(a)){
    b=_Zero;
    if(a.num!=0){
       g=gcd(a.num,a.den);
       b.sgn=a.sgn; b.num=a.num/g; b.den=a.den/g;};
    reutnr b;}
 else return a;};

func REq(a,b)
/* DOCUMENT REq -- 有理数の同値性を検証
 *
 * REq(a,b) ==1 <-> a と b が同値の場合.
 * REq(a,b) ==0 <-> a が b が同値でない場合.
 *
 * ここでの同値性は型を込めて考慮したもので, _Zero!=0, _One!=1である.
 */
{if(is_Ratio(a)&&is_Ratio(b))            return a==b;
 else if( is_integer (a)&&is_integer(b)) return a==b;
 else if( is_integer (a)&&is_Ratio(b))   return I2Ratio(a)==b;
 else if(is_Ratio(a)&&is_integer(a))     return a==I2Ratio(b);
 else return 0;};

func RGr(a,b)
/* DOCUMENT RSimp -- 有理数の大小関係を判定.
 *
 * RGr(a,b) ==1 <-> a>b の場合
 * RGr(a,b) ==0 <-> a<=b の場合
 *
 * ここでa,b は整数, あるいはRatio の何れかである.
 */
{local a1,b1,gn,gd;
 a1=a; b1=b;
 if( is_integer (a) && is_integer(b)) return a>b;
```

```
 else if( is_integer (a)) a1=I2Ratio(a);
 else if( is_integer (b)) b1=I2Ratio(b);
 if(is_Ratio(a1)&& is_Ratio(b1)){
    if(a1.sgn+b1.sgn==1){if(a1.sgn==0) return 1;}
    else{
        gn=gcd(a1.num,b1.num); gd=gcd(a1.den,b1.den);
        return (1-2*a1.sgn)*(a1.num/gn)*(b1.den/gd)>
                  (1-2*b1.sgn)*(a1.den/gd)*(b1.num/gn);};}
 else return 0;};

func RLs(a,b)
/* DOCUMENT RSimp -- 有理数の大小関係を判定.
 *
 * RLs(a,b) ==1 <-> a<b の場合
 * RLs(a,b) ==0 <-> a>=b の場合
 *
 * ここでa,b は整数, あるいはRatio の何れかである.
 */
{local a1,b1,gn,gd;
 a1=a; b1=b;
 if( is_integer (a) && is_integer(b)) return(a<b);
 else if( is_integer (a))a1=I2Ratio(a);
 else if( is_integer (b))b1=I2Ratio(b);
 if(is_Ratio(a1)&& is_Ratio(b1)){
    if(a1.sgn+b1.sgn==1){if(a1.sgn==0) return 1;}
    else{
        gn=gcd(a1.num,b1.num); gd=gcd(a1.den,b1.den);
        return (1-2*a1.sgn)*(a1.num/gn)*(b1.den/gd)<
                  (1-2*b1.sgn)*(a1.den/gd)*(b1.num/gn);};}
 return 0;};
```

RSimp 函数は有理数の約分を行う函数です. 有理数 a/b と c/d の同値性を調べるために $ad-bc$ を計算するよりも, 約分を行って等しいかどうか検証する方が Yorick の long 型の整数を利用する以上は安全です.

REq 函数で 2 つの有理数が等しいかどうかを判別しますが, 配列では論理演算子 “==” によって配列の各成分毎の同値性が示されるのに対し, ここでは 0 次元の配列や構造体を扱うために ‘0’ か ‘1’ が返却されます.
函数 RGr と函数 RLs は有理数の大小関係の判別を行う真理函数です. これらの函数では有理数 a/b と有理数 c/d の大小関係を調べるために $ad > bc$ や $ad < bc$ を調べますが, ここで整数の上限があるために安易に分子と分母の積を計算せずに, 有理数の共通因子を取出して処理させています.

11.2.6　有理数の四則演算を行う函数

有理数の四則演算を行う函数を定義しましょう. そこで最初に和と差を計算する函数を次で定義します;

```
func RAdd(a,b)
/* DOCUMENT RAdd -- 有理数の和を計算
 *
 */
{local  c,a1,b1,g,g1,g2;
 a1=RSimp(a); b1=RSimp(b);
 if( is_integer (a)&& is_integer(b)) return a+b;
 else if( is_integer (a)) a1=I2Ratio(a);
 else if( is_integer (b)) b1=I2Ratio(b);
 if(is_Ratio(a1)&& is_Ratio(b1)){
    c=_Zero;
    g1=gcd(a1.num,b1.num); g2=gcd(a1.den,b1.den);
    if(g1==0)g1=1;
    c.num=(1-2*a1.sgn)*(a1.num/g1)*(b1.den/g2)+
           (a1.den/g2)*(1-2*b1.sgn)*(b1.num/g1);
    c.den=(a1.den/g2)*(b1.den/g2);
    c.num=c.num*g1; c.den=c.den*g2;
    if(c.num<0){c.sgn=1; c.num=-c.num;};};
 return RSimp(c);};

func RNeg(a)
/* DOCUMENT RNeg -- 有理数の和の逆元を計算
```

```
 *
 */
{local c;
 if(is_Ratio(a)){c=a; c.sgn=(a.sgn+1)%2; return c}
 else if( is_integer (a)) return -a;};

func RSub(a,b)
/* DOCUMENT RSub -- 有理数の減算
 *
 * 内部的にRNeg と RAdd を利用
 *
 */
{return RAdd(a,RNeg(b));};
```

RAdd 函数は 2 つの有理数の和を計算する函数です．RNeg 函数は与えられた有理数 a に対して有理数 $-a$ を返す函数です．これらの RAdd 函数と RNeg 函数を用いることで 2 つの有理数の差を計算する函数 RSub が構成されます．
今度は有理数の積と商を計算する函数を定義しましょう;

```
func RTimes(a,b)
/* DOCUMENT RTimes -- 有理数の積
 *
 */
{local c,a1,b1,g1,g2;
 a1=RSimp(a); b1=RSimp(b);
 if( is_integer (a) && is_integer(b)) return a*b;
 else if( is_integer (a)) a1=I2Ratio(a);
 else if( is_integer (b)) b1=I2Ratio(b);
 if(a1==_Zero || b1==_Zero) c=_Zero;
 else if(is_Ratio(a1)&& is_Ratio(b1)){
    c=_Zero;
    g1=gcd(a1.num,b1.den);  g2=gcd(a1.den,b1.num);
    c.sgn=(a1.sgn+b1.sgn)%2; c.num=(a1.num/g1)*(b1.num/g2);
    c.den=(a1.den/g2)*(b1.den/g1);
    return c;};};
```

```
func RRev(a)
/* DOCUMENT RRev -- 有理数の逆数を計算
 *
 * 0に対してはnil を返す.
 *
 */
{local c,a1;
 a1=a;
 if( is_integer (a))  return I2Ratio(a);
 if(is_Ratio(a1)){
     c=_One;
     if(c.num==0) return nil;
     else{c=a1; c.num=a1.den; c.den=a1.num; return c;};};};

func RDiv(a,b)
/* DOCUMENT RDiv -- 有理数の商を計算
 *
 * 内部ではRTimes と RRev を利用.
 * 第 2引数が 0に等しい場合はnil を返す.
 *
 */
{if(is_RI(a)&& is_RI(b)){
     if(RSimp(b)==_Zero || b==0) return nil;
     else return RTimes(a,RRev(b));};};
```

函数 RTimes で 2 つの有理数同士の積を計算します．函数 RRev は 0 と異なる有理数 a に対して有理数 a^{-1} を返す函数です．そして 2 つの有理数の商を計算する函数 RDiv では函数 RTimes と函数 RRev を利用しています．

これで有理数で遊ぶための函数は揃いました．それではいろいろな数をこれから計算してみることにしましょう．

11.2.7 とにかく計算!

黄金比の計算

ここでは手始めに黄金比を計算してみましょう．ここで黄金比は $1 : (1+\sqrt{5})/2$ で, 次の連分数表示から計算できます;

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\ddots}}}}$$

連分数の周期性に注目し, 函数 Golden を定義します;

```
func Golden(n){
 local ans,i;
 ans=1;
 for(i=1;i<n;i++) ans=RAdd(1,RRev(ans));
 printRATIO(ans); print,R2Double(ans);
 return ans;};
```

では実際に計算をさせてみましょう;

```
> Golden(5)
      8
     --
      5
[]
1.6
Ratio(sgn=0,num=8,den=5)
> Golden(10)
      89
     ---
      55
[]
1.61818
Ratio(sgn=0,num=89,den=55)
> Golden(20)
      10946
     ------
       6765
[]
```

```
1.61803
Ratio(sgn=0,num=10946,den=6765)
```

ここでの計算結果から n=20 で十分な近似値が得られているようです．実際に Yorick で '(1.+sqrt(5))/2.' と差を計算してみましょう．そのために函数 Golden で計算した結果を R2Double 函数で浮動小数点数に変換して確認しています;

```
> R2Double(Golden(20))−(1.+sqrt(5))/2.
     10946
    −−−−−−
      6765
[]
1.61803
9.77191e−09
```

このように小数点 7 桁程度は一致しているようですね．では今回定義した有理数では非常に無謀なことですが無限級数の計算で少々遊んでみましょう．

$\log 2$ を計算する無限級数:

まず $\log 2$ と等しくなる無限級数を示しておきます;

$$\sum_{i=1}^{\infty} \frac{(-1)^{i-1}}{i} = \log 2$$

この公式を Yorick の函数で書き換えたものを次に示しておきます;

```
func Log2(n){
 local ans,i;
 ans=0;
 for(i=1;i<n;i++) ans=RAdd(ans,RDiv((−1)^(i−1),i));
 printRATIO(ans); print,R2Double(ans); return ans;};
```

この函数 Log2 は上記の公式で $l = n$ までの和を返す函数で計算結果を有理数と浮動小数点数による近似値の表示も行います．ここで $\log 2 = 0.69314\ldots$ ですが, この Log2 の結果はどうでしょう？

```
> Log2(10)
     1879
    −−−−−
     2520
```

```
[]
0.745635
Ratio(sgn=0,num=1879,den=2520)
> Log2(20)
   33464927
 −−−−−−−−−−
   46558512
[]
0.718771
Ratio(sgn=0,num=33464927,den=46558512)
```

と, あまり正確ではありません. しかも, 32 bit 環境であれば n=24 以上で Yorick の整数の範囲を越えてしまう欠点がありますが, 残念ながら n=23 でも近似値として十分な値ではありません.

Leibniz の公式による π の計算

$\pi/4$ が計算できる「Leibniz の公式」があります:

$$\sum_{i=1}^{\infty} \frac{(-1)^{i-1}}{2i-1} = \frac{\pi}{4}$$

今度はこの Leibniz の公式を有理数を用いて計算させてみましょう;

```
func LeibnizFormula(n)
{local ans,i;
 ans=0;
 for(i=1;i<n;i++) ans=RAdd(ans,RDiv((-1)^(i-1),2*i-1));
 ans=RSimp(ans); printRATIO(ans); print,R2Double(ans);
 return ans;};
```

この函数は先程の Log2 函数と殆ど同じ構成になっています. さて $\pi/4 = 0.78539\ldots$ ですが, LeibnizFormula による計算結果は;

```
> LeibnizFormula(5)
    76
 −−−−
   105
[]
0.72381
Ratio(sgn=0,num=76,den=105)
```

```
> LeibnizFormula(10)
        622637
       -------
        765765
[]
0.813091
Ratio(sgn=0,num=622637,den=765765)
```

と, これもあまり芳しいものではありません. このような超越数が得られるような無限級数に対しては今回構築した程度の有理数では明らかに力量が不足しています.

では十分な力量があった場合はどうなるでしょう. そこで, Maxima を使って検証してみましょう. 起動した Maxima に次の式を入力して下さい;

```
Log2(n):=block([ans:0,i],
          for i:1 thru n−1 do ans:ans+(−1)^(i−1)/i,ans);
LeibnizFormula(n):=block([ans:0,i],
          for i:1 thru n−1 do ans:ans+(−1)^(i−1)/(2*i−1),ans);
```

とりあえず Yorick の結果と比較しておきましょう;

```
(%i24) [Log2(10),Log2(20)];
                              1879   33464927
(%o24)                       [----, --------]
                              2520   46558512
(%i25) [LeibnizFormula(5),LeibnizFormula(10)];
                               76   622637
(%o25)                        [---, ------]
                              105   765765
```

このように Maxima でも同じ結果となりますね. では, n=100 でどの程度の近似になっているでしょうか？

```
(%i28) float([Log2(100)−log(2),LeibnizFormula(100)−%pi/4]);
(%o28)          [.005024998750249976, .002525188120330535]
```

と思った程, 精度は良くありませんが有理数としては Log2(100) と LeibnizFormula(100) の結果はそれぞれ;

```
(%i29) Log2(100);
                 97353652632905823380247894258032042316377
(%o29)           -----------------------------------------
                 139440750459424954329067617870624607113600
(%i30) LeibnizFormula(100);
(%o30) 104334757757026909214174502414645621748371218299386743853514281377759999\
708155776347203/1324173951134289669595892614615426379681196540456804468549179595\
```

```
77250447621155297542662 5
```

のように恐しいことになっているのです．そして n=10000 でやっと小数点下 3 桁が一致するのです．その意味で, この級数の計算は非常にキツい計算です．

BBP 公式による π の計算

Leibniz の公式では散々でしたが π の計算はもっと良い級数が使えます．たとえば Bailey, Borwein と Plouffe による次の公式 (BBP 公式) [14] を用いてみましょう;

$$\sum_{i=1}^{\infty}\frac{1}{16^i}\left(\frac{4}{8i+1}-\frac{2}{8i+4}-\frac{1}{8i+5}-\frac{1}{8i+6}\right)$$

この BBP 公式を用いて計算する函数 BBPsPI を次に示しておきましょう:

```
func BBPsPI(n)
{local ans,a1,a2,i;
 ans=0;
 for(i=0;i<=n;i++){
     a1=RAdd(RDiv(4,8*i+1),RDiv(-2,8*i+4));
     a2=RAdd(RDiv(-1,8*i+5),RDiv(-1,8*i+6));
     ans=RAdd(ans,RDiv(RAdd(a1,a2),16^i));};
 ans=RSimp(ans);
 printRATIO(ans); print,R2Double(ans);
 return ans;};
```

この函数の基本形は Log2 や LeibnitzFormula と全く同じものです．この計算結果を示しますが非常に高速であることが判ります;

```
> pi-R2Double(BBPsPI(1))
     102913
    -------
      32760
[]
3.14142
0.000170187
> pi-R2Double(BBPsPI(2))

     615863723
    ----------
```

```
        196035840
 []
3.14159
5.26324e−06
> pi−R2Double(BBPsPI(3))
        357201535487
       −−−−−−−−−−−−−−
        113700787200
 []
3.14159
1.96022e−07
> pi−R2Double(BBPsPI(4))
        16071212445820879
       −−−−−−−−−−−−−−−−−−−
         5115625817702400
 []
3.14159
8.12946e−09
```

ここでは 64 bit 環境で計算させているために n=4 までの計算ができます．32 bit 環境では n=2 が限界ですが, それでも Yorick の通常の表示では差は見えず, わざわざ差を計算させて誤差が非常に小さなものとなっていることが判る程です.
では Maxima ならどうでしょうか？ 次の式を入力して試してみましょう:

```
BBPsPI(n):=block([i,ans:0],for i:0 thru n do
    ans:ans+(4/(8*i+1)−2/(8*i+4)−1/(8*i+5)−1/(8*i+6))/16^i,ans);
```

ここで n=100 として計算させてみましょう．計算した有理数を浮動小数点数に変換しても面白くないので大域変数 fpprec に 100 を割当て, bfloat 函数を使って 100 桁の多倍長浮動小数点数に変換してみます;

```
(%i8) fpprec:100$
(%i9) BBPsPI(100);
(%o9) 817767058706814527596377101198961590990563006507316848090203872901278894\
3475215964160027329017114926448573863946585952299480108759888420967632238816 40\
0897010401837175974789945488577373023715468640688591470221689875547301256225 77\
8043277671246051531245056550967091820468750978957843214284803389274465737440 15\
3080229563676556732058417728390250671873801797908072373035825324293306 9/260303\
3393818193965835370520094584481252702771456570007897033071062121681839861734 23\
0188021090311626569877955012660007341874288310243718319029892967189797885962 22\
1796703066383876427327416822142776751204651243263467866327213591039978715585 93\
7416280913207472509790935300488933481514083801303249841316558392925357619422 48\
74228203052332856355677249981064004796964724715720540160000
(%i10) bfloat(%);
(%o10) 3.1415926535897932384626433832795028841971693993751058209749445923078 16\
```

```
4062862089986280348253421117068b0
```

ここで π の検証ですが `fpprec:1000;` によって桁数を 1000 桁にしてから `bfloat(BBPsPI(100)-%pi);` と入力してみましょう;

```
(%i17) fpprec:1000$
(%i18) bfloat(BBPsPI(100)−%pi);
(%o18) − 5.880802408811441129071491693911468140114072356896134792858731016652\
60988244259261434353935612101975490660489477336067464229803696321248641487396\
20127400584116555193502242038386349935440531014476338803703946330802931089749\
53088883387180942018100789309471726992867418811574841140419565547203013848528\
14117627847997331873104110835119615815432842378493410336723788058967011279100\
05186412261759727012778861420869987414293990951994144840314587628375124951618\
80129917938882701976288488711125948569393420277929883724479566524510949255289\
87886027812797125960846970836906574022847017933855436420618051365335245018907\
15150581171389406756757565549648857958795074817620587389653468748405695146050\
47605514646444030415282430203365569654569625446246520010701620537110658646917\
68251168358412161803461950521218663730392412236033763420677597685127005680781\
29988308848206602946893046972751901597716056767660872285460579388000723991418\
89409272114145610725850733481581598405498535026226035313950857338292707030290\
66001929b−127
```

と, 小数点下 125 桁辺は大丈夫そうですが, そもそも Maxima で 100 桁以上が正しいかどうかは別の問題です. したがって「円周率 100,000,000 桁表」[10] のような数表で実際に確認する方が正確でしょう.

11.3 多項式の表現

11.3.1 多項式の定義

有理数を定義してみましたが数列では思った程は使えませんでした. 無限級数の項が Yorick の整数で表現可能でも, それらの和となると分子や分母が大きくなってしまい, 32 bit 環境では Log2 や LeibnizFormula で n=20 程度, BBP なら n=2 程度でしか使えませんでした. とはいえ有理数係数の多項式のちょっとした処理なら実用に耐えるでしょう. そこで今度は多項式をファイル “Poly.i” に定義してみましょう.
ここで多項式の性質上, 変数は文字列, 係数列は数値のために Yorick の 1 つの配列だけで表現はできません. そこで LISP 風のリストを用いる手法が挙げられますが, ここでは利便性を考慮して構造体を用いた表現としましょう l;

```
/*
 * 多項式の定義
 *
```

```
 * coeff  – 係数で構成された配列. 1変数の場合は [c_0,c_1,...c_n]
 * vars   – 変数で構成された配列. 1変数の場合は ["x"]
 */
struct Polynomial{pointer coeff, vars;};
```

Yorick の構造体では項目に 1 次元以上の配列を直接代入することはできません．つまり 'poly1.coeff=[1,2,3,4]' のような処理はできません．そこで配列を代入する場合には pointer 型を介在させます．すなわち 'polyn1.coeff=&[1,2,3,4]' のようにします．ただし pointer 型を介在させなければならないのは 1 次元以上の配列で, 0 次元の配列の場合は poly1.coeff=1 のような通常の代入ができます．
それでは具体的に値を割当ててみましょう．多項式 $x^2 - 2$ を表現する場合, 係数配列を [-2,0,1], 変数配列を ["x"] とします．これらは全て 1 次元以上の配列であるために次のように入力することになります;

```
> p1=array(Polynomial)
> p1.vars=&["x"];
> p1.coeff=&[-2,0,1];
> p1
Polynomial(coeff=0x718d68,vars=0x76ce88)
> *p1.coeff
[-2,0,1]
> (*p1.coeff)(3)
1
```

このように配列を pointer 型として変数に代入する場合, C のように対象の先頭に記号 "&" を置きます．そして対象の名前, ここでは `p1` と入力すると, 各項目が置かれている番地が返されて実際の値は `*p1.coeff` のように変数名の先頭に記号 "*" を置くことで返却されます．そして配列の元を取出す場合は `(*p1.coeff)(3)` のように括弧 "( )" を利用しなければなりません．

11.3.2　多変数多項式の表現

1 変数多項式の場合は多項式の係数を単純に次数の低い順に並べた 1 次元配列を構成するだけで済みます．すなわち次の対応関係があります;

1 変数多項式と Polynomial の関係

$$\text{poly} \stackrel{def}{=} c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1} + c_n x^n \Rightarrow$$
$$\texttt{Polynomial}\{\texttt{poly.coeff} = [c_0, c_1, c2, \ldots, c_n], \texttt{poly.vars} = [\text{"}x\text{"}]\}$$

多変数 $x, y, z, \cdots$ の場合はどうでしょうか？ この場合, ‘vars=[x,y,z,..]’ の左端からの順序に意味を持たせます. つまり多項式の各項を $cx_1^{i_1}x_2^{i_2}\cdots x_n^{i_n}$ とするときに整数配列 $[i_1+1,\ldots,i_n]$ と項 $c*x_1^{n_1}\cdots x_n^{i_n}$ の係数 c を対応させます. このことによって多次元配列と多変数多項式の係数の集合との間に対応が付けられます;

多変数多項式と Polynomial の関係

$$\{poly \overset{def}{=} \sum_{0\le i_1\le m_1,\ldots,0\le i_n\le m_n} c(i_1+1,\ldots,i_n+1)\ x_1^{i_1}\cdots x_n^{i_n} \Rightarrow$$

$\texttt{Polynomial}\{\texttt{poly.coeff} = c,\ \texttt{poly.vars} = [”x_1”,\ldots,”x_n”]\}$

たとえば, $x^2y^2+x^2z^2+y^2z^2-17xyz$ で考えてみましょう. この多項式の係数配列を a とすると, $-17xyz$ より a(2,2,2)=-17, x^2y^2 より ‘(3,3,1)=1’, x^2z^2 より ‘a(3,1,3)=1’, そして y^2z^2 より ‘a(1,3,3)=1’ で, それ以外は全て 0 になります. この配列は $3\times3\times3$ の大きさの配列です. したがって, ここでは次で表現されることになります;

```
> poly1=array(Polynomial)
> poly1.coeff=&array(0,3,3,3)
> (*poly1.coeff)(2,2,2)=−17
> (*poly1.coeff)(3,3,1)=1
> (*poly1.coeff)(3,1,3)=1
> (*poly1.coeff)(1,3,3)=1
> poly1.vars=&[”x”,”y”,”z”]
> *poly1.coeff
      [[[0,0,0],[0,0,0],[0,0,1]],[[0,0,0],[0,−17,0],[0,0,0]],
  [[0,0,1],[0,0,0],[1,0,0]]]
```

11.3.3 多項式の真理函数

ここでは簡単に nameof 函数と structof 函数を組合せて, 対象の構造体の名前が文字列”Polynomial”かどうかで判断するようにしましょう:

```
func is_Polynomial(a)
/* DOCUMENT is_Polynomial −− 有理数係数多項式の真理函数
 *
 */
{if(catch(−1))return 0;
 return nameof(structof(a))==”Polynomial”;};
```

11.3.4　多項式の表示函数

ここで多項式の表示を行う函数を定義しましょう．何故なら得られた多項式の係数配列が ‘[[[0,0,0],[0,0,0],[0,0,1]],[[0,0,0], [0,-17,0],[0,0,0]], [[0,0,1],[0,0,0],[1,0,0]]]’, 変数列が ‘[”x”,”y”,”z”]’ といった表示では何が何だか判り難く, ここは矢張り $x^2y^2 + x^2z^2 + y^2z^2 - 17xyz$ のような通常の数式表示の方が判り易いからです．ただし, ここでは有理数の場合のように無理して数式風に表示させるのではなく, 一行で収まるような簡潔な表示とします．このときに問題となるのが多項式の項の表示順序です．ここでは辞書式順序と呼ばれる順序を入れています;

辞書式順序”$>_{\mathrm{lex}}$”

$$x_1^{i_1}\cdots x_m^{i_m} >_{\mathrm{lex}} x_1^{j_1}\cdots x_n^{i_n} \quad \Leftrightarrow \quad i_1 = j_1 \quad \cdots \quad i_k = j_k \text{ かつ } \quad i_{k+1} > j_{k+1}$$

今回, $vars = [x_1, \ldots, x_N], N = \max(m, n)$ としています．この順序で比較する場合, Yorick では項の係数を 1 で置換した項に対して演算子 “>” や演算子 “<” のような比較の演算子で判断が行えます．次に定義する P2String 函数で, 項を辞書式順序で並び換えた多項式を文字列に変換し, printPOLYNOMIAL 函数で文字列を表示しています:

```
func printPOLYNOMIAL(a)
/* DOCUMENT printPOLYNOMIAL -- Polynomial を多項式として表示.
 *
 * 内部でP2String 函数を利用する.
 *
 */
{write,format="%s\n",P2String(a);};

func P2String(a)
/* DOCUMENT P2String -- Polynomial を文字列の多項式に変換
 *
 */
{local N,Term,STerm,Coef,coeff,i,j,m,n,op,ot,q;
 if(is_Polynomial(a)){
    coeff=*a.coeff;    vars=*a.vars; n=numberof(vars);
    i=is_Ratio(coeff); op=i*(coeff!=_Zero)+(1-i)*(coeff!=0);
    q=where(op);       if(q==nil) return 0;
    else{
       ot=where2(op);    N=numberof(q);
```

```
        Term=array("",N); Coef=array("",N); STerm=Term;
        for(i=1;i<=N;i++){
            m=coeff(q(i));
            if(m!=_One && m!=1){
                if(is_Ratio(m)){
                    Coef(i)=num2str(m.num);
                    if(m.den>1)  Coef(i)=Coef(i)+"/"+num2str(m.den);
                    if(m.sgn==1) Coef(i)="-"+Coef(i);}
                else{
                    if(m==1)      Coef(i)="";
                    else if(m==-1)Coef(i)="-";
                    else if(m>0) Coef(i)=num2str(m);
                    else          Coef(i)="-"+num2str(-m);};}
            else Term(i)="";
            for(j=1;j<=n;j++){
                pt=ot(,i)(j);
                if(Term(i)=="") op="";else op="*";
                Term(i)=Term(i)+op+vars(j)+"^"+num2str(pt-1);
                if(STerm(i)=="") op="";else op="*";
                if(pt>1) STerm(i)=STerm(i)+op+vars(j);
                if(pt>2) STerm(i)=STerm(i)+"^"+num2str(pt-1);};};
        k=sort(Term); STerm=STerm(k); Coef=Coef(k); q="";
        for(i=1;i<=N;i++){
            if(strmatch(Coef(i),"-")||q=="") op="";
            else op="+";
            if(STerm(i)=="") ot=""; else ot="*";
            if(Coef(i)==""){
                if(STerm(i)=="") q=q+"1";
                else q=q+op+STerm(i); }
            else q=q+op+Coef(i)+ot+STerm(i);};
        return q;};};};
```

では, この printPOLYNOMIAL 函数を試してみましょう. 次に, $p1 = 1 + x^3 + x^6$, $poly1 = poly2 = -17xyz + x^2y^2 + x^2z^2 + y^2z^2$ を定義し, 表示させてみましょう:

```
> p1=array(Polynomial);
```

```
> p1.vars=&["x"]
> p1.coeff=&[1,0,0,4,0,6]
> poly1=array(Polynomial)
> poly1.coeff=&array(0,3,3,3)
> (*poly1.coeff)(2,2,2)=-17
> (*poly1.coeff)(3,3,1)=1
> (*poly1.coeff)(3,1,3)=1
> (*poly1.coeff)(1,3,3)=1
> poly1.vars=&["x","y","z"]
> poly2=array(Polynomial)
> poly2.coeff=&array(_Zero,3,3,3)
> (*poly2.coeff)(2,2,2)=I2Ratio(-17);
> (*poly2.coeff)(3,3,1)=I2Ratio(1);
> (*poly2.coeff)(3,1,3)=I2Ratio(1);
> (*poly2.coeff)(1,3,3)=I2Ratio(1);
> poly2.vars=&["x","y","z"]
> printPOLYNOMIAL(p1)
1+4*x^3+6*x^5
[]
> printPOLYNOMIAL(poly1)
y^2*z^2-17*x*y*z+x^2*z^2+x^2*y^2
[]
> printPOLYNOMIAL(poly2)
y^2*z^2-17*x*y*z+x^2*z^2+x^2*y^2
[]
```

上で示すように, きちんと多項式が表示されていますね.

11.3.5　代入処理

ここで多項式に Yorick の数値を代入する函数を構築しておきましょう. 考え方としては, 多項式とその変数ベクトルに対応する数値ベクトルの二つの対象を引数とする函数で, 多項式を文字列に変換したのちに, 変数を対応する数値で置換する方式です:

```
func SubstVal(ps,x,v)
/* DOCUMENT SubsetVal -- 文字列ps 中の文字列 x を文字列 v で置換.
 */
{local nps;
 nps=ps;
 while(strmatch(nps,x)) nps=streplace(nps,strfind(x,nps),v);
 return nps;};

```

```
func Subst(p,vlist)
/* DOCUMENT Subst -- 多項式p の変数を vlist で指定した値に変換し,
 *                         文字列で変換する函数.
 *
 * ここでvlist の書式は:
 *
 *  vlist =[["x_1","v_1  "],...,["  x_n","v_n"]];
 *
 * vlist の左側から順に式に適用されることに注意.
 */
{local i,m,n,p1;
 if(is_Polynomial(p)){
     n=numberof(*p.vars); i=0;
     for(i=0;(vlist(1,)==(*p.vars)(i+1))(sum)==0&&i<n-1;i++);
     if(i<n){
         p1=P2String(p);m=numberof(vlist)/2;
         for(j=0;j<m;j++){
              p1=SubstVal(p1,vlist(1,j+1),vlist(2,j+1));};
         return p1;};};};
```

SubstVal 函数は文字列に変換した多項式に対し, 変数をそれに対応する値で置換する函数になります. この函数では strmatch 函数を使って文字列に対応する文字が含まれるかどうかを調べ, 対応する文字が存在する場合に streplace 函数を用いて変換を行います. ここで変換すべき文字の位置は strfind 函数で検出します. そして, 変数と置換すべき値は, 変数を表現する文字列と値を表現する文字列の対で表現し, 配列 vlist をこれらの対で構成された配列とします.
では, 実際の実行例を次に示しておきましょう:

```
> Subst(poly1,[["x","10"],["y","20"],["z","30"]])
"20^2*30^2-17*10*20*30+10^2*30^2+10^2*20^2"
> Subst(poly1,[["x","10"],["z","30"]])
"y^2*30^2-17*10*y*30+10^2*30^2+10^2*y^2"
```

11.3.6　多項式のグラフ

多項式の定義, 代入と文字列による通常の数式の表示を行いましたが, これを使って何ができるでしょう？ 文字列があれば, それを外部のアプリケーションに引渡して遊ぶことができます．そこで, 手始めに gnuplot を使ってグラフ表示をさせてみましょう．ここで gnuplot は KNOPPIX/Math には最初からあります．また, MS-Windows 版もあるので, 入れていなければこの際に是非, 入れておくと良いでしょう．
まず最初に, popen 函数を使って gnulot 用のパイプを開きます:

```
> fp=popen("gnuplot",1)
```

これで gnuplot 側へのパイプが開かれましたが, MS-Windows 版の Yorick では popen 函数によって gnuplot が立上がるだけで, Yorick 側からの処理は行えません．そのために後述の中間ファイルを経由する方法を使います[3]．UNIX 環境であれば, popen 函数の第 2 引数を 1 としてるので, Yorick 側で通常の処理が行えます．そこで, 多項式を次で定義しておきましょう:

```
> sf=array(Polynomial)
> sf.vars=&["x","y"]
> sf.coeff =&[[0,0],[0,1]]
> P2String(sf)
"x*y"
```

さて, 今度は gnuplot 側に描画領域の設定を行います:

```
> write,fp,format="set xrange [-10:10];\n",""
> write,fp,format="set yrange [-10:10];\n",""
> write,fp,format="set zrange [*:*];\n",""
> fflush(fp)
write-only text stream at:
  LINE: 1  FILE: gnuplot
```

popen 函数の第 2 引数が 1 のときに write 函数等のストリーム出力を行う函数によってアプリケーションに命令を送り込みます．ここでは write 函数を使ってストリームに命令の書込を行いますが, ストリームに命令を出力してもバッファに蓄えられるだけなので fflush 函数を使ってバッファの内容を強制的にアプリケーションに引渡します．同様に write 函数を使って命令を gnuplot に送り込めばグラフの描画が行えます; ストリームを閉じるためには close 函数 を用います．ここで `close,fp` と入力すればパイプを閉じて gnuplot も終了させます．しかし, この処理も MS-Window 上では上手

[3] 数式処理の Maxima でも, UNIX 環境ではパイプを経由して gnuplot を操作できますが, MS-Windows 環境では中間ファイルを介して描画させています．

```
> write,fp,format="%s\n","set pm3d at bs"
> write,fp,format="splot %s;\n",P2String(sf)
> fflush(fp)
write-only text stream at:
  LINE: 1  FILE: gnuplot -
> gnuplot> gnuplot>
```

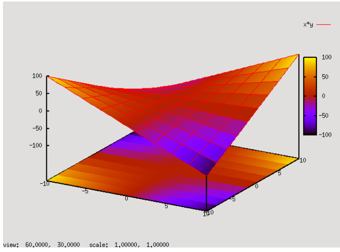


図 11.1: gnuplot による描画

く動作しません．そこでパイプではなく中間ファイルを用いた方法で遊んでみましょう．この方法であれば計算機環境に依存することはありませんし，パイプが使えないアプリケーションも対処できます．

ここでは「**surfer**」というアプリケーションで代数曲面を描かせてみましょう (入手先は [21] を参照して下さい)．ここで代数曲面とは多項式 $f(x_1, x_2, \ldots, x_n)$ に対して $f(x_1, \ldots, x_n) = 0$ となる点 $(x_1, \ldots, x_n)$ の集合 (=零点集合と呼びます) のことです．たとえば半径 1 の円は多項式 $x^2 + y^2 - 1$ の零点集合です．この surfer には MS-Windows 版と Linux 版があり，KNOPPIX/Math にも収録されています．この surfer で多項式の代数曲面を描かせたければ `surface=多項式` という内容のファイルを surfer に引渡すだけでよいのです．

Yorick 側から外部アプリケーションの起動には前述の popen 函数と system 函数があります． ここで system 函数で起動させるとアプリケーションを終了させるまで Yorick 側では何も処理ができなくなるので，ここでも popen 函数を用いましょう．そして,surfer に引渡す中間ファイルの生成は create 函数で実行し，多項式を 1 つ書込んで popen 函数を使って surfer を起動させるという仕組です．ここでは最初に多項式を定義しておきましょう;

```
> AlgeS=array(Polynomial)
> AlgeS.vars=&["x","y","z"]
> AlgeS.coeff=&transpose  ([[[1,2,3,4,5],[5,4,3,2,1],[9,5,1,-3,-7]]])
```

この多項式は非常に適当な多項式です．次にストリーム処理を行います:

```
> f=create("surfer.tmp")
> write,f,format="surface=%s;\n",P2String(AlgeS)
> close,f
> fp=popen("surfer surfer.tmp",1)
```

この中間ファイルは使い切りなので close 函数で閉じています．こうして描画された

曲面を図 11.2 に示しておきます:

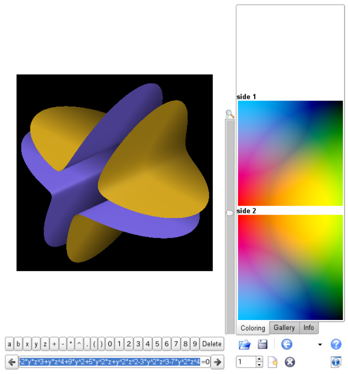


図 11.2: surfer による代数曲面の表示

surfer のウィンドウの右側に上下二つのパレットがあります．このパレットの上側が表面の色, 下側が裏面の色を定めます．画像の下に多項式が現われていますが, この欄に多項式を入れると, その多項式の零点集合を描きます．そして, 画像の右側のバーで拡大・縮小, 画像をマウスで直接把持するとグルグルと回すことができるので楽しんでみてください[4].

さて, 多項式の描画を行ってみましたが, この多項式の曲面は 4 枚に分かれていますね．ここで代数曲面では多項式が $f(x) \cdot g(x)$ に因子分解できれば, 曲面は $f(x)$ と $g(x)$ の和になるという性質があります．実際, $f(x) = 0$, あるいは $g(x) = 0$ なら $f(x) \cdot g(x) = 0$ になります．では, この多項式は因子分解できるものなのでしょうか．そこで, 数式処理の Maxima を使って調べてみましょう:

```
> fp=popen("maxima",1)
> Maxima 5.17.1 http://maxima.sourceforge.net
Using Lisp SBCL 1.0.23
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1) write,fp,format="surface:%s;\n",P2String(AlgeS)
> write,fp,format="%s\n","factor(surface);"
> fflush(fp);
write-only text stream at:
  LINE: 1  FILE: maxima
>
```

[4] surfer の描画では, surf というアプリケーションで生成した絵を表示させているだけです．それで苦もなく回転等が行える程, 現在の計算機は高性能なのです!

```
               2  4       4       4       2  3          3       3    2  2          2       2
(%o1) - 7 y   z   + y z   + 5 z   - 3 y   z   + 2 y z   + 4 z   + y   z   + 3 y z   + 3 z
                                                           2                    2
                                                    + 5 y   z + 4 y z + 2 z + 9 y   + 5 y + 1
(%i2)
                2  4       4       4       2  3          3       3    2  2          2
(%o2) - (7 y   z   - y z   - 5 z   + 3 y   z   - 2 y z   - 4 z   - y   z   - 3 y z
                                                2       2                     2
                                        - 3 z   - 5 y   z - 4 y z - 2 z - 9 y   - 5 y - 1)
(%i3)
>close,fp;
```

ここでは SBCL を使ってコンパイルした Maxima を使っているために最初の表示が KNOPPIX/Math のものと幾分異なっています. それ以外には違いはありません. この例では factor 函数を使って多項式が因子分解できるものか調べていますが, 返ってきた与式は複数の多項式の積になっていません. どうも簡単に既約多項式に分解できる多項式ではないようです. そこで今度は数式処理の Reduce でも試してみましょう. この Reduce も KNOPPIX/Math 2009 以降に含まれています. 起動方法は Maxima の場合と同様です:

```
> fp=popen("reduce -w",1)
> REDUCE, 15-Sep-08 ...

1: write,fp,format="surface=%s;\n",P2String(AlgeS)
> fflush(fp);
write-only text stream at:
  LINE: 1   FILE: reduce -w
>
                  2  4       2  3      2  2        2          2       4         3         2
surface :=  - 7*y *z  - 3*y *z  + y *z  + 5*y *z + 9*y  + y*z  + 2*y*z  + 3*y*z

                                    4       3       2
             + 4*y*z + 5*y + 5*z  + 4*z  + 3*z  + 2*z + 1

2: write,fp,format="%s\n","factorize(surface);"
> fflush(fp);
write-only text stream at:
  LINE: 1   FILE: reduce -w
>
           2  4       2  3      2  2        2          2       4         3         2
{{ - 7*y *z  - 3*y *z  + y *z  + 5*y *z + 9*y  + y*z  + 2*y*z  + 3*y*z  + 4*y*z

                      4       3       2
   + 5*y + 5*z  + 4*z  + 3*z  + 2*z + 1,
```

```
  1}}

3: close,fp

4:
5:
6:
7:
*** End-of-file read
>
```

今度は factorize 函数を用いていますが, 返却値が多項式の積の形にはなっていないことから, どうも複数の多項式の積に分解できない多項式, つまり既約多項式のようです．このようにパイプが使えると Yorick から実に様々なアプリケーションを使って遊ぶことができるのです！ そしてこのようなことが簡単に行える KNOPPIX/Math は宝の山なのです.

お休みはここで終了としましょう．これから多項式の演算処理を色々と入れたいところですが, popen 函数で Maxima や Reduce とも繋ってしまったので, このような処理は Maxima や Reduce に任せることにして, Yorick が得意とする数値計算の例に移ることにしましょう.

11.4　Yorick を使って微分方程式で遊ぶ

11.4.1　微分方程式から差分方程式へ

ここでは被食者・捕食者の個体数に関する有名な方程式である「Volterra-Lotokka の連立微分方程式」で遊びます．この微分方程式は被食者 (シマウマ) の個体数を x, 捕食者 (ライオン) の個体数を y としたときに, 連立微分方程式 11.1 になります:

Volterra-Lotka の被食者・捕食者モデル

$$\begin{aligned} \frac{dx}{dt} &= (A - k_1 y)x \\ \frac{dy}{dt} &= (k_2 x - B)y \end{aligned} \tag{11.1}$$

この連立微分方程式 11.1 の左辺には微分項があって, Yorick でそのまま扱うことはできません．だから, この式を Yorick で処理できるように書き換える必要があり

ますが、 そのためには, まず”微分”を別の式で置換える必要があります. ここで dx/dy は $\lim_{h\to 0}(x(t+h)-x(t))/h$ ですが, h が十分小さいと仮定すれば, dx/dy は $(x(t+h)-x(t))/h$ で近似できます. 同様に dy/dx も $(y(t+h)-y(t))/h$ で近似できるので, 微分方程式 11.1 は方程式 11.2 に書換えられます:

Volterra の連立微分方程式を差分方程式化したもの

$$\begin{aligned} x(t+h)-x(t) &= h(A-k_1y(t))x(t) \\ y(t+h)-y(t) &= h(k_2x(t)-B)y(t) \end{aligned} \tag{11.2}$$

ここで $x(t+h)-x(t)$ を t における x の差分, あるいは階差と呼び, 方程式 11.2 を差分方程式と呼びます. さて, h を固定して $t=0$ から開始するとしましょう. そして $x_k \overset{def}{=} x(kh)$, $y_k \overset{def}{=} x(kh)$ とすれば差分方程式 11.2 は次の式で置換えられます:

数列としての Volterra の方程式

$$\begin{aligned} x_{k+1} &= h(A-k_1y_k)x_k + x_k \\ y_{k+1} &= h(k_2x_k-B)y_k + y_k \end{aligned} \tag{11.3}$$

このように Volterra の連立微分方程式は Yorick でも簡単に計算できる数列 11.3 になりました.

11.4.2 Yorick による解の計算と描画

今度は Volterra の方程式を数列 11.3 として計算する函数を Yorick で構築しましょう. Fibonacci 数の経験を踏まえて手計算で数列を計算する要領で求める函数 Volterra を構築します:

```
func Volterra(A, B, K1,K2, X0, Y0, t1, h){
 local X, Y, Xt, Yt, i, n;
 n = ceil(t1*1/h);
 X=X0; Y=Y0;
 for(i=1;i<n;i++){
      X=h*(A-K1*Y)*X+X;
      Y=h*(K2*X-B)*Y+Y;};
 return([X,Y]);};
```

ここで A, B, K1, K2 が数列 11.3 で現われる各係数. X0 と Y0 が初期値, t1 が開始時間, h が時間の刻幅です. これらの変数の値を定めて早速, 計算をさせてみましょう:

```
> Volterra (1,2,1,1,4,6,0,0.01)
[4,6]
> Volterra (1,2,1,1,4,6,0.1,0.01)
[2.44125,6.61277]
> Volterra (1,2,1,1,4,6,0.2,0.01)
[1.37051,6.48144]
> Volterra (1,2,1,1,4,6,0.3,0.01)
[0.80066,5.87735]
> Volterra (1,2,1,1,4,6,0.4,0.01)
[0.503149,5.11558]
> Volterra (1,2,1,1,4,6,0.5,0.01)
[0.342596,4.35669]
```

このように一々, 時間を指定してゆくのも面倒ですね. そこで一度に 500 点を描画する函数 drawVolterra を次で定義します:

```
func drawVolterra(A, B, K1, K2, X0, Y0, h, color=){
 local i;
 X=X0; Y=Y0;
 for(i=1;i<501;i++){
     X=h*(A-K1*Y)*X+X;
     Y=h*(K2*X-B)*Y+Y;
     pldj,X0,Y0,X,Y,color=color;
     X0=X;
     Y0=Y;};}
```

この函数 drawVolterra は差分方程式から得られた式を基に, 引数で与えた初期値 X0, Y0 に対する数列を 1000 点計算し これらの点列を計算順に結ぶ線分を描きます. この線分の描画では pldj 函数を用いています. ここで, 函数 Volterra の引数に見慣れない表記 color=がありますが, こうすることで, この変数 color がキーワードとなります. つまり, 引数 color は省略可能な引数となるわけです. ここで, 引数 color を省略すると color=0 として処理されるので, pldj 函数で描画される軌道の色は黒になります.
では `fma;drawVolterra(1,2,1,1,4,6,0.05,color="red")` の結果を図 11.3, `fma;drawVolterra(1,2,1,1,[2,2,4,2,1], [1.5,2,4,6,8],0.05,color="red")` の結果を図 11.4 に示しておきます.
これらの結果は Volterra の方程式が $(A, B, K_1, K_2) = (1, 2, 1, 1)$ の場合:

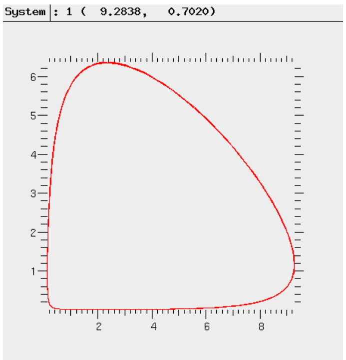


図 11.3: Volterra の方程式の周期解

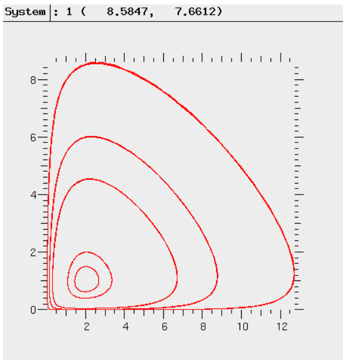


図 11.4: Volterra の方程式の周期解

$$\begin{cases} \dfrac{dx}{dt} = (1-y)x \\ \dfrac{dy}{dt} = (x-2)y \end{cases} \tag{11.4}$$

点 $(2,1)$ は x と y の微分が共に零となる点, すなわち不動点となり, 僅かに不動点からずれると点 $(2,1)$ のまわりを巡る周期解となります．このような不動点のことを「不安定な不動点」とよびます．このように Yorick では引数に配列を指定することで一度に複数の計算ができます.

11.4.3　アニメーション表示

今度は簡単なアニメーションを作ってみましょう．drawVolterra 函数では pldj 函数を使って線分の描画を行っているので, pldj 函数で線分を描画する間に pause 函数を挿入して描画速度を落するとどうでしょう？ この方法で構築した animVolterra 函数を示しておきます:

```
func animVolterra(A, B, K1, K2, X0, Y0, h, color=){
 local i;
 X=X0; Y=Y0;
 for(i=1;i<301;i++){
     X=h*(A-K1*Y)*X+X;
     Y=h*(K2*X-B)*Y+Y;
     pause,50;
```

```
    pldj,X0,Y0,X,Y,color=color;
    X0=X;
    Y0=Y;};}
```

実行してみると判りますが, これだけでも立派なアニメーションになります.

11.4.4　3D-XplorMath による描画

Yorick で Volterra-Lotka 方程式の描画を行いましたが, 世の中にはこのような微分方程式の解曲線を描いてくれるアプリケーションが色々あります. ここでは Java で記述されたアプリケーション 3D-XplorMath-j を使って Volterra の微分方程式の解を表示させてみましょう. この 3D-XplorMath-j は Java で記述された可視化プログラムで, 平面・空間曲線, 曲面, 微分方程式やフラクタルといった数学的対象が可視できます. さらに各種パラメータや視点が容易に変更できます.
ここで KNOPPIX/Math 2010 なら Maxima と同様に画面左下側の を押して knxmLauncher を起動し, 左上にある 3D-XplorMath-J を押すか, 画面左下側の を押して現われたメニューの先頭の Math から 3D-XplorMath-J を選択すると図 11.5 に示すように 3D-XplorMath-J が立ち上ります:

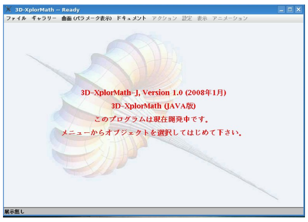


図 11.5: 3D-XplorMath-J の初期状態

もし利用している環境が KNOPPIX/Math 2009 以前であれば画面左下側にある メニューを押して現われる項目で, Java を選ぶと図 11.6 に示す項目が表示されます. :
ここで 3D-XplorMath-J を選択すれば同様に 3D-XplorMath-j が立上がります.
次に左側から二番目のメニューの「ギャラリー」にマウスポインタを移動させて項目を表示させます. すると一番下に「常微分方程式」という項目があり, その中には「1

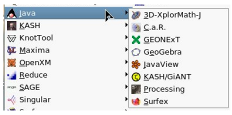


図 11.6: Java の副メニュー

階常微分方程式 (1D)」という項目があります．この様子は図 11.7 に示しておきますが, ここでは「1 階常微分方程式 (2D)」を選択しましょう:

図 11.7: 3D-XplorMath-J の初期状態

すると図 11.5 の初期状態のメニューから左から三番目のメニューが「1 階常微分方程式 (2D)」に切替わります．ここのメニューには図 11.7 に示す項目があり, その中の「ボルテラ-ロトカ方程式」を選択しましょう．すると図 11.8 に示すウィンドウに切替わります:

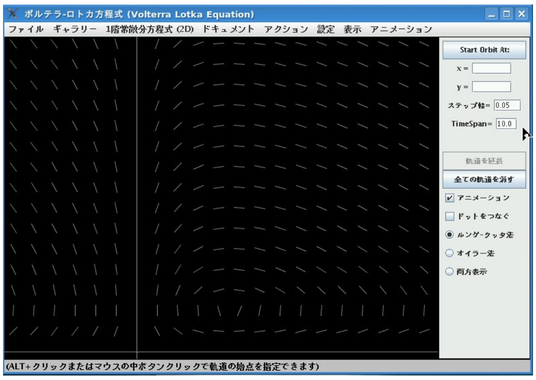


図 11.8: 「ボルテラ-ロトカ方程式」を選択した状態

ここで微分方程式の各係数の変更は右から三番目にある「設定」メニューから「パラ

メータ設定」を選択して現われた図 11.9 に示すウィンドウで行えます:

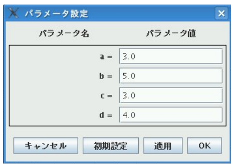


図 11.9: 「パラメータ設定」を選択して現われたウィンドウ

ここで a,b,c,d の値と式 11.1 の A, B, k_1, k_2 の対応は a↔ A, b↔ k_1, c↔ k_2, d↔ B となります．ここでは Yorick の drawVolterra(1,2,1,1,2,1.5,0.05,color="red") と比較するためには図 11.9 の a, b, c, d の欄を ‘1’, ‘1’, ‘1’, ‘2’ で置換し，それから図 11.8 に示すウィンドウ右端の「X=」と「Y=」の空欄にそれぞれ ‘2’ と ‘1.5’ を入れると良いのです．そして 

を押せばアニメーションで解軌道を描きます．最初は点列なので「ドットを繋ぐ」にチェックを入れると点列を繋いでくれます:

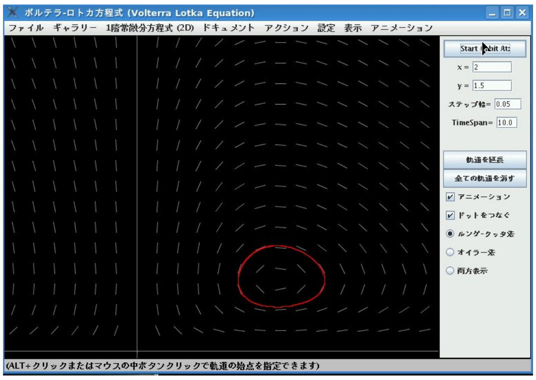


図 11.10: 初期値を指定して計算した様子

ここで直接マウスを使って初期値を指定することもできます．この場合は初期値となる点の指定はマウスポインタを移動させ，3 ボタンマウスであればマウスの中ボタン，ホイール付きであればホイールを押します．二つボタンのマウスであれば二つを同時に押します．初期値が指定されると，3D-XplorMath-J は解軌道の描画を行います．
この 3D-XplorMath-J には他に色々な曲面や曲線の描画がギャラリーにあるので色々試してみると楽しいでしょう．

第12章　Yorickを使った画像処理

First Clown. [sings]	第一の墓掘人 [歌う]
A pick-axe, and a spade, a spade, For and a shrouding sheet: O, a pit of clay for to be made For such a guest is meet. (Throw up another skull.)	鶴嘴一本, おまけに鋤一本, 鋤一本, ふぅ, と経帷子: おぅ, 土塊の終の住処ができあがり こんなお客様にゃどんぴしゃだ. (別の髑髏を放り投げる.)
	Hamlet: 第五幕, 第一場

12.1　はじめに

ここでは Yorick を使った画像処理の話をします．ここでの話は Yorick に限定されません．画像を配列として読込めれば MATLAB 系のシステム全般で行える内容です．まず画像ファイルを Yorick に取込む必要がありますが，この取込では “pnm.i” ライブラリや各種プラグインに含まれる函数を用います．これらの函数で縦が M 画素，横が N 画素の $M \times N$ の大きさのカラー画像を読込むと $3 \times M \times N$ の大きさの配列が得られます．このようにカラー画像が 3 次元配列になる理由は，光が所謂「**光の三原色**」，つまり「**赤 (Red)**」，「**緑 (Green)**」，「**青 (Blue)**」に分解されるからです．このRGB による画像の分解による最初のカラー写真は図 12.1 に示す 1861 年に物理学者の Maxwell[1]によるもので，最初に R, G, B のフィルターを用いて三枚の写真を撮影し，それらを使って今度は逆に R, G, B のフィルターをかけて投光器で重ね合せたものだそうです[2]

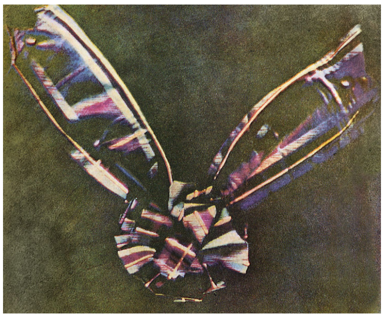


図 12.1: Maxwell のカラー写真 (Tartan_Ribbon)

また 1900 年代になると三色カメラによるカラー写真もあります．たとえばロシアの Sergei Mikhailovich Prokudin-Gorskii の教会の写真[3]は非常に美しいものです．ただし動きの速いものは流石に当時の技術では捉え切れていません．

[1] 電磁気学の有名な方程式「Maxwell の方程式」の Maxwell です

[2] 詳細は `http://ja-jp.colourlovers.com/blog/2008/04/30/ the-history-of-color-in-photography` 等にあります．なお，Maxwell の写真の画像は「`http://ja.wikipedia.org/wiki/`ジェームズ・クラーク・マクスウェル」の中のものを引用しています

[3] 「`http://ja.wikipedia.org/wiki/`ファイル`:Prokudin-Gorskii-09.jpg`」，彼が用いたカメラの仕組は `http://lcweb2.loc.gov/pp/prokhtml/prokcolor.html` を参照

12.2 画像の読込・書込に関連する函数

Yorick には PNM 形式の画像ファイルを扱う “pnm.i” ライブラリが標準で付属しています. JPEG 等の一般的な形式の画像が扱えるパッケージやプラグインには, yorick-z プラグインや yutils プラグインがあり, ここではこれらのパッケージやプラグインが包含する画像の読込・書込に関連する函数についてのみ簡単に解説します.

画像読込・書込に関連する主な函数

構文 (jpeg_read, jpeg_write)
⟨ 画像配列 ⟩=jpeg_read(⟨ ファイル名 ⟩)
⟨ 画像配列 ⟩=jpeg_read(⟨ ファイル名 ⟩, ⟨ 変数 ⟩)
⟨ 画像配列 ⟩=jpeg_read(⟨ ファイル名 ⟩, ⟨ 変数 ⟩,⟨ 配列 ⟩)
⟨ 配列 ⟩ =jpeg_read(⟨ ファイル名 ⟩, ⟨ 変数 ⟩,[0,0,0,0])
⟨ 画像配列 ⟩=img_read(⟨ ファイル名 ⟩)
jpeg_write,⟨ ファイル名 ⟩,⟨ 画像配列 ⟩
jpeg_write,⟨ ファイル名 ⟩,⟨ 画像配列 ⟩,⟨ 文字列 ⟩,⟨ 正整数 ⟩
img_write,⟨ 画像配列 ⟩,⟨ ファイル名 ⟩

jpeg_read 函数: yorick-z プラグインに含まれる画像ファイルの読込を行う函数です. カラー画像ファイルであれば 3 × 幅 × 縦 の大きさの与件型が char 型の 3 次元配列を生成し, 白黒や灰色階調画像の場合は 幅 × 縦 の大きさの与件型が char 型の 2 次元配列を生成します.

ここで jpeg_read 函数で読込まれた画像で構成される配列を A とすると A(1,,) が「赤」, A(2,,) が「緑」, A(3,,) が「青」に対応します. ここで配列の大きさが画像配列として利用可能なものであったとしても, 与件型が char 型でなければ pli 函数による表示ではエラーとなります. そのために函数 char で char 型に変換する必要が生じます.

第 3 引数の変数には画像ファイルに記録された注釈が割当てられます. この引数は第 4 引数を指示する際に省略はできません. また第 4 引数の 4 成分の配列に ‘[0,0,0,0]’ のような正整数以外のベクトルに対して 3 成分の整数配列を返却します. ここで返却される配列の第 1 成分がチャンネル数, 第 2 成分が横幅, 第 3 成分が縦幅となります. ここで正整数ベクトル ‘$[i_0, i, 1, j_0, j_1]$’ を指定すると本来の画像配列を A としたときの $A(.., i_0 : i_1, j_0 : j_1)$ を返却します. これは非常に大きな画像を読込むための対策です[4].

[4]KNOPPIX/Math 2009 や MS-Windows 版の Yorick では, 第 4 引数を本来読込む画像の大きさ以外を指定しても動作しないようです.

img_read 函数:　yorick-z プラグインに含まれる画像ファイルの読込を行う函数です. 生成する画像配列は jpeg_read 函数と同様ですが, 唯一違う点は配列の第 3 添字が逆向きになることです. つまり img_read 函数で読込んだ画像配列を A, jpeg_read 函数で読込んだ画像配列を B とするとき, 'A(..,::-1)==B' の関係があります. img_read 函数は内部では "pnm.i" ライブラリの函数を用いて画像ファイルを PNM 形式に変換して pnm_read 函数を使って読込を実行します. ここで pnm_read 函数は PNM 形式で無圧縮の画像ファイルにのみ対応します. それ以外の形式のファイルは NetPBM パッケージ[5] の変換函数を介在しなければなりません. また ImageMagick の convert 命令で PNM 形式の画像へ変換を行う場合は "-compress none" オプションが必要になります.

jpeg_write 函数:　yorick-z プラグインに含まれる函数で, 配列を指定したファイルに書込む函数です. 第 3 引数の文字列は画像に注釈として付与されます. この注釈の既定値は nil です. そして第 4 引数に画質を定める数値として 0 から 100 までの整数を指定します. この画質の既定値は 75 です.

img_write 函数:　yurils パッケージに含まれる函数で, 画像配列を指定したファイルに JPEG ファイルとして書込みます. キーワードを用いて様々な設定が行えますが, 詳細はオンライヘルプを参照して下さい.

注意事項:　yorick-z と yutils はオプションです. そのために自力で Yorick を入れている方は yorick-z プラグインや yutils パッケージをインストールする必要があります. ここで yutils パッケージは Yorick 言語だけで記述されて PNM 形式以外の画像の読込では "pnm.i" ライブラリや NetPBM パッケージを利用しています. ここで MS-Windows 版の Yorick には最初から yorick-z プラグインが含まれているので yorick-z プラグインの函数を利用する方が良いでしょう. KNOPPIX/Math については KNOPPIX/Math 2009 から双方が含まれていますが, それ以前は yorick-z プラグインが含まれてないので yutils パッケージの img_read 函数を使って下さい.

[5] `http://netpbm.sourceforge.net/`を参照.

12.3 簡単な処理例

12.3.1 jpeg_read 函数による画像の読込

では実際に簡単な画像を読込んでみましょう．そこで歌川国芳の骸骨の絵を読込むことにしましょう．この歌川国芳の絵は Wikipedia から入手できます[6]．また，付録 DVD にも収録されているので §13.7.2 を参照して下さい．

Yorick への画像ファイルの取り込みは非常に簡単です．画像配列 skel にファイル名 "Utagawa.jpeg" の画像を割当てるためには `skel=jpeg_read("Utagawa.jpeg");` と入力するだけで良いのです．もし jpeg_read 函数の代りに img_read 函数を用いているのであれば

`skel=img_read("Utagawa.jpeg");` と入力してください．それから読込んだ画像の表示は pli 函数を用います．ここでは `pli,skel;` と入力してみましょう;

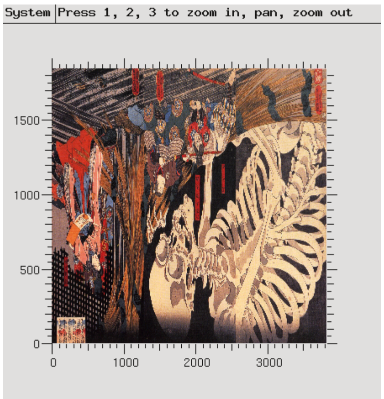


図 12.2: `pli,skel` で得られた画像

図 12.2 に示すような正方形に押し込められ，上下が逆の画像が表示されている筈です[7]．まず絵が正方形に押し込められている状態は limits 函数でキーワード 'square=1' を指定すれば調整できます．上下が逆になる理由は画像ファイル本来の Y 座標と配列の 3 次の次数が jpeg_read 函数で読込むと逆になるからです．こちらは 3 次の次数で逆に並べ直すことで解決できますが，jpeg_write 函数を使って画像配列を JPEG 画像として保存すると今度は Y 座標が逆になるのでまた戻す必要があります．この逆に並

[6] `http://ja.wikipedia.org/`にて「がしゃどくろ」で検索すると良いでしょう．
[7] img_read 函数で読込むと上下は逆になりません．

べる方法として Yorick では添字 "::-1" を用います[8];

```
limits,square=1
fma;
pli,skel(..,::-1)
```

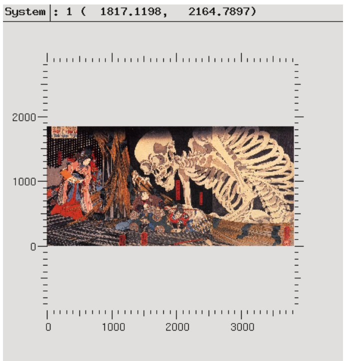


図 12.3: `pli,skel(..,::-1)` で得られた画像

図 12.3 に示すように今度はきちんと表示されていますね．ただ limits 函数のキーワードが 'square=0' の方が見易いので，以降では既定値の 'square=0' を使います．ではこの絵の大きさはどうなっているのでしょうか？ dimsof 函数で調べると判りますが $3 \times 3829 \times 1847$ の大きさの 3 次元の配列になります．どうして 2 次元の画像なのに配列が 3 次元になっている理由は，カラー画像が赤 (Red)，青 (Blue)，緑 (Green) の三色に分解して表現されるからです．これは冒頭の Maxwell の実験に関連します．そして 1 次の添字が 1 の配列が赤，添字が 2 の配列が緑，添字が 3 の配列が青の配列になります．ここでの skel の例では 'skel(1,,)' が赤，'skel(2,,)' が緑，'skel(3,,)' が青の配列になります．実際に通常の場合と 1 次の添字を逆にした場合を観察してみましょう; RGB が BGR になるので，赤が青，青が赤になります．そのために図 12.4 に示す標準の場合は赤みがかかり，図 12.5 では青みがかかっており，さらに左側の滝夜叉姫の裳が赤から青になっています．

ここで赤，緑，青のみを表示させる場合は pli 函数で直接見ても構いませんが，この場合は 0 から 255 までの整数値でしかないので palette を予め "gray.gp" に変更しておくと良いでしょう．この palette の変更がなければ値の大小は地図の高度の様な色使いとなります;

```
> palette,"gray.gp"
> pli,skel(1,,::-1)
> pli,skel(2,,::-1)
> pli,skel(3,,::-1)
```

[8] img_read 函数で読込んだ配列でこの処理は不要ですが試してみると面白いでしょう．

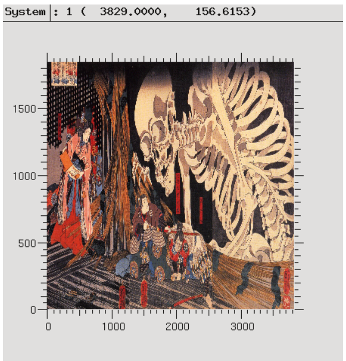


図 12.4: 標準

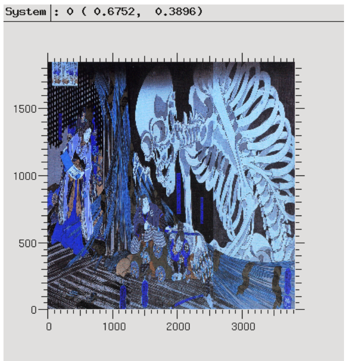


図 12.5: 1 次の添字を逆順

図 12.6 と図 12.7 に標準のパレットと “gray.gp” にパレットを変更したときの様子を示しておきます;

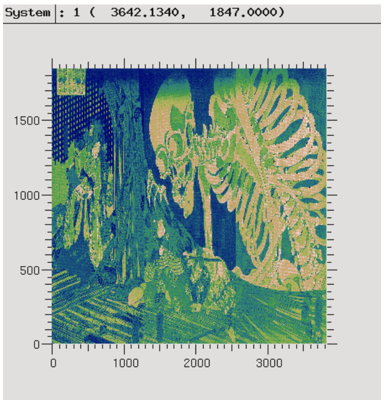


図 12.6: 標準のパレット

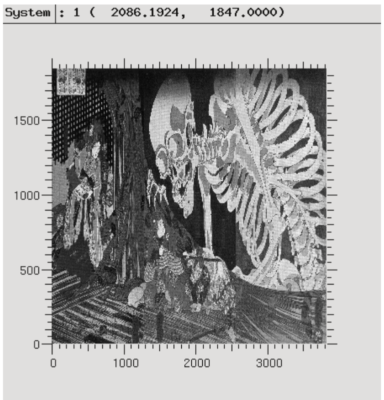


図 12.7: 灰色階調のパレット

図 12.6 は白→緑→青で輝度が表現された独特の表示ですが, 図 12.7 の灰色の階調の方が一般的でしょう.

12.3.2　領域の指定による切出し

ここでは最初に滝夜叉姫を取出してみましょう．領域は横が 1 から 800, 縦が 300 から 1500 とします;

```
yasya=(skel (,,::−1))(..,1:800,400:1500)
 limits ,square=1;
fma;
 pli ,yasya
```

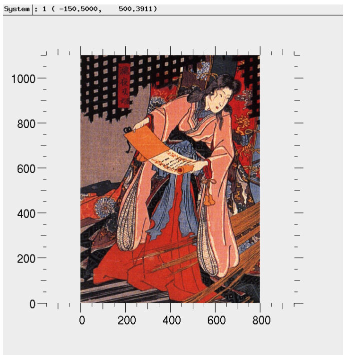


図 12.8: 滝夜叉姫

ここで jpeg_read 函数で取り込んだ配列は 2 次の添字が横軸, すなわち X 軸側, 3 次の添字が縦軸, すなわち Y 軸側となるので, 2 次の添字に領域 ‘1:800’, 3 次の添字に領域 ‘400:1500’ を指定しています.
これで図 12.8 に示すように滝夜叉姫の部分を抜き出しましたがいかがでしょうか？ちなみに MATLAB や Octave でも同様の手法で画像の切り出しが行えます.

12.3.3　述語による切出し

述語を定義してやれば画像の切出しも色々行えます．たとえば円領域の取出を行いたければ, 最初に画像に対応する座標を表現する 2 つの配列を生成します;

```
> dms=dimsof(Skel)
> X1=indgen(1:dms(3))(,−:1:dms(4));
> Y1=transpose(indgen(1:dms(4))(,−:1:dms(3)));
```

ここで生成した配列 X1 と配列 Y1 は画像の R, G, B の各チャンネルの配列と同じ大きさの配列です．配列 X1 は配列 ‘indgen(1:dms(3))’ を複製したもので, 行列に対応させれば全ての行の範囲を ‘indgen(1:dms(3))’ とする行列になります．同様に配列 Y1 は配列 ‘indgen(1:dms(4))’ を複製して得られる配列で, 全ての列の範囲を ‘indgen(1:dms(4))’ とする行列に対応します．このことは pli 函数で表示すると明瞭になります:

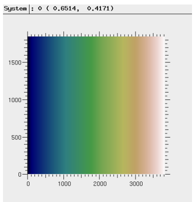


図 12.9: 配列 X1

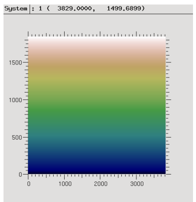


図 12.10: 配列 Y1

さて画像配列 A に対して添字 “(x,y)” に対応する点の XY 座標は, 先程の配列 X1 と配列 Y2 を使うと図 12.11 の対応関係で計算できます;

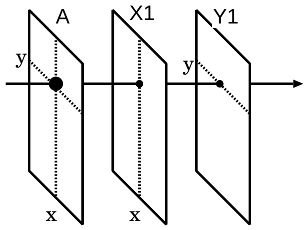


図 12.11: 点との対応

このことにより, 指定した点を中心点とする円内部を返す函数が構築できます;

```
func getCDOMAIN(image,point,radi){
 local dnm,pred,X,Y,Z;
 dnm=dimsof(image); Z=image;
 X=indgen(1:dnm(3))(,,-:1:dnm(4));
 Y=transpose(indgen(1:dnm(4))(,,-:1:dnm(3)));
 pred=((X-point(1))^2+(Y-point(2))^2<=radi^2);
 Z(1,,)=Z(1,,)*pred; Z(2,,)=Z(2,,)*pred; Z(3,,)=Z(3,,)*pred;
 return Z;};
```

この函数 getCDOMAIN では述語 ‘(X-point(1))^2+(Y-point(2))^2<=radi^2’ を用いて R, G, B の各チャンネルで領域を取出します. この函数を実際に試すと図 12.12 に示すように円領域が取出せています;

```
Taki=getCDOMAIN(
 Skel (,,::-1),
  [500,1300],300);
fma;
pli , Taki;
```

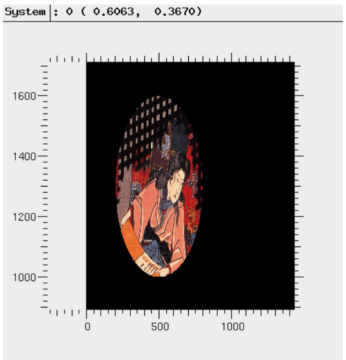


図 12.12: 円領域の抜出し (拡大)

この getCDOMAIN 函数で用いている述語をもっと柔軟に使えるように改良した函数が getDOMAIN 函数です;

```
func getDOMAIN(image, p0, pred, option=){
 extern _Px, _Py, _Q, _Pred;
 local dnm,i,n,X,Y,Z;
 dnm=dimsof(image); Z=image; n=dnm(1);
 _Px=indgen(1:dnm(3))(,,-:1:dnm(4));
 _Py=transpose(indgen(1:dnm(4))(,,-:1:dnm(3)));
 _Q =p0; predx=pred+" _Px _Py _Q "+option;
 funcdef(predx);
 for (i=1;i<=n;i++) Z(i,,)=Z(i,,)*_Pred;
 return Z;};
```

この函数では大域変数_Px, _Py, _Q, _Pred を利用します．まず大域変数_Px, _Py が画像配列の画素 P の X, Y 座標の配列, 大域変数_Q が画素上の点との関係を表現するために用いる固定点 Q の X, Y 座標, 大域変数_Pred が述語計算で得られた真理値が格納される大域変数です．そして getDOMAIN 函数の引数の pred が述語を表現する Yorick の函数名でキーワード option に点 P と点 Q 以外に述語に与える引数を文字列で記述します．

述語の処理では funcdef 函数を用います．この函数は与えられた文字列から函数を構成して処理を行う函数ですが, 文字列には函数名, 変数名と数値や文字列程度が許容されます．funcdef 函数では函数が定義されて処理されますが, 演算子 “=” を用いた

変数への割当・代入は行えません．値の返却が必要ならば pointer を用いるか大域変数を利用する必要がありますが，ここでは処理の確認の必要から大域変数を用います．実例で説明しましょう．まず円領域の述語 Pred1 を次で定義します;

```
func Pred1(Px,Py,Qxy,r){
 extern _Pred;
 _Pred=((Px−Qxy(1))^2+(Py−Qxy(2))^2<=r^2);};
```

この述語 Pred1 は点 P の座標行列 Px, Px と点 Q の座標 Qxy に加え，半径 r を引数としています．そのために半径 r を getDOMAIN 函数のキーワード option に指定しなければなりません;

```
Hero1=getDOMAIN(
 Skel (,,::−1),
 [1500,777],
 "Pred1",
 option="200");
fma;
pli ,Hero1
```

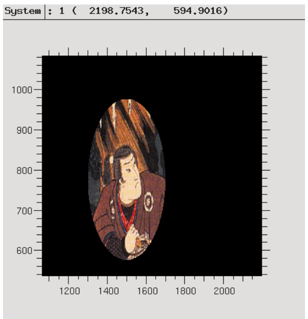


図 12.13: 大宅光圀

getCDOMAIN 函数と同様の処理ができていることが判りますね．この getDOMAIN 函数の最大の長所は必要に応じて述語を定められるので，より複雑な領域の取出ができる点です．そこで複雑な領域を表現する述語を次で定義してみましょう;

```
func Pred2(Px,Py,Qxy,r){
 extern _Pred;
 Px=Px−Qxy(1); Py=Py−Qxy(2); th=pi*2/5;
 L1=(1−cos(2*th))/sin(2*th)*Px+r−Py;
 L2=(cos(4*th)−cos(2*th))/(sin(2*th)−sin(4*th))*
     (Px+r*sin(2*th))+r*cos(2*th)−Py;
 L3=r*cos(th)−Py;
 L4=(cos(3*th)−cos(th))/(sin(th)−sin(3*th))*
     (Px+r*sin(th))+r*cos(th)−Py;
 L5=(1−cos(3*th))/sin(3*th)*(Px+r*sin(3*th))+r*cos(3*th)−Py;
```

```
pL1=(L1>=0); mL1=(L1<=0); pL2=(L2>=0); mL2=(L2<=0);
pL3=(L3>=0); mL3=(L3<=0); pL4=(L4>=0); mL4=(L4<=0);
pL5=(L5>=0); mL5=(L5<=0);
R1=pL1*pL5*mL3; R2=pL3*mL4*mL1; R3=pL1*mL2*pL4;
R4=pL2*mL4*pL5; R5=pL3*mL2*mL5; R6=pL1*pL3*pL5*mL2*mL4;
_Pred=((R1+R2+R3+R4+R5+R6)>0);};
```

述語 Pred2 は指定した星型の領域に点が存在するかどうかを判断することに使える函数です. 図 12.14 に示すように L1 から L5 が星型を構成する直線, R1 から R5 が星型の 5 個の腕の部分, R6 が中心の五角形の領域に対応し, 星型は R1 から R6 までの領域の総和で 0 よりも大の個所として表現されます;

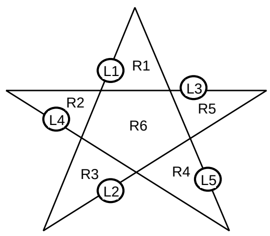


図 12.14: 領域

この函数を使って大宅光圀を英雄 (☆) らしく計算してみましょう;

```
Hero1=getDOMAIN(
 Skel (,,::-1),
 [1500,777],
 "Pred2",
 option="200");
fma;
pli ,Hero1;
```

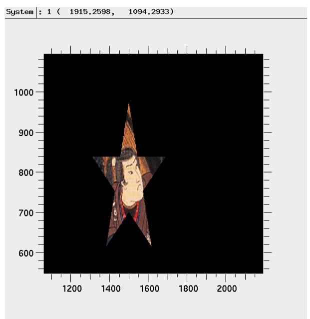


図 12.15: 千両役者 (スター)

さらに互いに重なり合わない領域でも各領域の配列の総和から得られます;

```
> Hime=getDOMAIN(Skel(,,::−1),[500,1300],"Pred1",option="300")
> Hero=getDOMAIN(Skel(,,::−1),[1500,777],"Pred2",option="200")
> Maru=getDOMAIN(Skel(,,::−1),[2000,500],"Pred2",option="200")
> Gai=getDOMAIN(Skel(,,::−1),[1750,1300],"Pred1",option="400")
> fma;pli,Hime+Hero+Maru+Gai
```

この例では滝夜叉姫と骸骨を円領域, 光圀と手下を星型領域として取出し, それらの和を表示させています. この処理結果を図 12.16 に示しておきます;

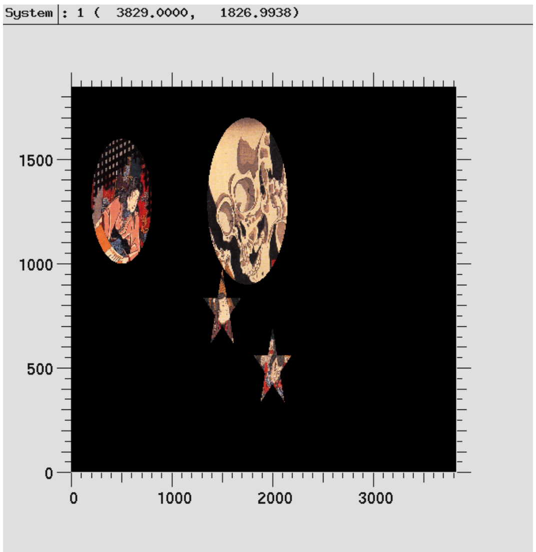


図 12.16: 役者勢揃い

12.3.4 輝度による取出

これだけでは面白くないので今度は骸骨を抜き出してみましょう. 骸骨の色は白で, この白という色は RGB の各配列で共に値が大きい筈です. そこで最初に R, G, B の平均を出しておきましょう;

```
> skel(1,*)(avg)
133.265
> skel(2,*)(avg)
101.802
> skel(3,*)(avg)
```

85.2086

赤が最大で, 以降, 緑, 青の順となっていることから図 12.4 が赤みがかかっている理由が判ります. そこで今度はヒストグラムを計算させてみましょう. そのためにはヒストグラムの計算を行う函数を構築しなければなりません. ここで画像は 256 階調, すなわち 0 から 255 までの整数値の与件として表現さているので $n \in \{0, 1, \ldots, 255\}$ に等しい画素が幾らあるかというベクトルを構築すれば良いのです. 画像配列を skel とするときに ‘skel(1,*)==10’ で赤で階調が 10 となる成分が 1, 他の成分は全て 0 になり, この総和が赤の階調が 10 となる画素の総数になります. つまり `(skel(1,*)==10)(sum)` から階調が 10 となる画素の総数が得られます. この操作を 0 から 255 まで行えば良いのでヒストグラムを生成する函数 getHIST は次で構築できます;

```
func getHIST(A){
      B=A(*);
      ans=indgen(0:255)(-:1:2,);
      for(i=0;i<256;i++){
         ans(2,i)=(B==i)(sum);};
      return(ans);};
```

上の函数を Yorick にそのまま入力するか, ファイル, たとえば “getHIST.i” というファイルに記述して include 函数で読込んで処理をさせましょう. この処理は非力な計算機では時間がかかるので注意して下さい. このグラフ表示の結果を図 12.17 に示しておきますが, この図から判るように赤には 240 付近, 緑は 210 付近, 青は 160 付近に極大値があります;

```
R_hist=getHIST(skel(1,..));
G_hist=getHIST(skel(2,..));
B_hist=getHIST(skel(3,..));
fma;plg,R_hist (2,), R_hist (1,), color="red"
plg,G_hist (2,), G_hist (1,), color="green"
plg,B_hist (2,), B_hist (1,), color="blue"
```

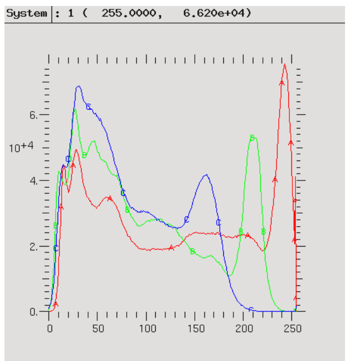


図 12.17: 画像のヒストグラム

なお, ここで定義した getHIST 函数のヒストグラムを計算する個所は, Yorick の函数の, その名前も histogram 函数を使えば高速化が可能です. ただし, この函数では分布は 1 以上の実数値でなければならず, 今回の画像ではそのままは使えません. この辺の工夫は読者の皆さんに練習課題として残しておきましょう.
さて, 赤の閾値を 250, 緑の閾値を 150, 青の閾値を 100 に設定して閾値よりも大きな輝度であれば 1 を, それ以下なら 0 を設定すれば, 全ての閾値で 1 となる点が「白の領域」として良いでしょう;

```
> B1=skel(1,,::−1)>200;
> B2=skel(2,,::−1)>150;
> B3=skel(3,,::−1)>100;
> C1=B1∗B2∗B3;
> fma;pli,C1
```

ここでは配列 B1, B2, B3 に赤, 緑, 青の閾値を越える点を 1, それ以外を 0 とする配列を割当てます. そして ‘C1=B1*B2*B3’ によって論理積の結果を配列 C1 に割当てています. その結果を図 12.18 に示しておきましょう;

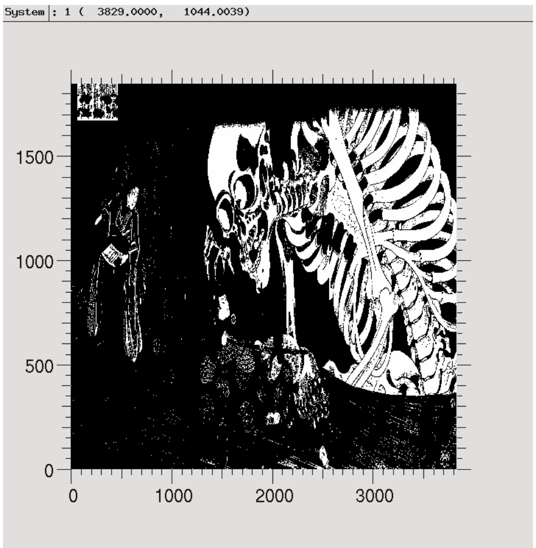


図 12.18: 二値化した画像

それから配列 skel と C1 の積を取れば該当する画像が取出せます;

```
> A1=skel;
> A1(1,..)=C1*skel(1,,::-1);
> A1(2,..)=C1*skel(2,,::-1);
> A1(3,..)=C1*skel(3,,::-1);
> fma;pli,A1
```

‘pli,A1’ の結果を図 12.19 に示します．ここでの計算は赤, 緑, 青の配列に対して C1 との積を計算させたもので, RGC の各閾値を越える個所のみがそのままの値となって, それ以外は 0 になる配列 A1 を最終的に求めています．このような手法は MATLAB 系の言語で利用可能な手法で, for 文のような反復を必要としないために比較的処理が高速に行えることに加え, 表現が非常に簡潔になる点が最大の長所です．
その一方で白くない部分の取出しはどうでしょうか？ ここで白い部分は配列 C1 で網羅されているとするならば, 白くない部分は配列 C1 を除外した個所, すなわち ‘1-C1’ から得られます;

```
> D1=1-C1;
> A2=skel;
> A2(1,..)=D1*skel(1,,::-1);
> A2(2,..)=D1*skel(2,,::-1);
> A2(3,..)=D1*skel(3,,::-1);
> fma;pli,A2
```

この処理による結果を図 12.20 に示しておきましょう;

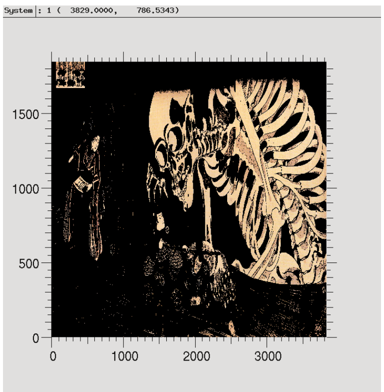


図 12.19: 白い個所の抜き出し

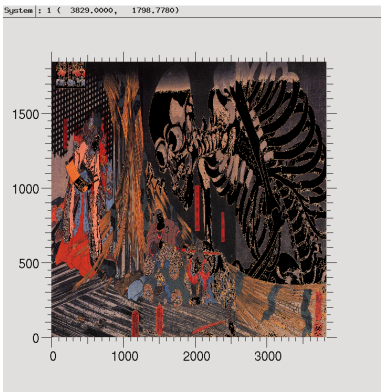


図 12.20: 白い個所の抜き出した後の画像

図 12.19 と図 12.20 を並べると明らかに C1 と D1 は排他的な配列, すなわち C1 と D1 の各成分毎の和が常に 1 になる関係です．したがって C1 と D1 が 1 となる成分を取

出した配列 A1 と A2 の和 'A1+A2' は本来の配列 skel と一致しなければなりません.
実際, 'A1+A2' の配列を pli 函数で表示させると図 12.19 が得られます:

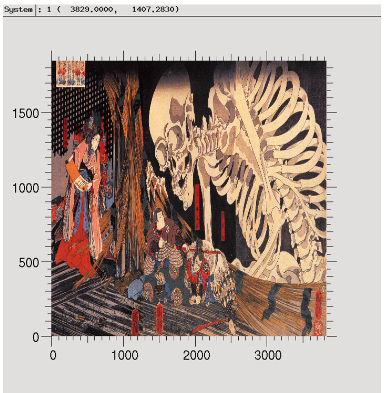


図 12.21: A1+A2

第13章　KNOPPIX/Mathについて

The rest is silence.　　あとは沈黙

Hamlet 第五幕

13.1　はじめに

この本には KNOPPIX/Math 2010 が附属しています．MS-Windows を使うことしか念頭にない方にとって本を必要以上に硬くして項を捲り難くする邪魔者でしかないかもしれませんが, この章を読んだあとには, この DVD は宝の山となるでしょう．なぜなら KNOPPIX/Math は 1-DVD/CD-ROM から起動可能な Linux 環境に数多くの数学関連のソフトウェアと文書を収録し, 同様の環境を個人で最初から整える労力は莫大なものになります．さらに仮想計算機環境を併用することで貴方の計算機環境と KNOPPIX 環境の双方の長所を容易に利用できるからです．実質的には, この本が DVD に附属していると言っても構わないでしょう．なお, この附属の KNOPPIX/Math 2010 はインターネットから入手可能な FTP 版とは商的利用で問題が生じる可能性のあるものを予め除外ている点で異なっていますが, 実際の仕事でも安心して使えるものになっています．
さて, KNOPPIX/Math は DVD から起動可能な Linux と言いましたが, 比較的新しいノート PC では一部のハードウェアに未対応な場合があります．そこで, 仮想計算機を併用することで, 多少の速度は犠牲になるにせよ, 利便性を向上させることが可能になります．そこで, ここでは最初に仮想計算機について簡単に述べることにしましょう．

13.2　仮想計算機環境について

現在の X86 の環境では Multi-core 化が進んでいます．これは CPU の高周波数化とは別の方向の計算機の高速化技術で, 計算機の CPU パッケージ内部に CPU のコアと呼ばれる部分を複数実装させることで一つのチップ上に複数の計算機を実装する方法です．これは CPU の動作クロックの高周波化に伴って周辺回路に与えるノイズの影響が大きな問題として顕在化したことや CPU の排熱の問題といったさまざまな問題によって CPU の動作周波数の高速化が停滞したことが一つの原因です．Multi-core 化によって並列処理を行わせることで周波数を無理に向上させなくてもプログラムを multi-thread 化することで処理性能の向上が図れることが Multi-core 化の大きな利点になりますが, 並列処理に適した処理でなければ有難味は少ないものです．そこで遊休気味のコアを独立した計算機に仕立てて別の処理をさせる手段があり, これが仮想計算機です．この仮想計算機の歴史は古く, 大型計算機で旧機種との互換性を高めるために用いられていました．現在の仮想計算機環境は CPU の Multi-core 化によって計算機の能力に余裕があることに加え, 仮想化支援機能として Intel VT(VT-x, VT-i, VT-d) や AMD-V といった機能も CPU に実装されたこともあって実用的な水準に

なっています. ここで実用的に使える仮想計算機の総数はコア数の 2 倍程度と言われ, その意味では仮想計算機を一つ運営するだけのハードウエア環境は最初から揃っていると言っても良いのです. 結局, 本質的に問題となるのは実装メモリの大きさです.
さて仮想計化にも大きく分けて二種類あります. 一つは「**完全仮想化**」と呼ばれるもので, ハードウエア側の支援を受けて完全に独立した計算機として扱う手法です. この手法が最も自由度が高いものですが, 土台の OS の上に別の OS がそのまま載るために, 専用のドライバがなければ I/O のオーバーヘッド[1] が生じることになるので処理の低下が発生し易くなります.
もう一つが「**準仮想化**」と呼ばれる手法で, 完全に独立した計算機として扱わずに I/O をある程度共通化して用いる手法です. この手法は完全仮想化と比較して自由な構成は行えませんが I/O が共通化されるので逆に処理速度全般の向上が望めます. したがって均質的な仮想計算機はこちらが向いています.
現在, Linux 上の仮想化環境として「**Xen**」が広く用いされています. Xen は計算機本体の CPU が仮想化支援機能を持っていれば完全仮想化, そうでなければ準仮想化に対応しています. ただし, この Xen は初心者が簡単に使えるものでもありません. むしろ, 仮想計算機環境の長所だけを気楽に使いたければ, アプリケーションとして仮想計算機が立ち上げられる方が何かと楽です. そこで気楽に使えるツールとして **VirtualBox** と **VMware Player** を順番に紹介しましょう.

13.3 VirtualBox で KNOPPIX を利用する場合

13.3.1 VirtualBox の概要

「**VirtualBox**」は Innotek GmbH が作成した仮想計算機環境で, Sum Microsystems Inc. に買収されてから Sun xVM VirtualBox がその正式名称となっています. VirtualBox の入手は `http://www.virtualbox.org/wiki/Downloads` から可能で, Open Source 版 (OSE と略記) と商用版の二種類があります. ここで OSE は GPL version 2 に基いてソースコードのみが配布[2] され, 商用版のバイナリには x86 と x64 環境の MS-Windows, LINUX [3] と OpenSolaris, Intel 版の MacOS X 環境に対応したものがあり, 個人的利用と教育的利用, あるいは評価目的の利用であれば無償で利用できます. ここでは VirtualBox が既にインストールされていると仮定して解説を行います.

---

[1] 土台の OS とその上に載る仮想計算機の OS の I/O と二つありますね.
[2] バイナリの OSE がパッケージ化されたディストリビューションもあります.
[3] Debian, Fedora, Mandriva, OpenSolaris, openSUSE, Ubuntu, RedHat Enterprize, openSUSE といった各 LINUX ディストリビューション向けと LINUX 環境全般向けがあります.

13.4　設定方法

VirtualBox 上での仮想計算機の生成と設定について手順を追って説明してましょう. VirtualBox を立ち上げると図 13.1 に示すウィンドウが現れます:

図 13.1: VirtualBox のウィンドウ

このウィンドウのメニュー群の下にアイコンが幾つか並んでいますが, こでらのアイコンを押して現われる Wizard に沿って処理を進めます. ここで仮想計算機の生成では図 13.1 の左上に並んだアイコンの  を押して図 13.2 に示す Wizard を起動させます:

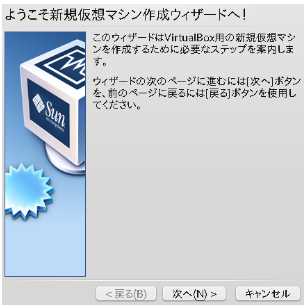


図 13.2: 仮想計算機生成 Wizard

それから 13.2 の右下の 次へ (N)> ボタンを押して図 13.3 の画面に進みます:

図 13.3: 仮想計算機の名称と OS の設定

ここでは仮想計算機の名称と OS を選択してウィンドウ右下の 次へ (N)> ボタンを押せば図 13.4 に移って仮想計算機の記憶容量の設定が行えます:

図 13.4: 仮想計算機の記憶容量の設定

記憶容量は KNOPPIX/Math 向けには最低で 128MB, KDE のような豪華なデスクトップ環境を使いたければ最低 256MB を必要としますが, KNOPPIX は計算機のメモリや書込みの領域一式をここでの記憶容量で賄うためにできるだけ多く設定すると良いでしょう. 記憶容量を指定すると今度はウィンドウ右下の 次へ (N)> ボタンを押して図 13.5 に示す仮想ハードディスクの設定に移ります:

図 13.5: 仮想計算機のハードディスクの設定

ここでの設定は仮想計算機が利用するハードディスクの設定で, 実体はファイルです. もし KNOPPIX だけを起動させるのであれば仮想計算機に割当てられる記憶容量と KNOPPIX の DVD/CD-ROM を読込むための DVD/CD-ROM ドライブ, あるいは

DVD/CD-ROM の ISO イメージファイルのみです．ここで仮想ハードディスクがあれば仮想計算機を停止したあとでも処理結果や諸環境等の保存が行えます．
もしハードディスクが不要であれば「起動ディスク (プライマリマスター)(D)」のチェックを外して 次へ N> を押せば図 13.6 のウィンドウが出ます:

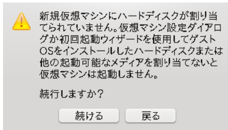


図 13.6: ハードディスクイメージを生成しない場合のメッセージ

この警告の内容は特に気にする必要はありませんが, 続ける を押せば図 13.7 のウィンドウに切り替り, ここで 完了 F を押せば仮想計算機の生成が終了します:

図 13.7: 仮想計算機のハードディスクなしの環境

仮想ハードディスクを新規に作成するのであれば図 13.5 の 次へ (N)> ボタンを押して図 13.8 に示す仮想ハードディスク生成の Wizard に移動します:

図 13.8: 仮想ハードディスク生成の Wizard

このウィンドウで 次へ (N)> ボタンを押すと今度は図 13.9 に移動してハードディス

クのイメージファイルの種類を指定します．一番上の「可変サイズのストレージ (D)」を指定しておけば必要な領域のみを確保するので無駄に膨れ上がることがありません:

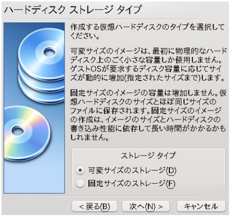


図 13.9: 仮想ハードディスクイメージの型を指定

イメージファイルの型を指定すると 次へ (N)> ボタンを押しましょう．すると図 13.10 に示すハードディスクのイメージファイルの大きさに指定に移動します:

図 13.10: 仮想ハードディスクイメージファイルとその大きさを指定

この設定のあとに 次へ N> ボタンを押しましょう．すると，これから生成する仮想計算機の概要が図 13.11 に示すように表示されます:

図 13.11: ハードディスクイメージファイルの概要

この設定で良ければ 完了 (F) を押して図 13.12 に切替えます:

図 13.12: 仮想計算機の概要

これで仮想計算機のハードディスクのイメージファイルが指定されました．これで良ければ 完了 (N) ボタンを押しましょう．すると図 13.13 に示すウィンドウに移りますが，このウィンドウの右側には生成した仮想計算機の概要が表示されています:

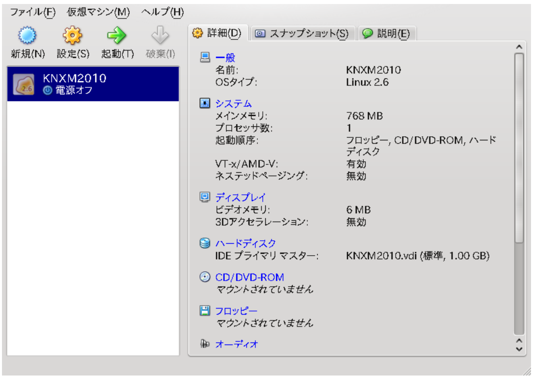


図 13.13: 生成した仮想計算機の概要

今度は仮想計算機の周辺機器の設定を行いましょう．ここで左側のリストから仮想計算機を選択すれば対応する仮想計算機の概要が右側のウィンドウに表示されているので，そこから「CD/DVD-ROM」の箇所をクリックします．すると図 13.14 の画面に切り替わります:

図 13.14: 仮想計算機の CD/DVD-ROM の設定画面

ここで KNOPPIX/Math は CD-ROM/DVD-ROM, あるいは ISO イメージファイルとして提供されます. ISO イメージファイルを用いる利点は CD/DVD-ドライブを用いるよりも読込速度が格段に速いこととネット経由では ISO イメージファイルとして配布されているので入手した ISO イメージファイルをわざわざ CD や DVD に焼かなくて済むことが挙げられます. ISO イメージファイルを利用するのであれば「ISO イメージファイル」のラジオボタンにチェックを入れて, その右側の  アイコンを押せば図 13.15 の画面に移動します:

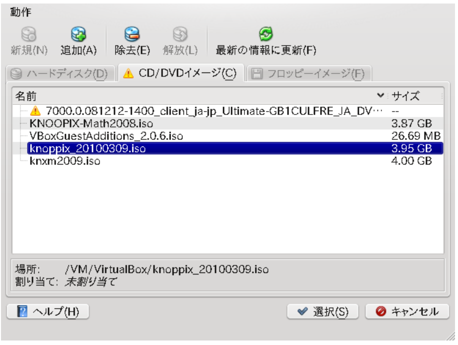


図 13.15: 仮想計算機の CD/DVD-ROM のイメージファイルを選択

このウィンドウの上に並んだアイコンを押せば仮想計算機のハードディスクやフロッピー等のイメージファイルの生成, 指定, 開放や削除が行えます. ここで左上にある  アイコンを押せば仮想計算機にメイディアのイメージファイルが設定できます. これで良ければ 選択 (S) ボタンを押して図 13.16 に戻ります:

図 13.16: 仮想計算機の CD/DVD-ROM 設定画面

そこで $\boxed{\texttt{Ok}}$ を押しましょう．すると図 13.17 に示す内容に切り替わっている筈です:

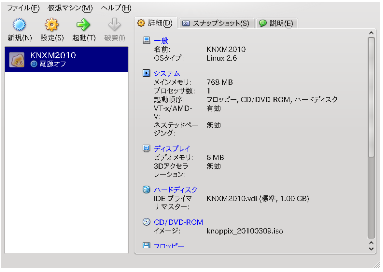


図 13.17: 仮想計算機の設定内容 (変更後)

これで準備が完了しました．仮想計算機の起動は図 13.17 の左側のリストから仮想計算機を指定し，それから左上の  アイコンを押すと指定した仮想計算機が別ウィンドウで起動するだけです．このときに図 13.18 に示すウィンドウが出てきます:

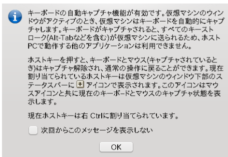


図 13.18: マウスポインタの移動について注意事項

これはマウスのポインタを VirtualBox の仮想計算機ウィンドウ内をマウスでクリックすることで移動させると特定のキーを押すか仮想計算機を停止しない限り，マウスのポインタが移動しないことを知らせるものです．このキーは仮想計算機の右下の枠

に 

のように表示されており, 既定値は 右 Ctrl キーとなっています. この設定は VirtualBox の v「ファイルメニュー」の 環境設定 (P)... から変更できます. 具体的には 環境設定 (P)... を選んで現われたウィンドウの左側の項目で「入力」を選択すれば図 13.19 に切替わってキーの設定できます:

図 13.19: VirtualBox のマウスポインタ切替キーの設定

このウィンドウで ホストキー (K): の右枠にマウスポインタを置いて好きなキーを押せば, そこで指定したキーに切り換えられます.

13.5 VMware Player で KNOPPIX を利用する場合

13.5.1 VMware Player について

VMware Player は VMware, Inc. の製品で, `http://www.vmware.com/products/player/` から入手できます. 以前の VMWare Player は QEMU というアプリケーションを使って仮想計算機を生成したり, 設定ファイルを直接編集する必要がありましたが, 現在の VMWare Player では Virtual Box と同様に Wizard 形式で仮想計算機の生成と設定が行えるようになっています.

13.5.2 設定方法

最初に VMWare Player を起動してみましょう. ちなみに Linux 環境で locale が EUC であれば起動に失敗するようなので, この場合は “LANG=C” にしておくと良いでしょう:

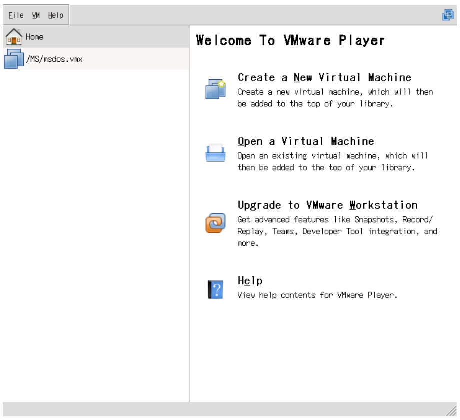


図 13.20: VMWarePlayer の起動画面

新規に仮想計算機を生成するので図 13.20 の左側の「Create a New Virtual Machine」を選択すると図 13.21 に示す Wizard が起動します:

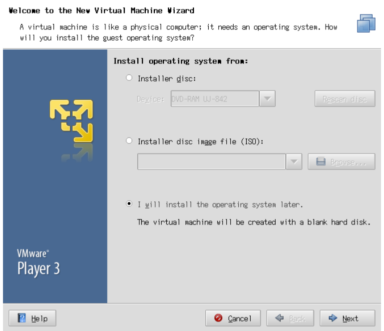


図 13.21: 新規生成の主画面

ここでは一番下の「I will install the operating system later」にチェックを入れて `underlineNext` を押します．すると図 13.22 に移動し，ここでは「Linux」にチェックを入れて下の Version から「Other Linux 2.6.x Kernel」を選択します:

図 13.22: OS の指定

この設定を行うと図 13.23 に移動します:

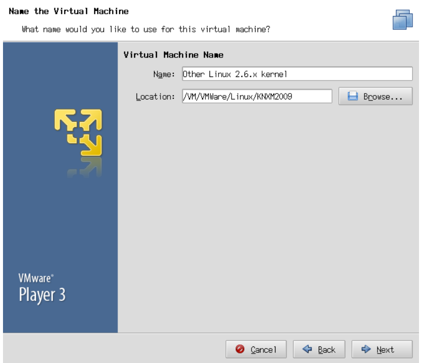


図 13.23: 仮想計算機の置き場所の指定

ここでは仮想計算機のファイルを置く場所を指定します. Location の欄に直接書込むか Browse を使って指示することもできます. この指定が終わると仮想ディスクの指定を行うための図 13.24 に示す Wizard に移動します:

図 13.24: 仮想ディスクの指定

ここでの指定は VirtualBox と同様で, KNOPPIX/Math を起動するだけであれば仮想ディスクは不要ですが, VMWarePlayer では仮想ディスクの大きさは最低 0.1MB が指定されます. この指定を終えると仮想計算機の諸設定を行う図 13.25 に示す Wizard が起動します:

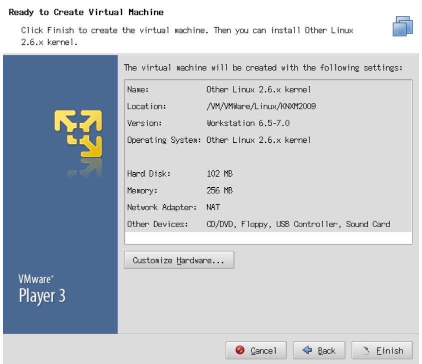


図 13.25: 仮想計算機の設定

ここでの設定は仮想計算機に割当てる記憶容量, CD/DVD ドライブ, サウンドカードや USB コントローラ等の設定が行えます. ここでは ISO ファイルを指定するので Cunstomize Hardware... を押して図 13.26 に示す Wizard を起動します:

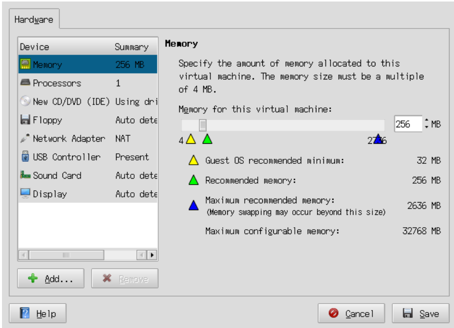


図 13.26: 仮想計算機の詳細設定

ここで左側の Summary より「New CD/DVD (IDE)」を選択し,「Use ISO image:」を選択し, ISO ファイルを下の空欄に直接記入するか Browse を選択して指定します. これで VMPlayer の準備は完了です. この状態で VMWare Player の起動画面を図 13.27 に示しておきます:

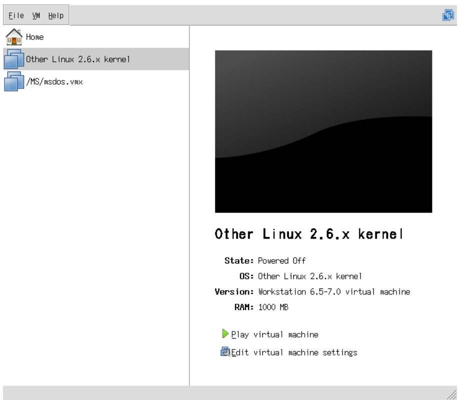


図 13.27: 仮想計算機の起動 (VMPlayer)

左側のリストから該当する仮想計算機を選択し, 右下の Play virtual machine を押すと仮想計算機が起動します VMWare Player の場合, 仮想計算機へのキーの切替は仮想計算機のウィンドウをピックすればよく, ホスト側への切替は Ctrl+Alt でできます.

13.6　仮想計算機と既存環境との共存

ここで openSUSE 上で KNOPPIX/Math2010 を起動させている様子を図 13.28 に示しておきます:

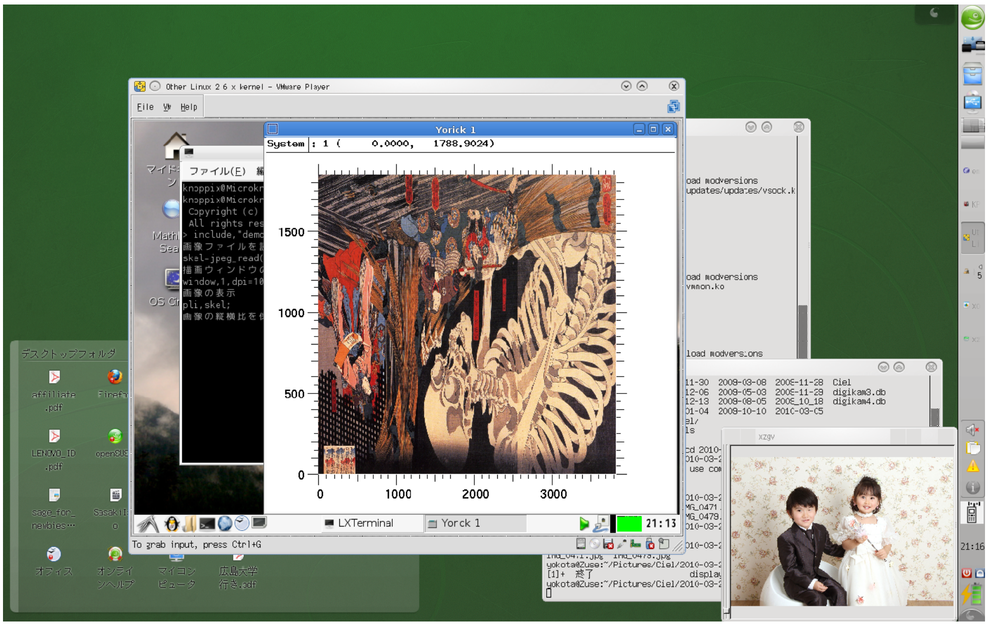


図 13.28: openSUSE と KNOPPIX/Math の共存

中央の大きなウィンドウが仮想計算機です．この様にアプリケーションと同様の状態で起動して従来の環境と併用することができます．VirtualBox や VMWare Player でも仮想計算機のウィンドウを閉じれば仮想計算機がハイバネートされ，次に仮想計算機を立上げればウィンドウを閉じた状態から作業が続けられます[4]．また，KNOPPIX/Math は立上げ時にネットワークに自動的に接続するので，ネットワークを介して本体側と仮想計算機側のファイルの遣り取りもできます．

[4]VirtualBox や VMWare Player の既定値で，電源を落す等の処理に変更することもできます．

13.7 KNOPPIX/Math2010 の使い方

ここで KNOPPIX/Math2010 のデスクトップの様子を示しておきましょう:

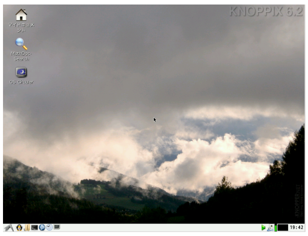


図 13.29: KNOPPIX/Math2010 のデスクトップ

左下端の が「**LXDE メニュー**」というもので, KDE の「**KDE メニュー**」, あるいは MS-Windows の「**スタートメニュー**」に相当します. それから隣にある が「KnxmLauncher」という昔の MacOS やワープロ専用機で見られたアプリケーションラウンチャーを起動させます. このラウンチャーには「**Math**」,「**Internet**」,「**Learn**」,「**Play**」と「**設定**」の 5 種類のラウンチャー画面があり, それぞれの項目で主要なアプリケーションが分類されています. たとえば, Yorick を起動させたければ図 13.30 に示す「**Math**」から Yorick の髑髏のアイコンを見付けてクリックすれば良いのです:

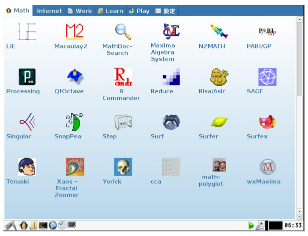


図 13.30: 「Math」の内容

なお, このラウンチャーからは立上げられないアプリケーションを起動させる場合には左端から 四番目の「**Terminal emulator**」のアイコン を押して仮想端末の LXTerminal を起動させて, そこからアプリケーションの起動を行います.

13.7.1　Flash memory へのインストール

KNOPPIX/Math 2010 より USB メモリディスク等の Flash memory へのインストールが容易に行えるようになりました．Flash memory へのインストールを行うと何が良いかと言えば, CD/DVD-ROM よりも読込が高速であることと, Flash memory に個人用のディレクトリを同時に作成しておくことで作業データも保存ができるようになり, Flash memory を持ち歩いていさえすれば何処でも貴方の仕事ができるという訳です．因に現在の KNOPPIX/Math 2010 は 4GB 程度を必要とするので, 8GB 程度の Flash memory があれば 4GB 程度作業領域に利用できます．
インストールは非常に簡単です．LXDE の lxdelauncher をクリックして上にある「設定」を押しましょう:

図 13.31: 「設定」の内容

ここで「**install KNOPPIX to flash disk**」をダブルクリックすると Flash memory 用のインストーラが起動します．ここでインストーラを起動するとディバイスが幾つか現われている筈です．インストールでは媒体のフォーマットを行うので指定したディバイスに保存されたデータは消えてしまいます! 内蔵ディスクは現在 ATA のものが多いので ATA の名前があるディバイスを決して指定しないようにして下さい。

13.7.2　Yorick の例題について

KNOPPIX/Math 2010 の DVD には Yorick 向けの例題を収録していますが, KNOPPIX/Math 2009 までは locale が ja_JP.eucJP だったために文字コードを EUC にしており, KNOPPIX/Math 2010 の UTF8 とは異なっています．そのために日本語の文字化けが生じます．ここで例題ファイルは`/usr/share/doc/knoppix-math-doc/ja/ponpoko/`の`yorick_DEMO.tgz` です．このファイルは tar を使って

```
tar xvf /usr/share/doc/knoppix-math-doc/ja/ponpoko/Yorick_DEMO.tgz
```

で展開可能で, 展開すると Yorick_DEMO というディレクトリと複数のファイルが現われます. ここで文字コード変換は nkf でできますが, 流石にデモファイルを一々変更していては非能率なので次の簡単なシェルスクリプト `toUTF8` を利用します:

```
tar xvf /usr/share/doc/knoppix-math-doc/ja/ponpoko/Yorick_DEMO.tgz
cd Yorick_DEMO
if ! [ -d "Trans" ];then mkdir Trans; fi;
ls * | awk '/.i/{print "nkf -w8",$1 ">Trans/"$1;}
           /.jpeg/{print "cp",$1,"Trans/"$1}'|sh
```

上記の内容のファイルをホームディレクトリに記述しておけば, 仮想端末から `sh toUTF8` と入力すればシェルスクリプトの内容が実行されます. このシェルスクリプトでは最初に tar を使って書庫ファイルを展開し, それから Yorick_DEMO ディレクトリに移動します. ここで Yorick_DEMO ディレクトリに Trans ディレクトリが存在していなければ Trans ディレクトリを生成し, それから Yorick のデモファイルを UTF8 に変換したものと JPEG ファイルを Trans ディレトリに送り込むという操作を行っています. このシェルスクリプトを実行したあとで `cd Yorick_DEMO/Trans` と入力して Trans ディレクトリに移動し, それから Yorick を起動して `include,"Demo.i"` と入力すると Yorick はデモを順番に実行します. 1 つの処理を終えると rdline 函数を使って停止するようにしているので適当なキーを押して下さい.

13.7.3　KNOPPIX-Math-Start

さて，KNOPPIX/Math 2010 を利用する上で重要な「機能」が KNOPPIX-Math-Start です．この KNOPPIX-Math-Start は LXDELauncher では 

です．この KNOPPIX-Math-Start から開かれたページを図 13.32 に示しますが，ここでは収録アプリケーションの概要とリンクが記載されています．

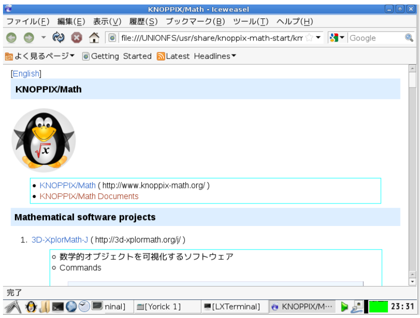


図 13.32: KNOPPIX-Math-Start

また，文書の先頭にある `KNOPPIX/Math Documents` は KNOPPIX/Math 2010 に収録したディレクトリ `/usr/share/knoppix-math-doc/ja` へのリンクになっています．このリンク先のディレクトリの様子を図 13.33 に示しておきます：

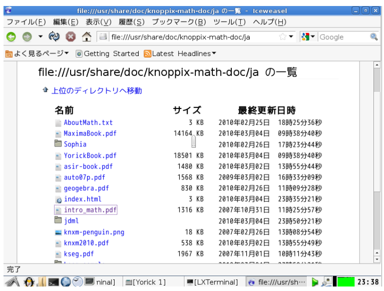


図 13.33: KNOPPIX-Math-Startja

このディレクトリに含まれているファイルの概要は `index.html` に記述されているのでここでは詳細は述べませんが，Maxima の優れた入門書である中川さんの「Maxima 入門ノート」，私の解説書になりますが「はじめての Maxima(α版)」，「たのしい

Yorick」等の PDF 文書や資料があります．ここでは特に重要な jdml について述べておきましょう．

13.7.4 JDML

jdml は "Japan Digitak Mathematics Library" が示すように日本国内の数学系雑誌の情報を集約して再構築する活動の 1 つの成果物です．JDML には大学や研究所等が出している雑誌に掲載された論文の情報が収録され, 論文の PDF をリンク先から入手することができます．ここでは図 13.34 に "index.html" を開いた様子を示しています:

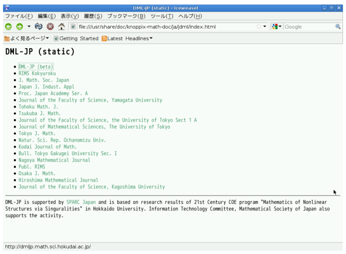


図 13.34: JDML/index.html を開いたところ

ここで "DML-JP(beta)" から `http://dmljp.math.scihokudai.ac.jp/`に飛ぶことができて, 著者や項目などで論文の検索を行うこともできます．たえば「knot alexander」で検索した結果を図 13.35 に示しておきましょう:

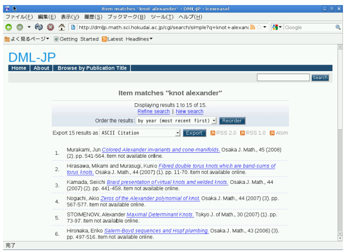


図 13.35: "`knot alexander`" での検索結果

13.7.5　KNOPPIX/Math 上での全文検索

KNOPPIX/Math のデスクトップの左上に  というアイコンがありますね．このアイコンをクリックすると Namazu による KNOPPIX/Math 全文検索システムが立ち上がります:

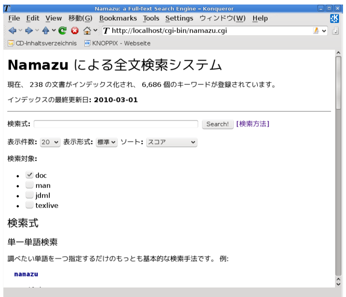


図 13.36: MathDoc-search による全文検索

ここで検索式の箇所で，たとえば「`Frege 概念記法`」と入力して Enter キーを押せば該当語句を含む文書の検索を行います．ただし，バイナリファイルであれば該当箇所へのリンクとは限らず，開いて自分で捜す必要があります．先程の例でリンク先の `MaximaBook.pdf` を開いて該当箇所を自分で捜した結果を図 13.37 に示しておきます:

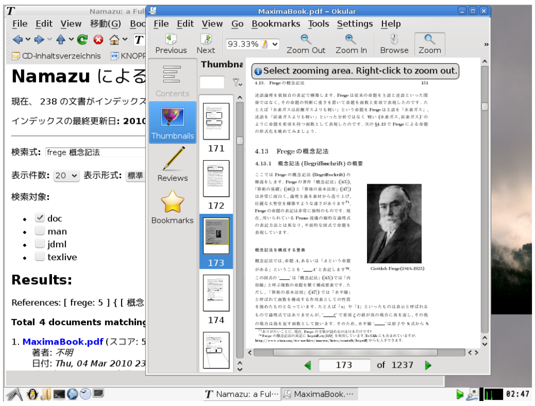


図 13.37: 該当箇所を見付けたところ

このようにマニュアル類を全て自分で調べなくても絞り込みが行えるのです．

以上の説明からもお判りのように, KNOPPIX/Math は数学アプリケーションを集積した Linux 環境だけではなく, 数学に関連する文献やその情報を集めた「数学支援」環境, すなわち, 数学を楽しむための**「数学の玩具箱」**なのです.

最後に

さて, このマイナーな言語である Yorick を皆さんは楽しむことができたでしょうか？この Yorick はマイナーな言語ですが, マイナーであることが非実用的であることを意味しません. むしろ, コンパクトで使い易い優れた言語であると感じていただければ, この本の目的の半分は達せられたと言えるでしょう.
もう 1 つは KNOPPIX/Math に含まれるアプリケーションを活用することの可能性を感じて頂ければ, この本の残りの半分の目的も達せられたことになります.
一般的に OpenSource のアプリケーションは「GUI の貧弱さ」,「総合的な機能の弱さ」を問題点として挙げられることが多く見受けられます. しかし, OSS の優れた点は一芸に秀でた複数のアプリケーションを束ねることで, 利用者が本当に必要とする非常に強力な環境を創り出すことが可能な点でしょう. この構成方法はある意味, オフィスもののように全てを自分に取り込む方向とは逆で, 処理すべき対象が中心にあって利用者は自分が必要なものを取捨選択する方式です. この大規模なシステムの一例として既存のアプリケーションを Python を糊にして繋ぎ合せて構成された数式処理システム SAGE を挙げておきましょう.
ただ, このような複雑で大規模なものでなくても, ちょっとした計算でも, パイプラインで繋いだり, 中間ファイルを利用したり, もっと原始的に式をウィンドウ間でカット&ペーストといった処理だけでも, 実に様々な処理が行え, そして利用者の知識や技術の蓄積に従い, システムを育てて行くことも可能なのです.
そして, これらのアプリケーションを色々と使いこなすことで, 計算機言語等の計算機環境を習得することに労力を費やすのではなく, 本来の目的である筈の数学を堪能して頂ければと願っています. と, 言いたいことを全て言ったので, この本を終えるにあたり, 次の言葉で締め括りたいと思います;

Wovon man nicht sprechen kann, darueber muss man schweigen.	語り得ぬものについては, 沈黙しなければならない. Wittgenstein Tractatus Logico-philosophicus

関連図書

[1] 大石進一, 精度保証付き数値計算, コロナ社, 2000.

[2] 落合豊行,C 言語による数学解析, 近代科学社, 1988.

[3] 桂田祐史,IEEE754 倍精度浮動小数点数のフォーマット
`http://www.math.meiji.ac.jp/~mk/labo/text/ieee_format/`

[4] 熊ノ郷準, 擬微分作用素, 岩波書店, 1992.

[5] シェークスピア (著), 野島秀勝 (訳), ハムレット, 岩波文庫, 2002.

[6] フレーゲ, フレーゲ著作集 3 算術の基本法則, 勁草書房,2000.

[7] 夏目漱石, 我輩は猫である, 岩波文庫.

[8] 新開謙三, 疑微分作用素, 裳華房,1994.

[9] 皆本晃弥, IEEE754 と数値計算,
`http://www.ma.is.saga-u.ac.jp/minamoto/doc/kyudai.pdf`

[10] 牧野, 円周率 100,000,000 桁表, 暗黒通信団,2007.

[11] 吉野邦生, 荒井隆行, デジタル信号と超関数, 海文社, 1995.

[12] 横田博史, はじめての Maxima, I/O Books, 工学社, 2006.

[13] 横田博史, はじめての Maxima 改訂 α 版 (KNOPPIX/Math に収録)

[14] David Bailey, Peter Borwein, Simon Plouffe,
On the rapid computation of various polylograithmic constants, 1991.
(`http://crd.lbl.gov/~dhbailey/pi/`)

[15] G. J. Brose, MATLAB 数値解析,Ohmsha, 1998.

[16] MathWorks 日本: http://www.mathworks.co.jp/

[17] MathWorks によるドキュメンテーション:
http://www.mathworks.com/access/helpdesk_ja_JP/help/helpdesk.html

[18] GNU Octave のサイト: http://www.gnu.org/software/octave/

[19] Scilab のサイト: http://www.scilab.org/ja/

[20] Sun Microsystems ドキュメント: http://docs.sun.com

[21] Surfer の入手先: http://www.imaginary2008.de/surfer.php?lang=en

[22] Yorick の公式サイト: ftp://ftp-icf.llnl.gov/pub/Yorick/

[23] Yorick の非公式サイト: http://www.maumae.net/yorick/doc/index.php

[24] rlwrap のサイト: http://utopia.knoware.nl/~hlub/uck/rlwrap

[25] Sun xVM VirtualBox のサイト: http://jp.sun.com/products/software/virtualbox/

[26] VirtualBox のサイト: http://www.virtualbox.org/

[27] WMware, Inc. の商品紹介ページ: http://www.vmware.com/jp/products/player/

[28] Hamlet に出てくる Yorick について: http://en.wikipedia.org/wiki/Yorick_(Hamlet)

[29] 相馬の古内裏 http://ja.wikipedia.org/wiki/歌川国芳

索 引

函数