情報処理 II 資料

Mathematica 入門

桂田 祐史

2004 年 7 月 1 日

この文書 (の訂正版) は

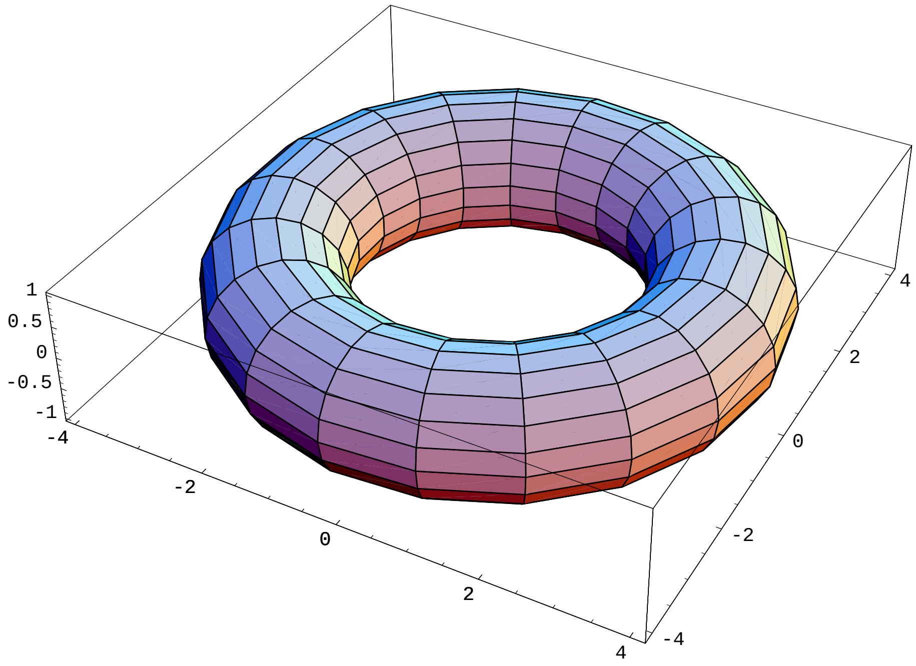
`http://www.math.meiji.ac.jp/~mk/syori2-2003/mathematica.pdf`

でアクセスできます。

代表的な数式処理系である Mathematica（マセマティカ）を体験しましょう。数式処理でどういうことが出来るのか大体の雰囲気をつかんで、今後の学習・研究のヒントにしてもらうのがねらい。

1 Mathematica ってこんなもの

以下の例は、数学科の計算機である oyabun にログインして、Mathematica を実行してみたものです (Windows 環境での実行例ではありませんが、本質は同じです)。ここではプログラムなどは書かずに、式を順次入力して計算結果を表示させています。



```
oyabun% math
Mathematica 4.0 for Solaris
Copyright 1988-1999 Wolfram Research, Inc.
 -- Motif graphics initialized --

In[1]:= 1/2 + 1/3                    ← 分数計算

        5
Out[1]= -                            → ちょっと見難いですけどね
        6

In[2]:= a={{0,1},{6,1}}              ← 行列の入力

Out[2]= {{0, 1}, {6, 1}}

In[3]:= Eigenvalues[a]               ← 行列の固有値の計算

Out[3]= {-2, 3}

In[4]:= Eigenvectors[a]              ← 行列の固有ベクトルの計算

Out[4]= {{-1, 2}, {1, 3}}

In[5]:= Expand[(x+y)^6]              ← 式の展開

         6      5         4  2       3  3       2  4        5    6
Out[5]= x  + 6 x  y + 15 x  y  + 20 x  y  + 15 x  y  + 6 x y  + y

In[6]:= N[Pi,50]                     ← 円周率 50 桁

Out[6]= 3.1415926535897932384626433832795028841971693993751

In[7]:= Integrate[Log[x],x]          ← 不定積分

Out[7]= -x + x Log[x]

In[8]:= Plot3D[x^2 - y^2, {x,-1,1}, {y,-1,1}]  ← グラフ

Out[8]= -Graphics-                   → ここで画面に図が表示されます

In[9]:= Solve[x^3+2x==1,x]           ← 3 次方程式を解かせてみる

  結果は一見に価するけれど、紙を食うのでカットします。
```

```
In[10]:= ParametricPlot3D[{Cos[t](3+Cos[u]),Sin[t](3+Cos[u]),Sin[u]},
         {t,0,2Pi},{u,0,2Pi}]         ← トーラスを描かせる。
```

2　数式処理とは

プログラミング言語 (計算機言語) の中には、数値や文字だけでなく、

数式をデータとして扱うことの出来る「数式処理言語」

と呼ばれるものがあります。数式処理言語を使えるシステム (ソフトウェア) を数式処理系と呼びます。数式処理系としては Mathematica 以外に $\overset{\text{リデュース}}{\textbf{REDUCE}}$, $\overset{\text{メイプル}}{\textbf{Maple}}$ などが有名です。

C のようなプログラミング言語は、プログラムの中では「数式」を書けますが、関数 `scanf()` や `printf()` 等で入出力可能なデータは、数や文字列だけで、例えば `-2/5` のような分数式の入力は出来ません。またグラフを描くプログラムを作る場合に、範囲や、分割数の指定等は実行時に入力出来ても、グラフを描こうとしている関数自体は (普通の方法では) 入力できず、プログラムの中に自分で埋め込むしかなかったわけです。そういう意味では C は不自由な言語であると言えます[1]。

3　この講義で用いる Mathematica

C 言語等は国際規格があり、無償で利用できる処理系もありますが、Mathematica は Wolfram Research という一企業の所有物で、処理系は同社が作成・販売しているものしかありません。予算の関係で、利用できる個数が限られています。

2004 年度の情報科学センターには、Mathematica 4 のライセンスが 40 あるそうですので、これを使います。(万一、出席者数が多くてあぶれる人がいた場合には、数学科の Mathematica 4 を利用します。)

4　基本的な使い方 (工事中)

4.1　起動・終了

情報科学センターパソコンでの **Mathematica** の利用法

1. スタート・ボタンをクリック
2. プログラム → Mathematica 4 → Mathematica 4 を選択
3. 画面左側に現われる “Untitled-1” というウィンドウに、キーボードからコマンドを入力して、最後に $\boxed{\text{Shift}}+\boxed{\text{Enter}}$ をタイプする。
   (コマンドの入力に適宜、テンプレート・キーを用いると便利。)

---

[1] もちろん不自由さを補って余りある大きな利点があるから、現在でも盛んに使われているわけです。例えば、実際の処理系の数値計算の速さで比べると C が圧勝します。原理的には一つのプログラミング言語があれば、どんな計算でも出来るはずなのですが、実際的な意味で万能のプログラミング言語と呼べるものは存在せず、適材適所を心がけることが重要です。みなさんも、あまり一つの言語、一つのシステムにこだわらずに、機会があったら色々なものを勉強してみましょう。

4.2 起動後の基本操作

- “`In[番号]:= `” というプロンプトに対してコマンドを入力。最後にリターン。
- “`Out[番号]= `” に続いて結果が出る。

入力に関して便利な記号として

- `%`　直前の結果
- `%%%···%` (k 個重ねる)　k 回前の結果
- `%`n (n は自然数)　Out[n] と同じ (n 番目の結果)。

以下のコマンドは昔は便利でしたが、現在はほとんど必要ないでしょう。

- `Edit[]` () 環境変数 EDITOR で指定したコマンドを呼び出す)
- `Edit[式]` (式を変更するためにエディターを呼び出す)
- `EditIn[]` (直前の `In[]` の編集), `EditIn[番号]` (`In[数]` の編集)
- `EditDefinition[f]` 関数 `f` の定義の編集

4.3 ファイル入出力

ファイルからの入力 `<< ファイル名`　ファイルの内容をキーボードからの打鍵の代わりに入力

ファイルへの出力 `式 >> ファイル名`　式の値をファイルに出力

ファイルへの追加出力 `式 >>> ファイル名`　式の値をファイルに追加出力
(>> の場合は古い内容は上書きされてしまう。)

ファイルの内容表示 `!!ファイル名`

外部コマンドの実行 `!外部コマンド名`

外部コマンドの出力を入力 `<< "!外部コマンド名"`

式の内容を外部コマンドに与える `>> "!外部コマンド名"`

関数定義をファイルへ出力 `Save["ファイル名", 関数名1, 関数名2,...]`
上に述べたように後から `<< ファイル名` で入力可能。
`In[]`, `Out[]` も関数なので、`Save["todaymath.txt", In, Out]` とすると、`In[ ]`, `Out[ ]` が `todaymath.txt` というファイルに出力されます (昔はレポート作成に利用しましたが、現在ではノートブックがあるので不要でしょう)。

その瞬間のすべての状態をファイルに出力 `Dump["ファイル名"]`
後から `math -x ファイル名` で再開できます。

5　簡単な文法 — 電卓的な使用のために

最初の「Mathematica ってこんなもの」を試してみた後は、簡単な文法と、関数を知るだけでかなり便利に使えるはずです。

5.1　かっこ

- （ ）は結合順位をグループ化する普通のかっこ。
- `[ ]` は関数 (コマンド) に渡す引数を示す。
- { } はリストを作るかっこ (リストについては後述)。
- `[[ ]]` はリストの要素のアクセス (これも後述)。

5.2　変数 (名前) の宣言、変数への代入

- 組み込みの名前は先頭が大文字。ユーザーはなるべく小文字を使うことを奨励されている (衝突を回避するため) 。
- a=式, a=b=式, 等で代入。
- `?Global`*` とすると現在何を定義しているか表示する。
- `?*Sin*` とすると、“Sin” を含む名前の一覧が表示される。
- `a=.` または `Clear[a]` で変数 a を解放 (内容を除く — 名前は残る)。すべての変数の内容を解放する `ClearAll[]` は便利。
- `Remove[a]` 名前まで込めて完璧に除く。

不要になった変数 (または後で出て来るユーザー定義の関数) は、`Clear[名前]` または `Remove[名前]` 等でこまめに削除しておくことを勧めます (一度使った変数に残っているゴミのせいで首をひねることが良くあります)。

変数の定義は不要になったら消さないと副作用が出る

```
In[1]:= f=x^2+2x+3


                         2
Out[1]= 3 + 2 x + x

In[2]:= x=1

Out[2]= 1

In[3]:= f

Out[3]= 6

In[4]:= D[f,x]

General::ivar: 1 is not a valid variable.

Out[4]= D[6, 1]

In[5]:= Clear[x]

In[6]:= D[f,x]

Out[6]= 2 + 2 x
```

5.3 演算子

- 四則演算は普通の記号で OK。`a+b`, `a-b`, `a*b` または `a b` (空白抜きで `ab` は一つの名前。`a5` も一つの名前だが、`5a` は `5` かける `a` の意味), `a/b`.
- べき乗 `a^b`, 階乗 `a!` もあります。

5.4 組み込み関数

`Sqrt[x]`, `Exp[x]`, `Log[x]`, `Log[b,x]`, `Sin[x]`, `Cos[x]`, `Tan[x]`, `ArcSin[x]`, `ArcCos[x]`, `ArcTan[x]`, `Abs[x]`, `Round[x]`, `Random[]`, `Max[x,y,...,z]`, `Min[x,y,...,z]` のように普通のプログラミング言語にそなわっているような関数はもちろん、色々なものがあります (いざとなったらマニュアルや書籍を見て下さい)。

以下、重要なことを (重複をいとわず) いくつか紹介します。

- 関数の引数を囲む括弧は `[ ]`

- システム組み込みのものは名前の先頭の文字が大文字です。
- `?文字列*` で一覧表が出ます。さらに `?関数名` や `??関数名` でその関数の説明が出ます。
- 関数がどういうオプションを持っているか見たい場合は `Options[]` を用います。

`Options[Plot]`	`Plot` のオプション全部を表示
`Options[Plot, PlotRange]`	`Plot` の PlotRange オプションを表示

- オプションを一時的に変更するには “`SetOptions[]`” を用います。

5.5　無限多倍長整数 (小数も任意精度まで)

整数や有理数等は (可能な限り) 誤差のない計算をします。小数に対しては有限精度の計算となります。その場合 `Sin[]` などの関数の精度はデフォルトでは C 言語の double と同程度 (つまりパソコンや WS では 10 進 16 桁弱) ですが、精度はかなり自由に指定できます。

整数や有理数については、特に意識せずに使うことが出来ます。`100!` や `2^100` などで試せます。

`Sqrt[2]` などの値を小数で近似する場合は、`N[Sqrt[2], 100]` のように精度を指定できます。

5.6　分数

`1/2+1/3` のようにして分数が入力できて、分数のまま計算できます。

5.7　文字式 (多項式、分数式、その他)

```
2 f[x] + 3 f[x]
p1 = Expand[(1+x)^10]            多項式の展開結果を変数に代入
p2 = (1+x)^3
PolynomialQuotient[p1,p2,x]      多項式の商
PolynomialRemainder[p1,p2,x]     多項式の剰余
PolynomialGCD[p1,p2]             最大公約多項式
Apart[1/(x^3-1)]                 部分分数への分解
Remove[p1,p1]                    おそうじ
```

5.8　定数

`Pi`(円周率 π), `E`(自然対数の底 e), `Degree`($= 180/\pi$), `I`(虚数単位 $i = \sqrt{-1}$), `Infinity` (無限大 $+\infty$), `ComplexInfinity` (複素平面の無限大), `GoldenRation` (黄金比) などの定数があらかじめ定義されています。

5.9　ルートなど

ルートなどのシンボリックな数も使えます。例えば `Expand[(-1+Sqrt[3]I)^3]` など。

5.10　近似値の計算

`(1+Sqrt[3])^2; N[%,100]`	直前の結果を 100 桁
`N[Pi,100]`	
`N[E,100]`	
`2^100 // N`	
`N[Sin[30 Degree]]`	$\sin 30^\circ$ の近似値

5.11　整数 (素因数分解、素数判定)

`n=2^2^5+1`	$n = 2^{2^5} + 1 = 4294967297$
`PrimeQ[n]`	n は素数かどうか判定する
`FactorInteger[n]`	n の素因数分解
`Mod[123456,123]`	123456 を 123 で割った余り
`GCD[96,18]`	最大公約数
`Remove[n]`	おそうじ

5.12　複素数

`z=(3 + 4I) (1 + 2I)`	$(3+4i)(1+2i)$
`Re[z]`	実部
`Im[z]`	虚部
`Conjugate[z]`	共役複素数
`Abs[z]`	絶対値
`Arg[z]`	偏角
`ComplexExpand[E^(Pi I/6)]`	$a + ib$ の形にする
`Remove[z]`	おそうじ

5.13　結果の簡単化

計算結果が自分の望んでいる形に表されないことがしばしばあります。まず `Simplify[]` という関数を覚えておきましょう。結果が複雑だったら、取りあえず `Simplify[%]` として、「直

前の結果の簡単化」を試みるのが良いでしょう(変わらないことも多いですが)。ちなみに、通分は `Together[]`, 因数分解は `Factor[]`, 展開は `Expand[]` です。例えば

```
y=1/(1+x)+1/(1-x)
Together[y]
Simplify[y]
Factor[y]
Remove[x,y]
```

その他に `FullSimplify[]` などがある。

5.14 一時的な代入

式に含まれる変数に一時的に値を代入して式の値を求めたい場合は、式 /. 名前 -> 値 とします。具体的には、例えば

```
y = x^2
y /.  x->1
y                y 自体は変っていない
Remove[x,y]
```

一度に複数の代入をするには、括弧{ }でくくって、式 /. {名前 -> 値, 名前 -> 値,...} とします。

```
(x + y + z + w)^2
% /.  {y->1,z->2}   y に 1 を、z に 2 を代入
Remove[x,y,z,w]
```

方程式の解を `Solve[]` を用いて、求めたときの結果はこの代入をするのに便利です。例えば $x^2+x+1=0$ の根の 3 乗を計算するには、次のようにすれば OK.

```
Solve[x^2+x+1==0,x]
x^3 /.  %               解の 3 乗を計算してみる
Remove[x]
```

5.15　微分

`D[x^n,x]`	$(x^n)'$
`f=x^2+2x y+3y^2+4x-5y+6`	
`D[f,x]`	f_x
`D[f,x,y]`	f_{xy}
`D[f,{x,2}]`	f_{xx}
`D[f,{x,2},{y,3}]`	f_{xxyyy}
`Remove[f]`	
`f[x_]:= x^2+2x+3`	
`f''[x]`	
`Remove[f]`	

5.16　不定積分、定積分

`Integrate[式、変数]`, `Integrate[式、変数の範囲]` とします。数値積分版 `NIntegrate[]` もあります。後者は近似値しか計算できませんが、前者では計算できないような定積分も扱えます。

```
Integrate[1/(1+x^2),x]
Integrate[x^2,{x,0,1}]
Integrate[E^(-x^2),{x,0,Infinity}]
NIntegrate[Sqrt[Sin[x]], {x,1,2}]
NIntegrate[Sqrt[Sin[x]], {x,1,2}, WorkingPrecision->50,AccuracyGoal->40]
```

5.17　級数 (シグマ, Taylor 展開)

`Sum[n,{n,1,5}]`	$\sum_{n=1}^{5} n$
`Sum[k^2,{k,n}]`	$\sum_{k=1}^{n} k^2$
`Series[Exp[x],{x,0,10}]`	$x = 0$ のまわりの 10 次までの Taylor 展開！

5.18　方程式を解く

`Solve[]`, `NSolve[]` という二つの手続きがあります。

`Solve[左辺==右辺, 未知数]` は式変形で解きます。

```
Solve[x^2+3x+2==0, x]
Solve[{x+y+z==6, 2x-y+z==5,-3x+y+2z==0},{x,y,z}]
```

Mathematica は 2 次方程式だけでなく、3 次方程式、4 次方程式の根の公式も覚えていて解くことが出来ますが、やってみれば分かるように、結果は分かりにくくなることが多いです。どの程度の値なのか知りたい場合、つまり近似値でよければ直後に `% // N` あるいは `N[%, 50]` のように入力すれば求められます。

```
Solve[x^3+2x^2+3x+4==0, x]
% // N                             直前の結果を小数で
N[%%, 50]                          二つ前の結果を 50 桁の小数で
```

文字を係数に含む方程式も解くことが出来ます。

```
Solve[a x + b ==0, x]
Reduce[a x + b ==0, x]  ちゃんと場合わけをする
```

もっとも、特別簡単なものを除けば、方程式は式変形のみで解くことは難しいです (できないことが多い)。そういう場合は、近似値を求めることで我慢することにすれば、`NSolve[]`, `FindRoot[]` などの関数が利用できます。

```
NSolve[x^3+2x^2+3x+4==0, x, 40]       精度 40 桁で解く
FindRoot[x^3+2x^2+3x+4==0, {x, 0}]   Newton 法で 0 の近くの解を探す。
```

5.19 極限

その名もずばり `Limit[]` という関数があります。その際 ∞ を意味する “`Infinity`” が使えることに注目。

```
Limit[Sin[x] / x, x-> 0]
Limit[(x^2 + 2 x + 3)/(3 x^2 + 2 x + 1), x->Infinity]
```

いわゆる片側極限 $\lim_{x\uparrow a} f(x)$, $\lim_{x\downarrow a} f(x)$ も計算できます。

```
Limit[Tan[x], x-> Pi/2, Direction -> 1]    下から (左から) の極限
Limit[Tan[x], x-> Pi/2, Direction -> -1]   上から (右から) の極限
```

5.20 リストとその応用 (行列、ベクトル)

{ と } の中にカンマで区切って複数のものを並べて作った「リスト」というデータ構造が

あります。

`list = {1,2,3}`　1, 2, 3 という 3 つの要素からなるリスト `list` を定義

- 関数の多くはリストに対して作用します。

```
Log[{a,b,c}]
N[{1/2,1/3,1/4}]
```

- 結果をリストとして返す関数は多いので (方程式の解が複数個ある場合とか、固有値、固有ベクトルを求める関数など)、リストの中から特定の要素を取り出す `Part[番号]` や `[[番号]]` は重要です。

```
list = {1,2,3}
Part[list,2] または list[[2]]                    list の第 2 要素
Eigenvalues[{{1,2},{3,4}}]                       行列の固有値を計算
lambda1 = %[[1]]; lambda2 = %[[2]]               それぞれ変数に代入
{lambda1,lambda2}=Eigenvalues[{{1,2},{3,4}}]     同時に代入
```

- リストは集合、ベクトル、行列として扱えます。
- 集合的な演算をするための関数として `Union[]`, `Intersection[]`, `Complement[]` などがあります。
- ベクトル的な利用法として

```
{1,2,3}+{a,b,c}
2 {1 3 5}
{1,3,5} / 3
{1,2,3} .  {3,4,5}   内積
{1,2,3} {2,3,4}      成分ごとの積
{1,2,3} / {2,3,4}    成分ごとの商
```

- 行列演算もできます。

```
A={{a11,a12,a13},{a21,a22,a23},{a31,a32,a33}}
y={y1,y2,y3}
A . y                                            行列とベクトルの積
                                                 (ドットが必要なことに注意)
```

行列用の関数としては、行列式 `Det[]`, 転置行列 `Transpose[]`, 逆行列 `Inverse[]`, 固有値 `Eigenvalues[]`, 固有ベクトル `Eigenvectors[]` などがあります。

- Lisp 言語風の関数として `First[]`, `Rest[]` などがあります。
- その他 `Length[{a,b,c,d,e}]`, `MemberQ[{a,b,c,d,e,f},a]`, `Count[{a,b,a,b,a,b},a]`, `Reverse[]`, `Sort[]`, `RotateLeft[]`, `RotateRight[]` 等々。
- リストを生成する `Table[]` という関数も慣れると便利です。

```
Table[i^2, {i,6}]
Table[Sin[n Pi/5], {n,0,4}]
Table[x^i+2i, {i,5}]
Table[Sqrt[x], {x, 0, 1, 0.25}]
Table[x^i+y^j, {i,3}, {j,2}]
Table[PrimeQ[2^2^p+1],p,8]
Table[{2^2^n+1,PrimeQ[2^2^n+1]},{n,6}]
```

- `ReadList[]` という関数を使って外部ファイルからリストに読み込める。

```
ReadList["ファイル名", Number]
ReadList["ファイル名", Number, RecordList]
ReadList["外部コマンド名", Number]
ReadList["!外部コマンド名", Number, RecordList]
```

“`square.data`” を

```
1 1
2 4
3 9
4 16
```

という内容のファイルとする時、以下のコマンドで何が起こるか？

```
ReadList["square.data", Number]
ReadList["square.data", Number, RecordLists -> True]
```

6 Mathematica のグラフィックス

6.1 基本的なこと

グラフィックスは結構メモリーを消費するので、人数が多い場合は不要なウィンドウは早目に消すなどを心掛ける。グラフィックスのウィンドウを消すには、

“File” というメニュー項目がある場合 File メニューから “Quit” を選択する。

“File” というメニュー項目がない場合 単にウィンドウにマウス・カーソルを持って行って、マウス・ボタンをクリック。

(複数のバージョンの Mathematica が混在していて、扱い方が少し異なる。)

6.2 グラフィックス用の代表的な関数

まず、どんな関数が用意されているかざっと見てみよう。

2 次元グラフ `Plot[]` (関数のグラフ), `ParametricPlot[]` (パラメーター曲線の描画), `ListPlot[]` (リスト・データのプロット)

3 次元グラフ `Plot3D[]` (2 変数関数のグラフ), `ParametricPlot3D[]` (パラメーター曲線・曲面の描画), `ListPlot3D[]`

等高線グラフ `ContourPlot[]` (2 変数関数の等高線), `ListContourPlot[]`

濃淡グラフ `DensityPlot[]` (2 変数関数の濃淡図), `ListDensityPlot[]`

6.3 グラフィックス・オブジェクト

グラフィックスの関数では、対応するグラフィックス・オブジェクトを生成して、画面に表示する。グラフィックス・オブジェクトは変数に代入可能である。オブジェクトを表示するには関数 `Show[]` を用いる。またオブジェクトがどう表現されているかは `InputForm[]` で読むことが出来る。

次の例では、変数 g1, g2 に代入しておいて、後から再表示している。

```
g1 = Plot[Sin[x],{x,0,2Pi}]
g2 = Plot[Cos[x],{x,0,2Pi}]
Show[g1]
Show[g1,g2]
InputForm[g1]
```

6.4 グラフィックスの関数の引数、特にオプション

グラフィックスの関数が、引数として取るものは、順に

(1) 関数あるいは関数のリスト or 数値データによるリスト

(2) 描画する範囲

(3) 描画を制御するオプション

このうちオプションはその名前の通り省略可能である。

(オプションとは？実際にどうやって描画するかについて、自由度があり、それを一々指定するのはとても面倒なので、普段はそれを適当に決めているが、そのデフォールト値で満足できない場合に、ユーザーが自分の求めるものを指定することが出来るようにしたもの。)

関数がどういうオプションを持つかは “??関数名” で調べられる。例えば、関数 `Plot[]` のオプションを調べたければ、次のようにすればよい。

```
??Plot
```

オプションの指定の仕方は、” オプション名 `->` 値” である。例えば

`AspectRatio ->` 数値	デフォールト値は `1/GoldenRatio` `Automatic` とすると `1:1`
`Axes ->` 真偽値	
`AxesLabel -> {"x", "y=f(x)"}`	
`Compiled -> False`	デフォールト値は `True`
`Frame -> True`	デフォールト値は `False`
`GridLines -> Automatic`	
`PlotRange -> {zmin,zmax}`	
`PlotPoints->100`	

しばらくの間、オプションを変更したまま使いたい場合は、`SetOptions` を使う。次の例では、関数 `Plot3D` 使用時に `PlotPoints` を `100` にするよう設定している。

```
SetOptions[Plot3D, PlotPoints->100]
```

6.5 2次元グラフィックス

6.5.1 Plot[]

1 変数関数のグラフを描くには `Plot[]` を使う。基本的な使い方は

```
Plot[関数, 描画範囲を示すリスト]
Plot[{関数 1, 関数 2, . . . , 関数 n}, 描画範囲を示すリスト]
```

描画範囲を示すリストから、関数値を計算するための変数の適当な値を選びだして、そこでの関数値を評価して、そうして得た点 (標本点) を順に結んでグラフを描く。標本点の個数を多くするには PlotPoints オプションを指定する。例として

```
Plot[Sin[x], {x,0,2Pi}]
Plot[Sin[x], Sin[2x], Sin[3x], {x,0,2Pi}]
Plot[Sin[50x],{x,0,Pi},PlotPoints->1000]
```

6.5.2 ParametricPlot[]

いわゆるパラメーター表示された平面曲線を描く。例えば

```
ParametricPlot[{Sin[t], Sin[2t]}, {t,0,2Pi}]
```

6.5.3 ListPlot[]

数値データのリストから曲線を描く。これを用いると、例えば外部のプログラムで計算したデータを表示できる。次の例では `Table[]` で生成したリストから曲線を描いている。

```
fp = Table[{t,N[Sin[Pi t]]}, {t,0,0.5,0.025}]
ListPlot[fp]
ListPlot[fp, PlotJoined -> True]
```

6.5.4 NDSolve[]

```
NDSolve[{y'[x]==Sin[y[x]],y[0]==1}, y, {x,0,4}]
Plot[Evaluate[y[x] /.  %], {x,0,4}]
```

(この `NDSolve[]` は数値的に微分方程式を解くコマンドであり、結果は離散的な点での関数値を近似的に求めたものである。計算していない点での値は補間により簡略計算する。こういうものを InterpolatingFunction という。)

```
y[1.5] /. %%
```

6.5.5 2 変数関数の等高線、濃淡図

2 変数関数 $(x, y) \mapsto f(x, y)$ を可視化するために、等高線を描く `ContourPlot[]`[2], 濃淡図を描く `DensityPlot[]` が用意されている。 使い方は

```
ContourPlot[f, {x,xmin,xmax}, {y,ymin,ymax}]
DensityPlot[f, {x,xmin,xmax}, {y,ymin,ymax}]
```

例として

```
ContourPlot[Sin[x]Sin[y], {x,-2,2}, {y,-2,2}]
```

オプションとして

```
Contours -> 整数
PlotRange -> {zmin,zmax} または Automatic
PlotPoints -> 整数 （デフォールトが 15。小さい！非力さが出て来る。）
ContourShading -> False （デフォールトは True）
```

2 変数関数についての方程式で定義される陰関数のグラフを描くために、`ImplicitPlot[]` という関数があるが (ただし古いバージョンの Mathematica にはない)、それを使うには事前に `Needs["Graphics‘ImplicitPlot‘"]` を実行する必要がある。

---

[2] 等高線のことを contour (line) と呼ぶ。

```
Needs["Graphics‘ImplicitPlot‘"]
ImplicitPlot[x^2+y^2==1, {x,-1,1}]
```

6.6　3次元グラフィックス

2 変数関数のグラフを描く `Plot3D[]` は

```
Plot3D[f, {x,xmin,xmax}, {y,ymin,ymax}]
```

が基本的な使い方。例えば

```
Plot3D[Sin[x y], {x,0,3}, {y,0,3}]
```

オプションとして以下のようなものがある。

`HiddenSurface -> False`	隠面消去をしない
`PlotPoints ->` 個数	大きくすると細かい図を書く。
`ViewPoint -> {x,y,z}`	視点の指定

`ParametricPlot3D[]`, `ListPlot3D[]` 等もある。次の例は是非試してみよう。

```
ParametricPlot3D[{Sin[t], Cos[t], t/3}, {t, 0, 15}]
ParametricPlot3D[{t,u,Sin[t u]}, {t,0,3}, {u,0,3}]
ParametricPlot3D[{Sin[t],Cos[t],u}, {t,0,2Pi}, {u,0,4}]
ParametricPlot3D[{Cos[t](3+Cos[u]),Sin[t](3+Cos[u]),Sin[u]},
        {t,0,2Pi}, {u,0,2Pi}]
ParametricPlot3D[{Cos[t]Cos[u],Sin[t]Cos[u],Sin[u]},
        {t,0,2Pi}, {u,-Pi/2,Pi/2}]
```

6.7　グラフィックスの印刷のしかた

関数 `Export[]` を “`Export["`ファイル名 `"]`, グラフィックス`]`” のように使って、グラフィックスをファイルに出力できる。ファイル名の末尾を `.eps` にしておくと、自動的に EPS (Encapsulated PostScript) フォーマットが選択されて便利である。

トーラスの図をファイルに記録

```
g = ParametricPlot3D[{Cos[t](3+Cos[u]),Sin[t](3+Cos[u]),Sin[u]},
            {t,0,2Pi}, {u,0,2Pi}]
Export["torus.eps", g]
```

とすると “`torus.eps`” というファイルが出来るが、これを印刷するには

torus.ps を印刷

```
waltz21% lp -d プリンタ名 torus.eps
```

のようにしてプリンターに送ればよい。

こうして作成した EPS ファイルは LaTeX 文書に取り込むのも簡単である。

6.7.1　昔はこうだった

`Display[]` 関数を “ `Display["!psfix > ファイル名", グラフィックス]`” のように使うと図形を印刷するための PostScript データが作成できる。

例えば

トーラスの図をファイルに記録

```
g = ParametricPlot3D[{Cos[t](3+Cos[u]),Sin[t](3+Cos[u]),Sin[u]}, {t,0,2Pi}, {u,0,2Pi}]
Display["!psfix > torus.ps", g]
```

7　Mathematica のサウンド機能

(準備中)

8　Mathematica のプログラミング機能

Mathematica は、これまでに述べてきたような電卓的な使用法、つまり一つ一つの計算の指示を人間が手で入力する仕方でも十分役立つが、通常のプログラミング言語にひけを取らないプログラミングの機能を持っている。これまでに説明していないもので、プログラミングに必須 or 役立つものをあげてみよう。

- 複数の文を並べること
- 条件判断 (真偽の判定)
- 条件判断に基づく分岐
- 繰り返し処理用の専用メカニズム
- まとまった処理に名前をつけて一つの単位とする

8.1　基本的なプログラミング機能、特に制御構造

8.1.1　複数の文を並べる

(1) `";"` で区切って並べることができる。

(2) “(”, “)” で括ることもできる。

8.1.2 条件判断 (論理式)

関係演算子、論理演算子は C 言語のそれに良く似ている:

`a == b`	等しいか？
`a != b`	等しくないか？
`a < b`	a は b より小さいか？
`a > b`	a は b より大きいか？
`a <= b`	a は b より小さいか、等しい？
`a >= b`	a は b より大きいか？
`&&`	「かつ」.
`||`	「または」.
`!`	「否定」.

論理式の値は、`False` = 偽, `True`= 真, である。次の例を試してみること。

```
1+1 == 2
2^3 == 7
1 < 2 < 3
2 != 3
LogicalExpand[(p || q) && !(r || s)]
```

8.1.3 条件判断による分岐

`If[`*test*, *then-statement*, *else-statement*`]` とすると分岐が実現できる。例えば

```
If [1+1==2, Print["Yes, you are right."], Print["No, you are wrong."]]
```

8.1.4 繰り返し構文

計算機のプログラムで広い意味の繰り返しは重要である。それを実現するために、再帰的関数 (手続き) 定義や、繰り返しの構文が用意されている。

Do 関数 使い方は “`Do[`*statement*, *iterator*`]`”. Fortran の do 文、C の for 文、BASIC の FOR NEXT 構文、Pascal の for 文に似ている。*iterator* (繰り返し指定) で指定しただけ *statement* (文) を繰り返して実行する。

```
Do[Print[i!], {i,5}]                i=1,2,3,4,5
Do[Print[2^i], {i,0,5}]             i=0,1,2,3,4,5
Do[Print[I^i], {i,0,10,3}]          i=0,3,6,9
r=1; Do[r=1/(1+r), {100}]; r        100 回
Do[Print[i], {i,2a,4a,a}]           i=2a,3a,4a
Do[Print[r], {r,0.0,3.5}]           r=0.0,1.0,2.0,3.0
Do[Print[{i,j}], {i,3}, {j,i}]      do i=1,3
                                      do j=1,i
                                        Print {i,j}
                                      end do
                                    end do
```

While 関数 使い方は “`While[`*test*`,` *statement*`]`”. C や Pascal の while 文に似ている。*test* が真である間だけ *statement* を繰り返す。

```
i=1; While[i <= 10, Print[i]; i++]
```

この例では “`i <= 10`” が *test*, “`Print[i]; i++`” が *statement* になっている。

For 関数 使い方は “`For[`*statement1*`,` *test*`,` *statement2*`,` *statement*`]`”. C の for 文に似ている。

```
For[i=1, i <=10, i++, Print[i, " ", i^2]]
```

繰り返しの指定 Do 文で使われている iterator は、あちこちで使われる。和 $\sum$ を計算する `Sum` や乗積 Π を計算する `Product`、表を作る `Table` など。

`Sum[i, {i,1,10}]`	$i = 1, 2, \cdots, 10$
`Product[x-i, {i,0,5}]`	$i = 0, 1, \cdots, 5$
`Product[e, {e,x,x-5,-1}]`	$e = x, x-1, x-2, x-3, x-4, x-5$
`Table[i!, {i,5}]`	$i = 1, 2, 3, 4, 5$

`Print[]` の結果は表示されるだけで、後には残らないので、残して解析したい場合は `Table[]` などを使うのがお勧めである (このあたりは C 言語のプログラミングの発想とは異なる)。

8.2 関数定義

Mathematica のプログラムは (ユーザー定義の) 関数の集合という形になる。
百聞は一見にしかず。$f(x) = x^2$ なる関数 f は、

```
f[x_] := x^2
```

で定義出来る。“関数名 [引数名_] := 式” という形式で定義する。(また “`f[名前_ 型名] :=` 式” のように型名を指定することも出来る。) 関数を再定義すると古い定義は消えてしまう。

```
f[4]
f[a+1]
f[3x+x^2]
```

もちろん関数 “`f`” の定義が見たい場合は、“`?f`”, または “`??f`” とする。

多変数の関数も定義できる。

```
f[x_,y_] := (x^2 - y^2) / (x^2 + y^2)
```

変数の特定の値に対する関数値を代入文で定義することも出来る。次の例では Fibonacci 数列を計算する関数 `f[]` を定義している。再帰的な定義が出来ることに注意しよう。

```
f[x_] := f[x] = f[x-1]+f[x-2] ; f[1]=1; f[2]=1
Table[f[n],{n,10}]
```

他の例として

```
f[x_] := Sin[x]/x; f[0]=1
Plot[f[x],{x,-2Pi,2Pi}]
```

8.3 関数定義のための細かい注意

以下の記述はプログラミングの中級者向けであって、最初は読み飛ばして構わない。実用的なプログラムを書く際に必要な、他への悪影響が出ない＆他からの影響を受けないようにするための工夫の話。

局所変数の利用 特別なことをしないかぎり、変数は大域的なものとなるので (要するに、どこからでも見える)、名前の衝突[3]に気を付ける必要がある。例えば

```
PowerSum[x_, n_] := Sum[x^i, {i,1,n}]
```

とすると、“`PowerSum[x,5]`” のようなのは大丈夫だが、“`PowerSum[i,5]`” はダメになる。“`Module[`{*local-var1,local-var2,..*}`,` *procedure*`]`” を利用して

```
PowerSum[x_,n_] :=
  Module[{i},             ← 局所変数 i を使うことを宣言
    Sum[x^i, {i,1,n}]
  ]
```

とする方が良い (`Module[]` に似たものに `Block[]` という関数がある。前者は変数そのものを局所的に作るが、後者は変数の値だけを局所的に作る。)。

context の利用 実は、上の例はまだ完璧とは言えない。“`i`” という名前そのものはグローバルに見えてしまう (あまり実害はないけれど)。context (文脈と訳されることが多い) を導入して、名前の扱いを制御する。

---

[3] 異なるものに同じ名前をつけようとすると、しかられたり、古い方が上書きされて、動作が変になったりする。

```
Begin["Private‘"]
  PowerSum[x_,n_] :=
    Module[{i},
      Sum[x^i, {i,1,n}]
    ]
End[]
```

この例では “Private‘” という context を導入している。とはいえ、他人に使ってもらうプログラムを書く場合でもなければ、ここまで気にする必要はないかもしれない。

例 1.

先に出した Fibonacci 数列を計算する関数はあまり効率が良くないので、`Do[]` を用いて書いてみる:

```
Fibonacci[n_Integer?Positive] :=
  Module[{fn=1,fn2=0},
    Do[{fn1,fn2}={fn1+fn2,fn1}, {n-1}];
    fn1
  ]
```

例 2 .

平均値、分散を計算するプログラム (関数) を作れ。

```
mean::usage =
  "mean[list] returns the mean value of the elements of list."
variation::usage =
  "variation[list] returns the variation of the elements of list."

Begin["Private‘"]
mean[l_List] :=
  Module[{n = Length[l], i}, Sum[l[[i]],{i,n}] / n]
variation[l_List] :=
  Module[{n, m, i},
        n = Length[l]; m=mean[l]; Sum[(l[[i]]-m)^2, {i,n}]/n]
End[]
```

Mathematica では C のように何かをするための専用プログラムを記述して、コンパイルして実行するのではなく、知識を増やして賢くして行き、質問に答えさせるという感じ。何か複雑なことを計算させるときは、○ ○ 関係の関数集 (パッケージという) をシステムにロードして使う、という形態を取る。こういう場合、名前をきちんと扱うのは重要になる。(人間の知的活動でも、一つの名前が、状況によって全く違う意味になることがあるが、文脈の違いを適当に認識することによって使い分けている。)

9　ある年度の情報処理 II レポート問題

今回は問題を自作してもよい。自分の計算力だけでは解けない or 解くのに時間がかかる or 正確に解くのが難しい問題を設定して Mathematica を利用して解決してみる。テーマは Mathematica は数学の勉強 (研究) にどの程度役立てられるか探ることにある。レポートは、その過程で考えたことをまとめる。

問題 1

高校での数学の計算問題、大学の微分積分や線形代数の計算問題を解かせてみる。どこまで解けるか、どういう問題が解けないか (ごくまれに結果がおかしいこともある)、色々試し、また解けないのは何故か推測する。(問題によっては工夫が必要かもしれない。簡単に Mathematica では解けないと決めつけないで、考えること。色々考えた経過をレポートする。)

問題 2

1 変数、または 2 変数の関数 f を定め、グラフ、等高線、微分が 0 になる点、またその点での関数値等を計算する。多変数関数の極値問題を解く際にどのように利用できるか、試して見よ。(この方向に発展させると、極値問題を解くプログラムが作れるかも知れない? これまで完全なプログラムを書いた学生はいない。挑戦を期待する。

10　工事計画

- ノートブックの使い方
- 画像の印刷など
- 音声の取り扱い
- パレット
- レポートの作り方

A　式の評価順序の話 — Evaluate[]

```
Table[BesselJ[n,x],{n,5}]
```

とすると、`{BesselJ[1,x],BesselJ[2,x],BesselJ[3,x],BesselJ[4,x],BesselJ[5,x]}` という 5 つの関数を含んだリストができる。一方、

```
Plot[{BesselJ[1,x],BesselJ[2,x],BesselJ[3,x],BesselJ[4,x],
BesselJ[5,x]},{x,0.0,10.0}]
```

とすると、その 5 つの関数のグラフが表示される。すると、

```
Plot[Table[BesselJ[n,x],n,5],{x,0.0,10.0}]
```

とすれば 5 つの関数のグラフが描けるような気がするが、実はうまく行かない。Lisper だったら、その理由は分かるでしょう。そして、

```
Plot[Evaluate[Table[BesselJ[n,x],{n,5}]],{x,0,10}]
```

とすると、うまく行くというのにも納得できるでしょう。一件落着。

同様に

```
f[x_]:=Sin[x]
```

としたとき、`D[f[x],x]` は `Cos[x]` となるが、

```
Plot[D[f[x],x],{x,0,2Pi}]
```

としても `Cos[x]` のグラフは描けない。

```
Plot[Evaluate[D[f[x],x]],{x,0,2Pi}]
```

あるいは、

```
Plot[f'[x],{x,0,2Pi}]
```

とする。

B　プログラム例

B.1　2次元の Newton 法

```
f[x_,y_]:={x^2-y^2+x+1,2 x y +y}
Df[a_,b_]:=Module[
           {x,y},
           Transpose[{D[f[x,y],x],D[f[x,y],y]}] /. {x->a,y->b}
          ]
{x,y}={1,1}
Do[{x,y}={x,y}-Inverse[Df[x,y]].f[x,y];
   Print[{x,y},"=",N[{p,q},20]],
   {6}
  ]
```

あるいは、変数の方もベクトル的にリストを使って書いて、

```
f[{x_,y_}]:={x^2-y^2+x+1,2 x y +y}
Df[{a_,b_}]:=Module[
           {x,y},
           Transpose[{D[f[{x,y}],x],D[f[{x,y}],y]}] /. {x->a,y->b}
          ]
xk={1,1}
Do[xk=xk-Inverse[Df[xk]].f[xk]; Print[xk,"=",N[xk,20]], {6}]
```

B.2 二変数関数の極値問題 `kyokuchi.m`

```
(* 二変数関数 f の停留点を求める（よう努力する）*)
teiryuuten[f_]:=
Module[
  {fx,fy},
  fx=Simplify[D[f[x,y],x]];
  fy=Simplify[D[f[x,y],y]];
  Solve[{fx==0,fy==0},{x,y}]
]

(* 二変数関数 f とその停留点のリスト s を分析し、極値の判定をする *)
bunseki[s_,f_]:=
Module[
  {ff,HesseXY,aSolution,restSolutions,valf,l1,l2},
  ff=f[x,y];
  HesseXY = {{D[ff,x,x],D[ff,x,y]},
             {D[ff,y,x],D[ff,y,y]}};
  restSolutions = s;
  While [(restSolutions != {}),
         aSolution = First[restSolutions];
         restSolutions = Rest[restSolutions];
         valf = ff /. aSolution;
         {l1,l2} = Eigenvalues[HesseXY /. aSolution];
         If [l1 > 0 && l2 > 0,
            Print[aSolution, ", 極小 f(x,y)=", valf]];
         If [l1 < 0 && l2 < 0,
            Print[aSolution, ", 極大 f(x,y)=", valf]];
         If [(l1 l2 < 0),
            Print[aSolution, ", 極値でない"]];
         If [(l1 l2 == 0),
            Print[aSolution, ", 極値であるかどうか分からない。"]];
      ]
]
```

使用例

```
oyabun% math
Mathematica 4.0 for Solaris
Copyright 1988-1999 Wolfram Research, Inc.
 -- Motif graphics initialized --

In[1]:= << /home/syori2/kyokuchi.m

In[2]:= f[x_,y_]:=x y(x^2+y^2-4)

In[3]:= s=teiryuuten[f]

Out[3]= {{x -> -2, y -> 0}, {x -> -1, y -> -1}, {x -> -1, y -> 1},

>    {x -> 0, y -> 0}, {x -> 1, y -> -1}, {x -> 1, y -> 1}, {x -> 2, y -> 0},

>    {y -> -2, x -> 0}, {y -> 2, x -> 0}}

In[4]:= bunseki[s,f]
{x -> -2, y -> 0}, 極値でない
{x -> -1, y -> -1}, 極小 f(x,y)=-2
{x -> -1, y -> 1}, 極大 f(x,y)=2
{x -> 0, y -> 0}, 極値でない
{x -> 1, y -> -1}, 極大 f(x,y)=2
{x -> 1, y -> 1}, 極小 f(x,y)=-2
{x -> 2, y -> 0}, 極値でない
{y -> -2, x -> 0}, 極値でない
{y -> 2, x -> 0}, 極値でない

In[5]:=
```