情報処理 II No.10

Mathematica 入門 (2)

桂田 祐史

1995 年 6 月 30 日

1 Mathematica 電卓的使用法 — 前回の範囲 — の補足

定義の確認

- “?Global‘*” とすると現在何を定義しているか見える。
- 関数 “f” の定義が見たい場合は、 “?f”, または “??f”.
- 関数をクリアしたい場合は、 “Clear[f]” で定義の内容をクリア、 “Remove[f]” で名前まで込めてクリアする。

計算をしていて不要になったものは、早めに Clear[] あるいは Remove[] しておいた方が無難です。

結果の簡単化 計算結果が正しいけれども、自分の望んでいる形に表されないことがよくあります。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、取り敢えず “Simplify[%]” として「直前の結果の簡単化」を試みましょう（何も変わらないことも多いですけど）。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]
```

変数の代入 (置き換え) 式の値を変数に代入するには単に、 “変数 = 式” とすれば OK です。

```
y = (x+1)^10
y
x = 1
y
x = 2
y
Remove[x,y]
```

上のように y そのものを変化させるのでなく、式の値を変数に一時的に代入したい場合は、 “/. 名前 -> 値 ” とします。具体的には、例えば

```
y=x^2
y /. x-> 1
```

一度に複数の代入をするには “/. {名前 -> 値, 名前 -> 値,...}” です。

```
(x + y + z + w)^2
% /. {y->1,z->1}
```

ここで “%” という記号を使って、直前の計算結果を引用しています。

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、結果は分かりにくいものですね。どの程度の値なのか、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 OK です。

```
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します
```

もっとも、特別簡単なものを除けば、方程式は式変形で解くことは難しく、そういう場合は、近似値を求めることで我慢することにして[1]、 NSolve[] が利用できます。

```
NSolve[x^3+2x^2+3x+4==0, x, 40]
```

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 $\lim_{x\to a-0} f(x)$, $\lim_{x\to a+0} f(x)$ も計算出来ます。

```
Limit[Tan[x], x-> Pi/2, Direction -> 1]     左からの極限
Limit[Tan[x], x-> Pi/2, Direction -> -1]    右からの極限
```

[1] 他に “FindRoot” などという関数もあります。

リストの扱い (続き) リストを生成する Table[] にも慣れましょう。

```
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}]
```

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とか。)、リストの中から特定の要素を取り出す "[[番号]]" は覚えておくと便利です。

`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

2 Mathematica のプログラミング機能

Mathematica は、すでに述べ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ちますが、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ます。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ましょう。

- 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

2.1 複数の文を並べる

1. ";" で区切って並べることができます。
2. "(", ")" で括ることもできます。

2.2 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？ Fortran の ".eq." (equal to) に相当します。
`a != b`	等しくないか？ Fortran の ".ne." (not equal to) に相当します。
`a<b`	a は b より小さいか？ Fortran の ".lt." (less than) に相当します。
`a > b`	a は b より大きいか？ Fortran の ".gt." (greater than) に相当します。
`a <= b`	a は b より小さいか、等しい？ Fortran の ".le." (less than or equal to) に相当し
`a >= b`	a は b より大きいか？ Fortran の ".ge." (greater than or equal to) に相当します
&&	「かつ」. Fortran の ".and." に相当します。
||	「または」. Fortran の ".or." に相当します。
!	「否定」. Fortran の ".not." に相当します。

論理式の値は、 False = 偽, True= 真, である。

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

2.3 条件判断による分岐

例えば If[*test*, *then-statement*, *else-statement*]

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

2.4 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す。それを実現するために、再帰的関数 (手続き) 定義や、繰り返しの構文があります。

Do 関数 使い方は "Do[*statement*, *iterator*]". Fortran の do 文、 BASIC の FOR NEXT 構文、 Pascal の for 文に似ています。 *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}]
Do[Print[2^i], {i,0,5}]
Do[Print[I^i], {i,0,10,3}]
r=1; Do[r=1/(1+r), {100}]; r
Do[Print[i], {i,2a,4a,a}]
Do[Print[r], {r,0.0,3.5}]
Do[Print[{i,j}], {i,3}, {j,i}]
```

```
i=1,2,3,4,5
i=0,1,2,3,4,5
i=0,3,6,9
100 回
i=2a,3a,4a
r=0.0,1.0,2.0,3.0
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 文に似ています。 *test* が真である間だけ *statement* を繰り返します。

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

この例では "i <= 10" が *test*, "Print[i]; i++" が *statement* です。

For 関数 使い方は "For[*statement1*, |it test, *statement2*, *statement*]". C の for 文に似ています。

```
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、あちこちで使われます。

```
Sum[i, {i,1,10}]
Product[x-i, {i,0,5}]
Product[e, {e,x,x-5,-1}]
Table[i!, {i,5}]
```

2.5 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ります。
百聞は一見にしかず。 $f(x) = x^2$ なる関数 f は、

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

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

```
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" です。
多変数の関数も定義できます。

```
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ます。次の例では Fibonacci 数列を計算する関数 f[] を定義しています。

```
f[x_] := f[x] = f[x-1]+f[x-2] ; f[1]=1; f[2]=1
```

他の例として

```
f[x_] := Sin[x]/x; f[0]=1
```

2.5.1 関数定義のための注意事項

以下の記述はプログラミングの中級者向けです。最初は飛ばしても構いません。実用的なプログラムを書く際には、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が必要です。

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[2]に気を付ける必要があります。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ですが、 "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}]
  ]
```

とする方が良いでしょう (Module[] に似たものに Block[] という関数があります。)。

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。

[2]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します。

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

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ません。

例. 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[]
```

例： 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[]` を用いて書いてみましょう:

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

3 その他の関数

3.1 ファイル入出力

!! ファイル名	ファイルの内容を表示
<< ファイル名	ファイルの内容を入力
式 >> ファイル名	式の内容をファイルに出力
式 >>> ファイル名	式の内容をファイルに追加出力
! 外部コマンド	外部コマンドを実行
<< "! 外部コマンド "	外部コマンドの出力を入力
式 >> "! 外部コマンド "	式の内容をコマンドに与える

```
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]

Save["ファイル名 ", 関数名 1, 関数名 2, ...]
```

関数の定義をファイルに出力
後で "<< ファイル名 " で入力可

Dump["ファイル名 "]　その時点での Mathematica のすべての状態を出力
math -x ファイル名、で再開

“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]
```

3.2　図形の印刷のしかた

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データが作成できます。

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

こうすると “mygraph.ps” というファイルが出来ますが、これを

lpr -P プリンタ名 mygraph.ps

のようにしてプリンターに送れば、印刷することが出来ます。**注意**: lpr を数学科のワークステーションで実行すると、数学科計算機室のプリンターに出力されます。センターのプリンターでの印刷の仕方は次週説明しますので、それまでは我慢して下さい。

4　レポート問題について

問題は次回出しますが、自作しても OK です。

- 微分積分や線形代数のテキストの計算問題を解き、うまく解けるか or 解けないか or 解答のミスが見つかるか、試してみる。問題によっては工夫が必要かも知れない。うまく解けない場合は何故だか分析してみる。
- これまで自分が他の言語で作ったことのあるプログラムを Mathematica で書き直してみる。(Fortran で書いてあるプログラムを Mathematica に書き直すのは、あまり自然なことではないですが、、、)
- 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。(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....?)