物理で使う数値計算入門：Julia言語による簡単数値計算

令和元年5月1日
永井佑紀

1. はじめに：このノートの目的

初めて数値計算をする人がJuliaを使って簡単にコードを書いて計算できるように、という意図で書いています。特に、物理で使うことの多い計算を中心にまとめています。

1.1 Juliaの利点

Juliaは2018年にバージョン1となったばかりの非常に新しいプログラミング言語ですので、様々なプログラミング言語の良いところを取り入れており、非常に書きやすくかつ高速です。また、FortranやCと違い、コンパイルが必要ありません。ですので、Pythonのようなスクリプト言語のように使うことができます。Pythonは現在非常に人気のある言語ですので、書籍も豊富でWebでの文献も多く、最初のプログラミング言語として勧められることも多いと思います。しかし、Pythonを数値計算で使う場合、特別な処置をしないと非常に遅いという問題があります。
そのため、Pythonで数値計算をすると遅いために、FortranやCを数値計算のための言語として勧められることも多いと思います。
Juliaは、「Pythonのように書きやすく」「FortranやCど同程度に速い」言語となるように設計されているために、数値計算を学ぶ際の最適な言語の一つになっています。

つまり、Juliaは

- FortranやCと同程度に高速
- Pythonと同じくらい書きやすく
- 数式を扱うようにコードを書くことができる

という言語です。

1.2 具体的な利点

コードを見ればわかりますが、Julia の場合、Fortran や C で必要であったたくさんの「おまじない」や Lapackのインストールや呼び出しなどの煩雑なことを一切する必要がありません。Pythonでも似たような形でシンプルにコードが書けますが、Pythonは For ループが遅いという数値計算として使う

には問題となる点(工夫すれば速くなりますが Python ならではのこの工夫を習得するのに時間がかかります）があります。この問題点のせいで、教科書に書いてあるようなアルゴリズムを Python にそのまま移植するととんでもなく遅くなってしまうことがあります。Python はあらかじめわかっているアルゴリズムを呼び出すことにかけてはそのライブラリの豊富さとコミュニティの広さで圧倒的ですが、新しいアルゴリズムを書いたりする場合には、最適ではないと思います。その点、Julia はアルゴリズムをそのままコードにするだけで速いです。これは、「物理以外の余計なことを考えずに物理の結果が知りたい」という、物理をやる人間にとって重要な欲求を満たす可能性のある言語となっている、ということですので、有望だと思います。

2. Juliaのインストール

2.0. 二種類の実行方法

Juliaを使うには、二つの方法があります。

1. 対話的実行環境　REPL(read-eval-print loop)
2. 通常の実行方法

1は、普通のアプリケーションのようにJuliaを起動して、その中でコードを書いたり計算をしたりプロットしたりするものです。簡単な計算を気軽に試すことができます。

2は、通常の実行方法で、ファイルにプログラムコードを `test.jl` みたいな形で保存してから、

```
julia test.jl
```

で実行する方法です。

このノートでは、最初は簡単なので1.のREPLを使います。少し複雑になってきた場合には、ファイルにコードを保存して、2.で実行することにします。

2.1. バージョン

このノートでのJuliaのバージョンは、1.1.0とします。

2.2. Macの場合

https://julialang.org/downloads/
から `macOS 10.8+ Package (.dmg)` をダウンロードして、他のアプリケーションと同様にインストールします。インストールしたあとは、アプリケーションにJulia 1.1がありますので、それをダブルクリックするとターミナルが起動して使えるようになります。

2.3. Linuxの場合

Linuxの場合には、

```
wget https://julialang-s3.julialang.org/bin/linux/x64/1.1/julia-1.1.0-linux-x86_64.tar.gz
tar -xvf julia-1.1.0-linux-x86_64.tar.gz
echo 'export PATH="$PATH:$HOME/julia-1.1.0/bin"' >> ~/.bashrc
source ~/.bashrc
```

でインストールができますので、あとは

```
julia
```

で起動することができます。

2.4 Windowsの場合

Windows版のJuliaをインストールすれば使用できます。あるいは、Windows Subsystem for Linuxを使ってUbuntuを入れることで上のLinuxと同じようにインストールすることもできます。

3. 基本編

この章では、REPLの上でJuliaを使ってみましょう。

3.1 いじってみよう

まずはじめに、プログラミング言語での定番、Hello worldですが、これは、

```
println("Hello World!")
```

で出力されます。
ここで、 `println` は最後に改行あり、 `print` は改行なしです。
関数の説明がみたい時には、?を押すとHelpモードに入りますので、そこで関数名を入れます。

足し算、引き算

足し算は、

```
1+2
```

でできますし、引き算は

```
2-5
```

でできます。

かけ算と割り算

かけ算は、 * の記号で、

```
2*3
```

となりますし、割り算の記号は / で、

```
4/2
```

でできます。
割り算については注意があります。上の計算を行うと、

```
2.0
```

となりました。2と2.0の違いは、整数と実数の違いです。
もちろん、計算機では無限の桁の実数を扱うことはできませんから、この実数は桁があります。それについては後述します。
もし、割った結果を整数で欲しい場合には、

```
div(4,2)
```

とすると、答えとして 2 が返ってきます。

あまり

割り算のあまりを計算することもできます。その場合には、 % を使います。

```
5%2
```

とすると、5わる2のあまりである1が出ます。

べき乗、指数関数、対数関数

べき乗は ^ でできますので、

```
3^7
```

となります。
指数関数は `exp` で、

```
exp(3)
```

でできますし、自然対数は `log` で

```
log(2)
```

となります。対数の底が2と10の時はそれぞれ `log2` と `log10` が使えて、

```
log2(4)
```

や

```
log10(100)
```

が使えます。任意の底の場合には、nを底、xを値として、 `log(n,x)` で使えまして、

```
log(3,9)
```

となります。

三角関数

三角関数も普通に使うことができます。
例えば、

```
sin(0.1)+2*cos(0.3)
```

などができます。もちろん、tanhなども使えます。

円周率

円周率はデフォルトで入っています。日本語の漢字変換で「ぱい」としてπを入力するか、REPL上で `\pi` としてからタブキーを押すことでπを使うことができますので、

```
cos(3π)
```

というような形で書くことができます。
ここで、 `3π` と書きましたが、かけ算の記号 `*` を使って `3*π` と書くこともできます。後述しますが、数字と記号の積の場合には、記号 `*` を省略して書くことできます。
これにより、より数式に近い見た目になります。

虚数

虚数も複素数も簡単に扱うことができます。
虚数単位は `im` です。ですので、

```
4 + 5im
```

などと書きます。なお、円周率と同様に、 `5*im` は `5im` と記号 `*` を省略できます。
三角関数と組み合わせれば、

```
exp(im*π)
```

とすることもできます。
なお、この計算を行うと、結果は

```
-1.0 + 1.2246467991473532e-16im
```

のような形で表示されていると思います。この値は厳密には-1になるべきですが、すごく小さい虚数が入っています。これは、計算機の中の実数が本当の実数ではないことと関連していまして、ここで使われている数の桁数が16桁（倍精度実数と呼びます）であることを意味しています。
なお、より精度の高い計算をするための方法も実装されていまして、

```
exp(im*BigFloat(π))
```

と `BigFloat` を使うと、

```
-1.0 + 1.096917440979352076742130626395698021050758236508687951179005716992142688513354
```

とより精度の高い計算をすることができます。

関数

次に、自分で定義した関数を使うことを考えます。
例えば、

$$f(x) = \cos(x) + 2\sin(x^2)$$

という関数であれば、

```
f(x) = cos(x) +2*sin(x^2)
```

とそのまま関数を定義することができます。そして、例えばx=4での値などが知りたい場合には、

```
f(4)
```

とすれば出ます。

この `f(x)` をfunction、つまり関数と呼びます。上では一行で関数を定義しましたが、もう少し複雑な場合には、

```
function f(x)
    cos(x) +2*sin(x^2)
end
```

とすることができます。この形の関数については後述します。

ここまでで、電卓的な使い方を一通り見ることができました。

3.2 変数

次は、変数を紹介します。
先ほどの関数

$$f(x) = \cos(x) + 2\sin(x^2)$$

を

$$f(x) = \cos(x) + a\sin(x^2)$$

にしてみましょう。もし、この関数f(x)がaに依存しているならば、

$$f(x, a) = \cos(x) + a\sin(x^2)$$

と書くことも可能です。
このコードは、

```
f(x,a) = cos(x) +a*sin(x^2)
```

とそのままに書くことができます。ここで、あらかじめaに値を入れておけば、

```
a = 3
f(4,a)
```

などと書けます。このaを「変数」と呼びます。

変数には様々なものを入れることができます。

整数、実数、複素数

整数、実数、複素数であれば、

```
a = 3
b = 2.3
c = 4+5im
```

のようになります。これらはそれぞれ演算ができて、

```
a*b + c/a
```

は複素数が出てきます。
また、変数の記号として、アルファベット以外も使うことができます。
例えば、

```
H = 1.2
β = 2
Z = exp(-β*H)
```

でのβのようなギリシャ文字や、

```
りんご = 30
みかん = 20
りんご*2 + みかん*3
```

のような日本語も使用可能です。このように変数に様々な文字が使えるので、物理に出てくる数式をほとんどそのままコードとして書くことができます。

文字列

変数には文字を入れることができます。
例えば、

```
a = "Warrior"
b = "Magic"
c = b*a
println(c)
```

とすると、出力として、 `MagicWarrior` が出てきます。
注意点としては、文字列と文字列を合体させるときには、積の記号と同じ * を使うことです。これは、文字列の合体は非可換な積であるというコンセプトのもとに設定されているようです。なお、Pythonでは + を使います。

ベクトルと行列

物理の数値計算をするのであれば避けて通れないのは、ベクトルと行列です。ですので、変数にベクトルや行列を入れることができます。
Juliaでは、ベクトルは

```
a = [1,2,3,4]
```

と書くことができます。そして、行列は

```
B = [1 2 3 4
5 6 7 8]
```

と書くことができます。ここでBは2x4行列です。行列の要素同士はスペースで区切ります。また、

```
B = [1 2 3 4;5 6 7 8 ]
```

のように、改行の代わりに ; を使うこともできます。

2x4行列と4成分ベクトルの積は

```
B*a
```

と通常の積の記号 * でできます。
n成分ベクトルはnx1行列とみなすことができるので、

```
c = [1
2
3
4]
```

と書くこともできます。

サイズの大きい行列を定義するのにいちいち全部行列要素を書くのは大変です。ですので、まとめて扱う方法があります。
例えば、3x3の零行列は

```
B = zeros(3,3)
```

と定義することができます。もし、ほとんどがゼロの行列で一部だけ何か値がある場合には、

```
B[1,2] = 4
println(B)
```

とします。ここで、 `B[1,2]` は、1行2列目の行列要素です。
Juliaでは、行列の要素は1から数えます。3x3行列であれば、1,2,3、となります。Pythonは0から数え0,1,2ですので、違いに気をつけてください。

さて、零行列Bを定義した後に、その一部の行列要素が複素数である場合を考えます。その場合、

```
B = zeros(3,3)
B[1,2] = 4 + 2im
```

はエラーが出てしまいます。何が問題なのでしょうか？

問題は、

```
B
```

としてみるとわかります。
これを打ってみると、

```
3×3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

となります。この `Array` が、Bが行列であることを意味しています。Arrayは `A[1,2,3]` のような3つ以上の足をつけることができますので、配列、と呼ばれています。
Bは `3×3 Array{Float64,2}` のようです。
ここで、 `3x3` は3x3行列であることを意味しています。次の `Array{Float64,2}` は、行列要素の中身が `Float64` であり、足が２つある、ということを意味しています。 `Float64` とは、倍精度実数であり、変数の種類のことをさします。
例えば、

```
C = [1 2
     3 4]
```

とすると、

```
2×2 Array{Int64,2}:
 1  2
 3  4
```

となりまして、 `Array{Int64,2}` は、行列要素の中身が `Int64` であることを意味しています。つまり、行列要素が整数、ということを言っています。
このような変数の種類のことを「型」と言います。
先ほどの行列Bは行列要素がFloat64でなければならない、ということです。ただし、

```
B = zeros(3,3)
B[1,2] = 4
```

は可能です。4は整数ですが、整数から実数へは変換することができます。4が4.0になるわけです。しかし、 `4+2im` のような複素数の場合、これは実数に変換できません。そのため、エラーが出ました。
これを解決するためには、

```
B = zeros(ComplexF64,3,3)
B[1,2] = 4 + 2im
```

とすればよいです。ここで、 `zeros(ComplexF64,3,3)` の `ComplexF64` は倍精度複素数型を意味しています。このように、行列を初期化（zerosで０行列を作る）ときには、入れる行列要素の種類（型）が合っていなければなりません。

なお、初期化するにはもう一つ方法がありまして、

```
B = Array{ComplexF64}(undef,3,3)
```

とすると、行列要素の型は倍精度複素数 `ComplexF64` だけれども行列要素が何も定義されていない行列、を定義することができます。

以下執筆予定