物理で使う数値計算入門：Julia言語による簡単数値計算

作成日　令和元年5月1日
更新日　令和元年5月11日
永井佑紀

目次

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

2.1. 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。

2.2. 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。

2.3. 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。

2.4. 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。

2.5. 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。

なお、REPLを終了するには、

```
exit()
```

とします。

3.1.1. 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。

3.1.2. 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。

3.1.3. 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。

3.1.4. 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。

3.1.5. 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。

3.1.6. 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。

3.1.7. 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。

3.1.8. 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。

3.2.1. 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。

3.2.2. 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。

3.2.3. 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。

3.3. 数式をコードにしてみよう

物理で使うような数式をコードにしてみましょう。
紹介する具体例を通じて、Juliaでどのようにコードを書けばよいかを紹介します。

3.3.1. 関数の振る舞いをみる：forループとplot

ある関数：

$$f(x) = \cos(x)e^{-x}$$

という関数の振る舞いが知りたいとします。
一番簡単な方法は、xに様々な値を入れて値を見てみることでしょう。
例えば、x=0からx=1まで、n点でのf(x)の値を見ることにします。
Juliaでは様々な書き方ができますので、順番にみていくことにします。

3.3.1.1. forループ1

一番素朴な方法は、x=0から1まで少しずつ値を増やして様子を表示することでしょう。つまり、１番目はx=0で、2番めは0+1/(n-1)、i番め0+(i-1)/(n-1)と少しずつxを増やしていって値を計算してみます。
このような繰り返しをする構文として、for文があります。
例えば、

```
for i=1:10
    println(i)
end
```

とすると、1から10までが表示されます。2ずつ増やしたい場合には、

```
for i=1:2:10
    println(i)
end
```

ですし、負にも動かすことができて、10から1ずつ減らしたければ、

```
for i=10:-1:1
    println(i)
end
```

となります。このように、forループを使えば繰り返しを実行できます。
そこで、「１番目はx=0で、2番めは0+1/(n-1)、i番め0+(i-1)/(n-1)と少しずつxを増やしていって値を計算」するには、

```
f(x) = cos(x)*exp(-x)
n=10
for i=1:n
    x = (i-1)/(n-1)
    println(f(x))
end
```

とすればよいです。上のコードではf(x)だけ表示していますが、その時のxの値などを表示したければ、

```
f(x) = cos(x)*exp(-x)
n=10
for i=1:n
    x = (i-1)/(n-1)
    println("$x $(f(x))")
end
```

とします。ここで、printlnの別の文法が登場しました。""で囲まれたものは文字列です。文字列の中に変数を入れるためには、 `$(a)` などとします。なお、一文字であれば `$a` でも構いません。これを使って `println("$(a) $(b)")` とすると、変数aや変数bを表示することができます。この辺りは後述します。

3.3.1.2. forループ2

x=0から1まで少しずつ増やす方法はFortranなどの古くからあるプログラミング言語ではよく使われていた方法です。Juliaでは、上のようにxの増分をはっきり書かなくてもよい方法があります。

```
f(x) = cos(x)*exp(-x)
n = 10
xs = range(0, 1, length=n)
for x in xs
    println("$x $(f(x))")
end
```

このコードでは、「1番目はx=0で、2番めは0+1/(n-1)、i番め0+(i-1)/(n-1)と少しずつxを増やしていった値」を `range` であらかじめ求めてしまっています。 `rand(start,stop,length=n)` とすると、最初がstart、終わりがstop、長さがnの塊を作ってくれます。
そして、for文では `for i=1:n` の代わりに `for x in xs` という表記を使っています。これは、 `xs` の中身を `x` として順番に取り出す、という意味です。

3.3.1.3. ベクトルや行列の要素ごと計算

上の二つでは、for文を使ってそれぞれのxでのf(x)を求めていましたが、Juliaではこれをfor文を使わずにまとめて計算する方法もあります。それは、要素ごと計算です。

例えば、

```
f(x) = cos(x)*exp(-x)
n = 10
xs = range(0, 1, length=n)
f.(xs)
```

とすると、xsの中身のそれぞれに対してf(x)を計算します。
上と同様に表示させたければ、

```
f(x) = cos(x)*exp(-x)
n = 10
xs = range(0, 1, length=n)
fs = f.(xs)
for i=1:n
    println("$(xs[i]) $(fs[i])")
end
```

とします。ここでは、for文を使って、iを1からnまで回して、xsとfsのそれぞれの要素
を `xs[i]` 、 `fs[i]` として取り出しています。

ここで出てきたのは、ドット `.` です。ドットを使うと、行列やベクトルの要素のそれぞれに計算をすることができます。

例えば、

```
A = [1 2 3
4 5 6
7 8 9 ]
B = A .+ 3
```

とすれば、行列Bは行列Aの各要素に3を足したものになります。数学では、B = A+3と書くと自動的にAの各要素に3を足すと理解されますが、Juliaの場合には明示的に `.+` とすることで要素ごとに足すことを指定します。

3.3.1.4. プロット

関数の振る舞いを見るにはプロットすることも重要です。
Juliaにはプロットする関数もあります。
Juliaでは、非常に多彩な「パッケージ」と呼ばれるものがあり、それを追加することで新しい機能を使うことができます。プロットの場合にはPlots.jlというパッケージが有名です。
これを使うには、] キーを押して「パッケージモード」：

```
(v1.1) pkg>
```

にします。そして、

```
add Plots
```

と入れることで、プロットのパッケージを追加することができます。パッケージモードを終了するには、delキーを押してください。なお、一度入れたパッケージは毎回入れ直す必要はありません。

インストールしたPlots.jlを使うには、パッケージモードを終了した後に、

```
using Plots
```

としてください。これでプロット関連の関数が使えるようになります。
上の関数をプロットしたい場合には、

```
plot(xs,fs)
```

とすれば、グラフが表示されます。

3.3.2. 級数の和の計算：sumとリスト内包表記、If文、while文、引数の型、多重ディスパッチ

次に、

$$f(x) = \sum_{n=0}^{\infty} \frac{1}{n!} x^n$$

を計算してみましょう。この級数は指数関数exp(x)のテイラー展開ですね。
計算機では無限の和を取れないので、nの最大値を指定する必要があります。
ですので、nの最大値nmaxをパラメータとしましょう。

3.3.2.1. シンプルな場合:リスト内包表記

シンプルな方法として

```
f(x,nmax) = sum([x^n/factorial(n) for n=0:nmax])
f(2,10)
```

を紹介します。まず、 `[2*n for n=0:10]` のような構文をリスト内包表記と呼びます。これは、for文にある変数（ここではn）を回しながら配列を作ります。例えば、上のコードであれば、nは0からnmaxまで動きますので、要素数がnmax+1の配列ができます。そして、 `sum` という関数は、配列の中身を足す関数です。つまり、作った配列の中身を合計しているので、これは数式でいう所のsumになっています。なお、 `factorial(n)` はnの階乗を計算する関数です。

3.3.2.2. forループによる足し算

次に、forループで足してみます。
この場合、fを一行で書くよりもfunctionで定義した方がみやすいです。ですので、

```
function f(x,nmax)
    fsum = 0
    for n=0:nmax
        fsum += x^n/factorial(n)
    end
    return fsum
end
f(2,10)
```

としてみましょう。このコードでは、まずfsumを0に初期化して、そのあとはforループでひたすら項を足しています。そして、functionの結果として、fsumを返すようにしています。ここで `+=` という新しい文法が登場しました。これは、「左辺にある変数に右辺にある値を足す」、という意味です。同様に、 `-=` や `*=` 、 `/=` などもあります。
さて、functionの最後に `return` という文があります。これは、結果を返す変数を指定するものです。 `return a` とすると、変数aが結果として返ります。ここは複数にすることができて、その場合には `return a,b` などとなります。数学での関数では通常一つしか結果が出てきませんが、プログラムとしてはそこを一つと制限する必要はありませんので、複数結果を返すことができます。
プログラムで数値計算する場合には多くの場合関数が一行で書けるとは限りませんので、このようにで囲って `function` と `end` で関数を定義します。

3.3.2.3. 精度コントロール：If文とwhile文の紹介

次に、nmaxをどこまで取るか、という問題を考えます。
計算したい和は本来無限まで続くのですが、計算機では無限に和は取れません。ですので、なるべく大きなnmaxを取る必要があります。しかし、どこまでとれば良いでしょうか？
ここでは、If文を使って、設定した精度に到達するまで和を取るような関数を作ることにします。

まず、If文ですが、Juliaでは

```
a = 3
if a > 4
    println("a>4")
elseif a == 4
    println("a = 4")
else
    println("a < 4")
end
```

のようにします。「もしa>4ならば」は `if a >4` です。また、「あるいはもしa=4ならば」、
は `elseif a ==4` です。それ以外は `else` の後に書きます。このifの後に書かれているものを条件式と

呼びます。
条件式は真か偽になるものを入れることができます。例えば、

```
if true
    println("真")
end
```

のように `true` そのものを入れることができます。a=3の時、 `a > 4` は偽ですので、

```
a = 3
a > 4
```

とすると、 `false` が返ります。これを変数として使うこともものできて、

```
c = a >4
if c
    println("真")
else
    println("偽")
end
```

ともできます。ここでのcの型はbool型と呼ばれます。trueかfalseが入る型、という意味です。

さて、このif文を使って、精度を決めた数式を関数にしてみましょう。

```
function f(x,eps)
    fsum = 0
    fsumold = 0
    hi = 1
    n = 0
    while hi > eps
        fsum += x^n/factorial(n)
        hi = abs(fsum-fsumold)/abs(fsum)
        fsumold = fsum
        n += 1
    end
    return fsum,n
end
fapp,n = f(2,1e-4)
println(fapp-exp(2)," $n")
```

このコードでは、forループの代わりにwhileループを使っています。
whileループでは、 `while c` のように書き、「cが満たされている間ずっと繰り返す」という意味です。今回は、 `abs(fsum-fsumold)/abs(fsum)` で前回までの和との差を計算しており、その差がepsよりも大きい間はnを増やしてループを続けています。

また、 `return fsum,n` となっていますので、結果は二つの変数fsum,nが返ってきます。ですので、 `fapp,n = f(2,1e-4)` とすると、一個目の結果をfappに、二個目の結果をnに入れています。
epsの値を色々変化させて、指定された精度に達するのにどのくらいのnmaxが必要かを見てみると良いかもしれません。

3.3.2.4. 解析接続：引数の型の自由

さて、ここまで関数f(x)のxには実数を入れてきました。しかし、このxに複素数を入れても問題ありません。つまり、解析接続ですね。xに純虚数を入れると、この式は指数関数exp(ix)の定義になりますので、cos(x)+i sin(x)となります。
Fortranやc言語の場合には、関数の引数（ここではx）の型を指定しなければなりません。Juliaでは、xにどんな型が来ても計算可能であれば計算できます。
例えば、

```
fapp,n = f(0.1im,1e-4)
println(fapp-exp(0.1im),"  $n")
```

とすれば、複素数のxに対して計算ができます。

3.3.2.5. 同じ名前の関数の定義：多重ディスパッチ

この説で、関数として、nmaxを指定するものと、epsを指定するものを作りました。この両方とも使いたい場合があると思います。もともと計算したい関数は同じですので、同じ名前のfunctionとしたいです。
以下の二つの関数を定義してみましょう。

```
function f(x,nmax::Int)
    fsum = 0
    for n=0:nmax
        fsum += x^n/factorial(n)
    end
    return fsum
end

function f(x,eps::Real)
    fsum = 0
    fsumold = 0
    hi = 1
    n = 0
    while hi > eps
        fsum += x^n/factorial(n)
        hi = abs(fsum-fsumold)/abs(fsum)
        fsumold = fsum
        n += 1
    end
    return fsum,n
end

println(f(2,10))
println(f(2,1e-5))
```

一つ目のfunctionでは、 `nmax:::Int` としています。これは、引数のnmaxはIntすなわち整数でなければならない、という意味です。二つ目のfuncitonでは、 `eps::Real` となっており、引数のepsはRealすなわち実数でなければならない、という意味です。このように、Juliaでは、「同じfunction名を使って、引数の型に応じて呼ぶものを変える」ことができます。これを多重ディスパッチと言います。この時の関数の引数の数も変えることができます。
ですので、

```
function f(x;eps=1e-5)
    fsum = 0
    fsumold = 0
    hi = 1
    n = 0
    while hi > eps
        fsum += x^n/factorial(n)
        hi = abs(fsum-fsumold)/abs(fsum)
        fsumold = fsum
        n += 1
    end
    return fsum,n
end
println(f(2))
println(f(2,eps=1e-10))
```

ということもできます。ここで、 `f(x;eps=1e-5)` としていますが、この `;` 以降のepsは、キーワード引数と呼ばれるもので、 `f(2)` のように省略することができます。もし変えたい場合には、 `f(2,eps=1e-10)` などのようにします。

3.3.3. 特殊関数

次に、物理でよく使われる特殊関数を扱ってみましょう。
ということで、第一種変形ベッセル関数を使った式

$$g(x) = \sum_{n=0}^{10}(I_n(x) + n^2 I_{2n}(x))$$

を計算してみます。ここで、$I_n(x)$は次数がnの第一種変形ベッセル関数です。
特殊関数を使うためには、SpecialFunctions.jlを使います。
まず、]を押してパッケージモードにしてから、

```
add SpecialFunctions
```

をしてパッケージをインストールします。その後、delキーを押してパッケージモードを終了します。
以後は、 `using SpecialFunctions` とすることで使うことができます。
上の関数を計算するコードは

```
using SpecialFunctions
function g(x)
    gsum = 0
    for n=0:10
        gsum += besseli(n,x)+n^2*besseli(2n,x)
    end
    return gsum
end
```

です。これをプロットするためには、

```
using Plots
xs = range(0, 1, length=100)
temp = g.(xs)
plot(xs,temp)
```

とします。
なお、SpecialFunctionsの中にどのような特殊関数があるかは、

https://github.com/JuliaMath/SpecialFunctions.jl/blob/master/docs/src/index.md
をみるとわかります。

3.3.4. 1次元数値積分

物理では積分することは非常に多いです。
ここでは、数値積分をやってみましょう。積分する関数は、一次元の波数空間における積分：

$$I = \frac{1}{2\pi}\int_{-\pi}^{\pi} dk f(k)$$

とします。
関数$f(k)$は試しに

$$f(k) = \sin(k) + k^2$$

としてみます。積分は手で
できて、

$$I = \frac{1}{2\pi}\frac{2}{3}(\pi)^3 = 3.2898681336964524$$

となります。

3.3.4.1. 台形公式による積分

まず、一番基本的な数値積分である、台形公式による積分をやってみましょう。
積分をN分割すると、

$$\frac{1}{2\pi}\int_{-\pi}^{\pi} dk \sim \frac{1}{2\pi}\sum_i f(k_i)\frac{2\pi}{N} = \frac{1}{N}\sum_i f(k_i)$$

となります。ここで、台形公式の最初の点と最後の点が同一の点であることを利用し、ただの和にすることができることを用いました。
Juliaではfunctionを引数にすることができます。
したがって、

```
function daikei(f,N)
    dk = 2π/(N-1)
    fsum = 0
    for i=1:N
        k = (i-1)*dk - π
        fsum += f(k)
    end
    fsum /= N
    return fsum
end

f(x) = sin(x) + x^2
N = 400
fsum = daikei(f,N)
exact = ((π)^3/3 -(-π)^3/3)/(2π)
println("daikei $fsum, exact $exact")
```

で台形公式による数値積分ができます。

3.3.4.2. パッケージを使った数値積分

次に、Juliaのパッケージを使ってみましょう。
１次元の数値積分パッケージはQuadGK.jl
https://github.com/JuliaMath/QuadGK.jl
があります。
使うためには、]を押してパッケージモードにしてから、

```
add QuadGK
```

をしてパッケージをインストールします。
QuadGKでは、 `quadgk(f,a,b)` で、関数f(x)とaからbの積分区間で積分ができます。

```
using QuadGK
f(x) = sin(x) + x^2
fsum2 = quadgk(f,-π,π)[1]/(2π)
exact = ((π)^3/3 -(-π)^3/3)/(2π)
println("quadgk $fsum2, exact $exact")
```

ここで、 `quadgk` のアウトプットは `(I,E)` という形で二つでてきます。ここで、丸括弧はタプル、と呼ばれるものです。配列は[]でしたが、タプルは()です。配列は要素の中身を変更できますが、タプルは変更できない、という違いがあります。
アウトプットのうちIは積分の値、Eは誤差です。
この手法の方が、台形公式よりもはるかに精度が高いことが見てとれると思います。

3.3.5. 常微分方程式

常微分方程式：

$$\frac{du}{dt} = 1.01u$$

を解いてみましょう。常微分方程式を解くためのパッケージは、
DifferentialEquations.jlです。
http://docs.juliadiffeq.org/latest/

使うためには、]を押してパッケージモードにしてから、

```
add DifferentialEquations
```

としてください。

例えば、

$$\frac{du}{dt} = 1.01u$$

という微分方程式を初期値$u(t=0)=0.5$で$t=0$から$t=1$まで解きたい場合を考えます。

この場合、

```
using DifferentialEquations
f(u,p,t) = 1.01*u
u0=1/2
tspan = (0.0,1.0)
prob = ODEProblem(f,u0,tspan) #微分がfで初期値がu0の微分方程式du/dt = f(u)を解く。
sol = solve(prob,Tsit5(),reltol=1e-8,abstol=1e-8) #Tsit5は解き方を指定
nt = 50
t = range(0.0, stop=1.0, length=nt) #0.0から1.0までのnt点を生成する
for i=1:nt
    println("t= $(t[i]), solution: $(sol(t[i])), exact solution $(0.5*exp(1.01t[i]))")
end
```

とします。
なお、厳密解は$u = 0.5\exp(1.01t)$なので、それとの比較を出力しました。

微分方程式にパラメータがある場合やuがベクトルや行列の場合でも解くことができます。
DifferentialEquations.jlのドキュメントにあるようなx,y,zの三次元空間におけるローレンツ方程式

$$\frac{dx}{dt} = \sigma(y-x)$$
$$\frac{dy}{dt} = x(\rho-z)-y$$
$$\frac{dz}{dt} = xy-\beta z$$

は、

```
function parameterized_lorenz(du,u,p,t) #微分dy、関数u、パラメータp、変数t
 du[1] = p[1]*(u[2]-u[1])
 du[2] = u[1]*(p[2]-u[3]) - u[2]
 du[3] = u[1]*u[2] - p[3]*u[3]
end

u0 = [1.0,0.0,0.0] #x=1,y=0,z=0を初期値とする。
tspan = (0.0,1.0) #時間は0から1まで。
p = [10.0,28.0,8/3] #パラメータ三つ
prob = ODEProblem(parameterized_lorenz,u0,tspan,p)
```

で解くことができます。

また、マクロ@ode_defを用いるともう少し数学っぽい書き方ができます。
マクロ、とは、 `@` から始まるもので、これをつけるとその部分のコードを目的に応じて自動で書き換えることができるものです。有名なものとしては、時間を測る `@time` マクロがあります。
マクロを使って問題を解くコードは、]でParameterizedFunctions.jlをインストールした後に、

```
using ParameterizedFunctions
g = @ode_def LorenzExample begin
  dx = σ*(y-x) #dxとあるので、xが微分されるもの。
  dy = x*(ρ-z) - y
  dz = x*y - β*z
end σ ρ β #パラメータは三つある

u0 = [1.0;0.0;0.0] #x=1,y=0,z=0を初期値とする。
tspan = (0.0,1.0) #時間は0から1まで。
p = [10.0,28.0,8/3] #σ ρ β を設定
@time prob = ODEProblem(g,u0,tspan,p)
```

とすればよいです。
ここで、 `@time` マクロで解くのにかかった時間を計測しています。

3.3.6. 非線形関数の最小値

非線形関数$f(x)$を最小化する値xを求める問題もよくある問題です。
これは、Optimというパッケージを使えばよいです。
http://julianlsolvers.github.io/Optim.jl/stable/#user/minimization/
使うためには、]を押してパッケージモードにしてから、

```
add Optim
```

としましょう。

それでは、

$$f(x,y) = (1.0 - x)^2 + 100(y - x^2)^2$$

を最小化するx,yの組を求めてみましょう。コードは、

```
using Optim
f(x) = (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
x0 = [0.0, 0.0] #初期値を0,0とした。
a1 = optimize(f, x0)
xsol = Optim.minimizer(a1) #関数fを最小化するxの値
println("xsol = $xsol")
fmin = Optim.minimum(a1) #関数fの最小値
println("fmin = $fmin")
```

となります。
minimizerというのが、最小化する変数を得るもので、minimumというのは関数の値を求めるものとなっています。

次に、

$$f(x) = 2x^2 + 3x + 1$$

という関数が$x = -2$から$x = 1$の範囲で最小となるxを求めてみましょう。
コードは、

```
using Optim
f2(x) = 2x^2+3x+1
a2 = optimize(f2, -2.0, 1.0) #-2から1の間の最小値を探す。
xsol = Optim.minimizer(a2) #関数f2を最小化するxの値
println("xsol = $xsol")
fmin = Optim.minimum(a2) #関数f2の最小値
println("fmin = $fmin")
```

となります。
このように、非線形関数の最小値探索も簡単に計算することができます。

以下執筆予定