JAEA CCSE Center for computational science & e-systems RIKEN AIP

MLPhYs 学術変革領域研究(A) 学習物理学の創成
Foundation of "Machine Learning Physics"

Julia入門

国立研究開発法人日本原子力研究開発機構　システム計算科学センター

理化学研究所　革新知能統合研究センター (AIP)

永井佑紀

アウトライン

- イントロダクション
- 数値計算をする、とはどういうことか
- Juliaとはどんなプログラミング言語か
- Juliaでのライブコーディング：天体の運動　その場でゼロからコーディング
- まとめ

イントロダクション

自己紹介

永井佑紀

専門：物性理論等

日本原子力研究開発機構システム計算科学センター

理化学研究所 革新知能統合センター(AIP)

スパコンを使って、数千-数万CPUコアMPI並列計算を実行している

最近の研究分野：機械学習を物理学に使う

量子多体系へとか格子量子色力学へとか、分子動力学へとか

機械学習を「高速化」に使う話が好き（人間の代わりに頑張れ）

使えるプログラミング言語(得意な順)　Julia, Fortran2008以降, Python, c++

Juliaを知ったきっかけ

2016-2017(MIT客員研究員時代@米国ボストン)

同僚にJuliaを勧められたのでやってみる

ちょうどMITにいるしFortranと同じ配列が1始まりだし

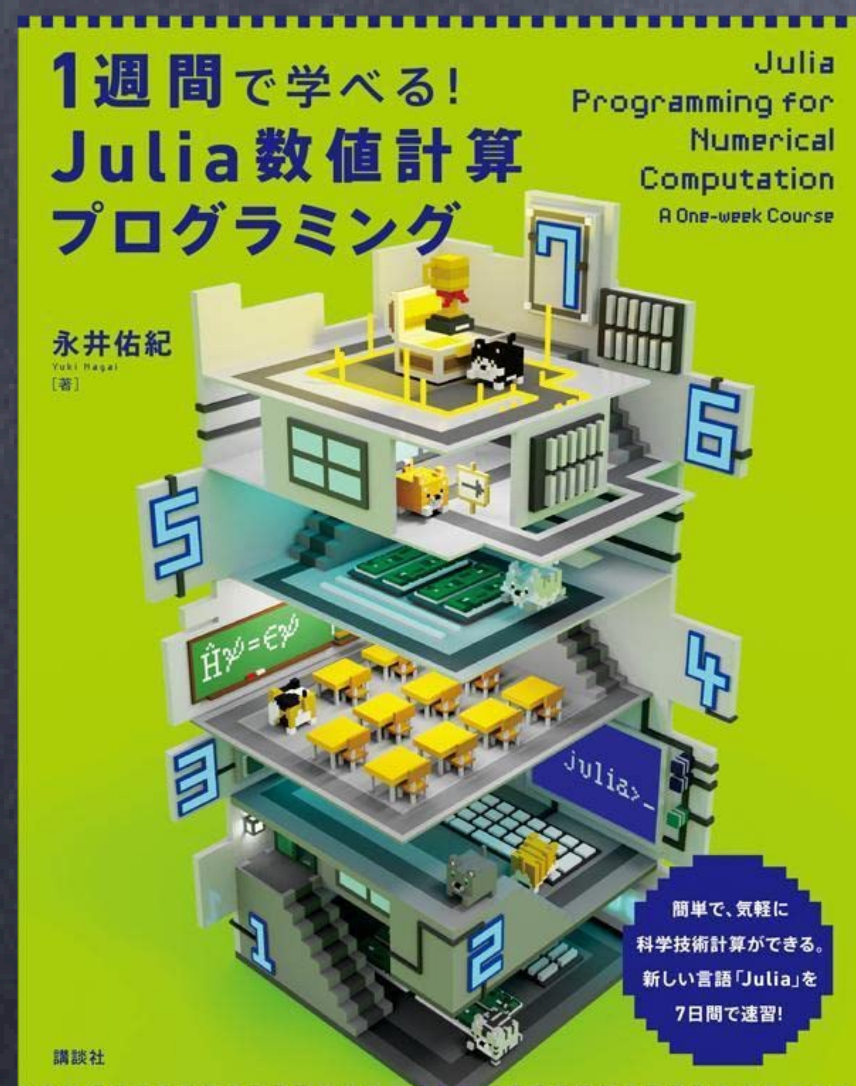
便利なパッケージがたくさんあるし速いし -> Juliaの数値計算で論文を書く

今日の講義について

目的

1. Juliaというプログラミング言語を知る
2. Juliaで数値計算するのが便利だということを知る

後で試したくなる？



1週間で学べる！Julia数値計算プログラミング　講談社

この本を読んでみたくなる？？

数値計算をする、とはどういうことか

数値計算とは？

数値計算　人間ができないor面倒or大変なことを計算機にやらせる

物理系の学科の学部で習うこと

力学、熱力学、統計力学、電磁気学、量子力学etc...

紙と鉛筆で学習：講義に数値計算は出てこない

何故？

容易に手で解ける問題しか扱っていないから

手で解ける領域

残りは解けない

物理で現れる計算

2つの物体同士に働く万有引力

$$F = -G\frac{Mm}{r^2}$$

r:2つの物体間の距離

運動方程式を立てると

$$m\vec{a} = \vec{F} \qquad \vec{F} = -\frac{GMm}{r^3}\vec{r}$$

2本の微分方程式となる

$$\frac{d\vec{v}}{dt} = -\frac{GM}{r^3}\vec{r} \qquad \frac{d\vec{r}}{dt} = \vec{v}$$

これを解けば2つの物体の運動がわかる

2つなら手で色々やることで性質がわかる

N個の物体がある場合は？

物理で現れる計算

N個の物体同士に働く万有引力

$$F_i = -G\sum_j \frac{m_i m_j}{|\vec{r}_i - \vec{r}_j|^2}$$

r_i: i番目の物体の位置

運動方程式を立てると

$$m\vec{a}_i = \vec{F}_i \qquad \vec{F}_i = -G\sum_j m_i m_j \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|^2}$$

2N本の微分方程式となる

$$\frac{d\vec{v}_i}{dt} = -G\sum_j m_j \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|^2} \qquad \frac{d\vec{r}_i}{dt} = \vec{v}_i$$

これを解けばN個の物体の運動がわかる

->手では解けない

-> 自然界の振る舞いを調べるには、紙と鉛筆だけでは足りない

物理で現れる計算

1粒子の時間依存しないシュレーディンガー方程式

$$\hat{H}\psi_n(x,y,z) = E_n\psi_n(x,y,z) \quad \hat{H} = -\frac{\hbar}{2m}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}\right)+V(x,y,z)$$

微分方程式を満たすようなエネルギーE_nと波動関数ψ_nを求める問題

-> V(x,y,z)=0なら、手で（フーリエ変換で）解くことができる

-> V(x,y,z)が特別な形をしている時のみ手で解ける

多くの場合は解けない

まだ電子1つの問題：現実には沢山電子がいる

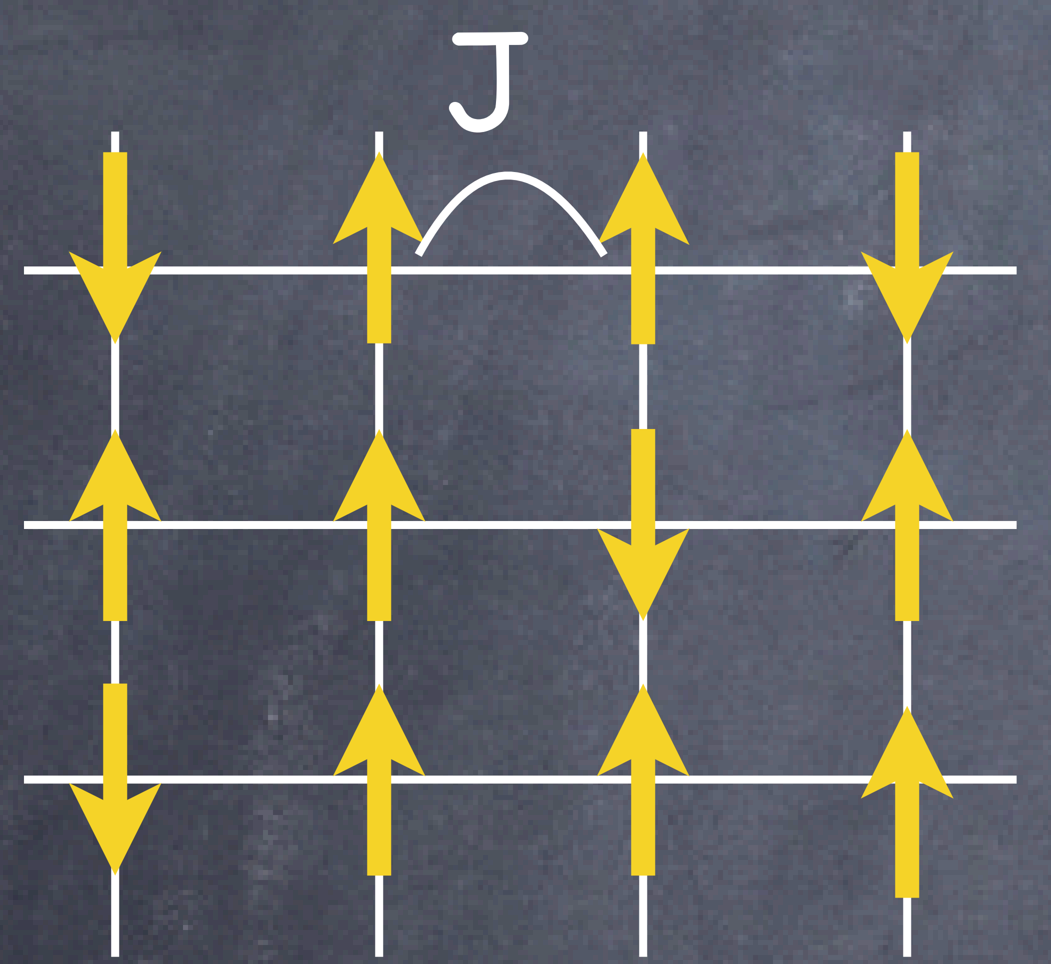
-> 自然界の振る舞いを調べるには、紙と鉛筆だけでは足りない

物理で現れる計算

統計力学

2次元イジング模型

例：磁性



$$H = -J\sum_{\langle ij\rangle}\sigma_i\sigma_j - h\sum_i \sigma_i$$

σ_i: i番目のスピンの値,+1か-1

$$\langle A\rangle = \frac{1}{Z}\sum_{\mathscr{C}}\left[\exp\left(-\frac{H(\mathscr{C})}{k_B T}\right)A(\mathscr{C})\right]$$

物理量の期待値

$\mathscr{C}$ ：あるスピン配置

全ての可能なスピン配置に対して和を取る

電子はスピンを持つ

->小さな磁石

向きが全部揃うと磁石になる

ある温度以下になると、突然磁石になる（相転移）

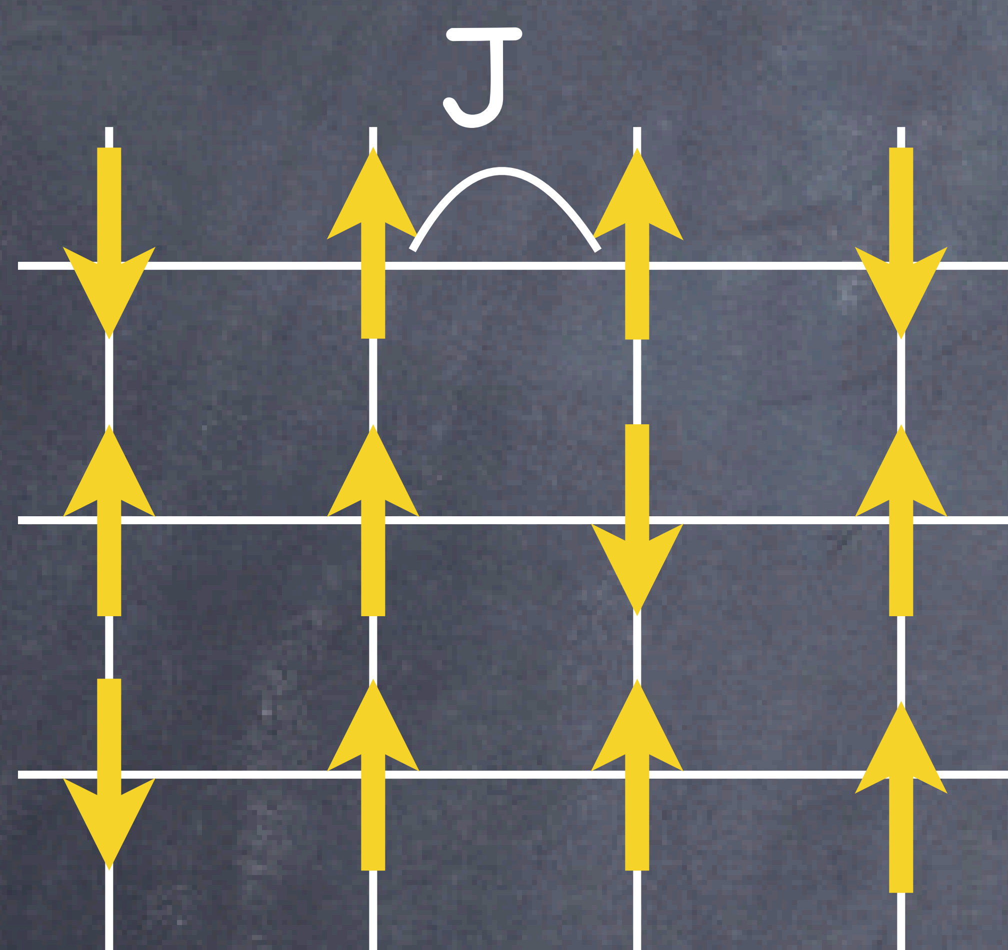
-> 自然界の振る舞いを調べるには、紙と鉛筆だけでは足りない

物理で現れる計算

統計力学

-> 自然界の振る舞いを調べるには、紙と鉛筆だけでは足りない

例：磁性



電子はスピンを持つ

->小さな磁石

向きが全部揃うと磁石になる

2次元イジング模型 $H = -J \sum_{\langle ij \rangle} \sigma_i \sigma_j - h \sum_i \sigma_i$

ただし、数学を駆使すれば解けることもある

2次元イジング模型は手で解けている

手で解けるに越したことはない

$x^2+2x+1 = 0$ の解を数値的に探すよりも

$(x+1)^2=0$, $x=-1$ とわかっていた方が良い

-> 手元の有益な情報が多いほど計算コストが下がる

JAEA CCSE RIKEN AIP

数値計算をする、とはどういうことか

手で解ける領域

残りは解けない

手で解けない面白い問題が沢山ある

計算機は道具である

物理屋は数学と計算機を道具とする

数学科ほど数学を深めない

情報系学科ほど計算機を深めない

どちらも例外あり

現実世界に存在する面白いものを対象とする

どんな方法であれ解ければ良い

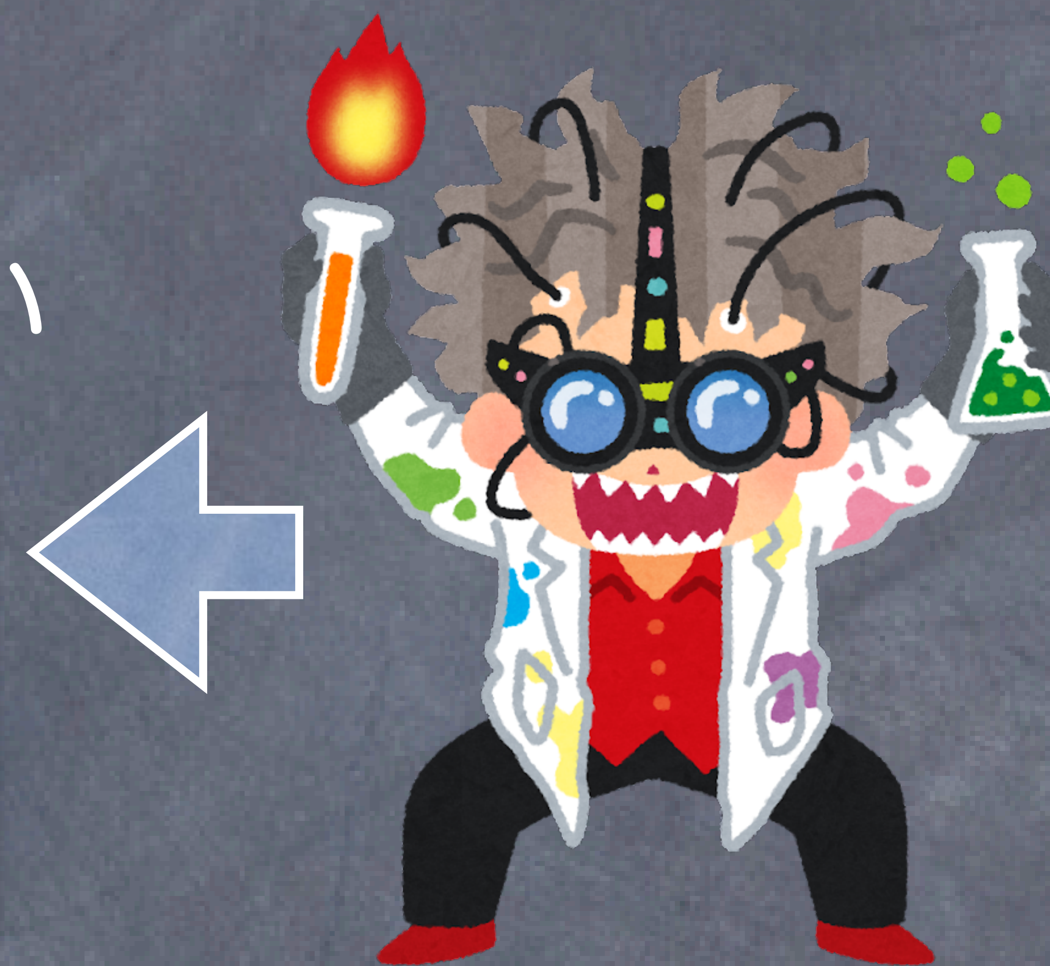
物理学者が数学者を刺激する場合もあり

(多くの)物理屋にとって数値計算とは

物理の問題に注力したい

物理以外の問題に手間をかけ過ぎたくない

自動車を運転して遠くに行きたいのであって、そのために一から車の部品を組み立てたいわけじゃない



現実世界に存在する面白いものを対象とする

どんな方法であれ解ければ良い

一方、

目的地に到達するためにはカスタマイズした移動手段が必要な場合もある

カスタマイズが楽しくなってくる場合もある

計算機科学者との違い：見据えているのは目的地

自由度が高い、書きやすい、そして速いプログラミング言語がいい

Julia言語??

数値計算の種類

1. 理論はすでにあり、適応する対象が無数にあるケース **Fortran,C++**

例：固体物理における「第一原理計算」

2. どのような理論を使うか試行錯誤が必要なケース **Fortran,C++,Python**

いっぱいある

グリーン関数による摂動論

ハミルトニアンの数値的対角化

3. 機械学習が絡むケース **Python**

最先端の機械学習の手法は個人で実装するには大変すぎる -> Pythonライブラリの使用

4. 上記の様々な組み合わせ

Juliaは2,3,4で有用

数値計算の種類

1. 理論はすでにあり、適応する対象が無数にあるケース

固体物理における「第一原理計算」　密度汎関数理論で多体電子系を扱う

原子位置の情報がわかれば（原理的に）全てがわかる

物質をデザインしたり、実験結果と比較したり

もちろん、手法を開発する研究もある

数値計算として重要な点:可能な限り高速に動く

クラスターマシンやスパコンなどで大規模に動かせる

-> FortranやC++で書かれていることが多い

この領域はすでに良いコードが複数あるので、そのまま使った方が良さげ

使い方さえわかれば、内部のコーディングがどうなっているか気にする必要がない

JAEA CCSE RIKEN AIP

数値計算のためのプログラミング言語

数値計算をする場合に、どのプログラミング言語を選べば良いのか

原理的には、どんな言語を選んでも数値計算は可能

例えば

FORTRAN: (formula translation)　1957年にコンパイラがリリース

有名なFORTRAN

FORTRAN77　1978年　**45年前！**

Fortran90　1991-92年　**31年前！**

速い！　**数値計算でいまだに使われているプログラミング言語**

Python　機械学習分野での基本言語の地位を確立（研究における英語みたいな）

Pythonは「知っていないとまずい」言語に

他の言語で書かれた高速なコードを呼ぶ、という形で使われることが多い

数値計算のためのプログラミング言語

数値計算をする場合に、どのプログラミング言語を選べば良いのか

原理的には、どんな言語を選んでも数値計算は可能

物理の問題に注力したい　どんなプログラミング言語が良いか

数式で表現されたものを数値計算したい

Mathematica等

その言語特有のお作法に悩まされたくない

Python: forループが遅いためにnumpyの書き方を習得する必要がある

c++: 行列やベクトルや複素数を扱うためには言語に慣れる必要がある

なるべく「楽に」数値計算がしたい

数値計算の例

1粒子の時間依存しないシュレーディンガー方程式

$$\hat{H}\psi_n(x,y,z) = E_n\psi_n(x,y,z) \qquad \hat{H} = -\frac{\hbar}{2m}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) + V(x,y,z)$$

1. 計算機では連続的な量を扱えないので、微分を差分に直す

$$\frac{\partial^2\psi(x)}{\partial x^2} \sim \frac{1}{2h^2}(\psi(x_{i+1}) - 2\psi(x_i) + \psi(x_{i-1})) \qquad h \equiv x_{i+1} - x_i$$

-> シュレーディンガー方程式は行列の固有値問題になる

2. $N_xN_yN_z$ x $N_xN_yN_z$の行列Hを作成
3. 固有値問題Hψ=Eψを解く

慣れれば難しくないが…… <-

2と3を計算機にやらせる

Pythonは（素朴にやると）2が遅い

C++は3のためにEigen等外部ライブラリをインストールする必要がある

Juliaはどんなプログラミング言語か

数値計算のためのプログラミング言語

数値計算をする場合に、どのプログラミング言語を選べば良いのか

原理的には、どんな言語を選んでも数値計算は可能

物理の問題に注力したい　どんなプログラミング言語が良いか

物理の数値計算では線形代数を多用する

特殊関数も使う

積分も微分もする　-> 数値計算のできる「電卓」が欲しい！

使うのが簡単がいい

インストールとかコンパイルとかに時間を溶かしたくない

バグとりが楽ならなお良い

Julia言語が最適

Juliaはどんな言語か

1次元シュレーディンガー方程式

$$\hat{H}\psi_n(x) = E_n\psi_n(x) \qquad \hat{H} = -\frac{\hbar}{2m}\frac{\partial^2}{\partial x^2} + V(x,y,z)$$

差分化して解いて、V=0の解析解と比較してみる

```julia
function make_H(N,L,V)
    Δx = L/(N+1)
    H = zeros(Float64,N,N)
    for i=1:N
        x = i*Δx
        H[i,i] = V(x)
        j = i+1
        dij = -1/Δx^2
        if 1 ≤ j ≤ N
            H[i,j] += dij
        end

        j=i
        dij = 2/Δx^2
        if 1 ≤ j ≤ N
            H[i,j] += dij
        end

        j=i-1
        dij = -1/Δx^2
        if 1 ≤ j ≤ N
            H[i,j] += dij
        end
    end
    return H
end
```

```julia
using LinearAlgebra

using Plots
function test()
    V(x) = 0
    N = 1000
    L = 1
    H = make_H(N,L,V)
    e,v = eigen(H)
    e0 = zeros(Float64,N)
    for n=1:N
        e0[n]=n^2*π^2/L^2
    end
    plot(1:N,[e,e0],labels=["Numerical result" "Analytical result"],xlabel="n",ylabel="energy")
    savefig("eigen.png")
    println(e0[1],"\t",e[1])
end
test()
```

本当にこれだけのコード

他に”おまじない”はいらない

Juliaはどんな言語か

```
function make_H(N,L,V)
    Δx = L/(N+1)
    H = zeros(Float64,N,N)
    for i=1:N
        x = i*Δx
        H[i,i] = V(x)
        j = i+1
        dij = -1/Δx^2
        if 1 ≤ j ≤ N
            H[i,j] += dij
        end

        j=i
        dij = 2/Δx^2
        if 1 ≤ j ≤ N
            H[i,j] += dij
        end

        j=i-1
        dij = -1/Δx^2
        if 1 ≤ j ≤ N
            H[i,j] += dij
        end
    end
    return H
end
```

1次元シュレーディンガー方程式

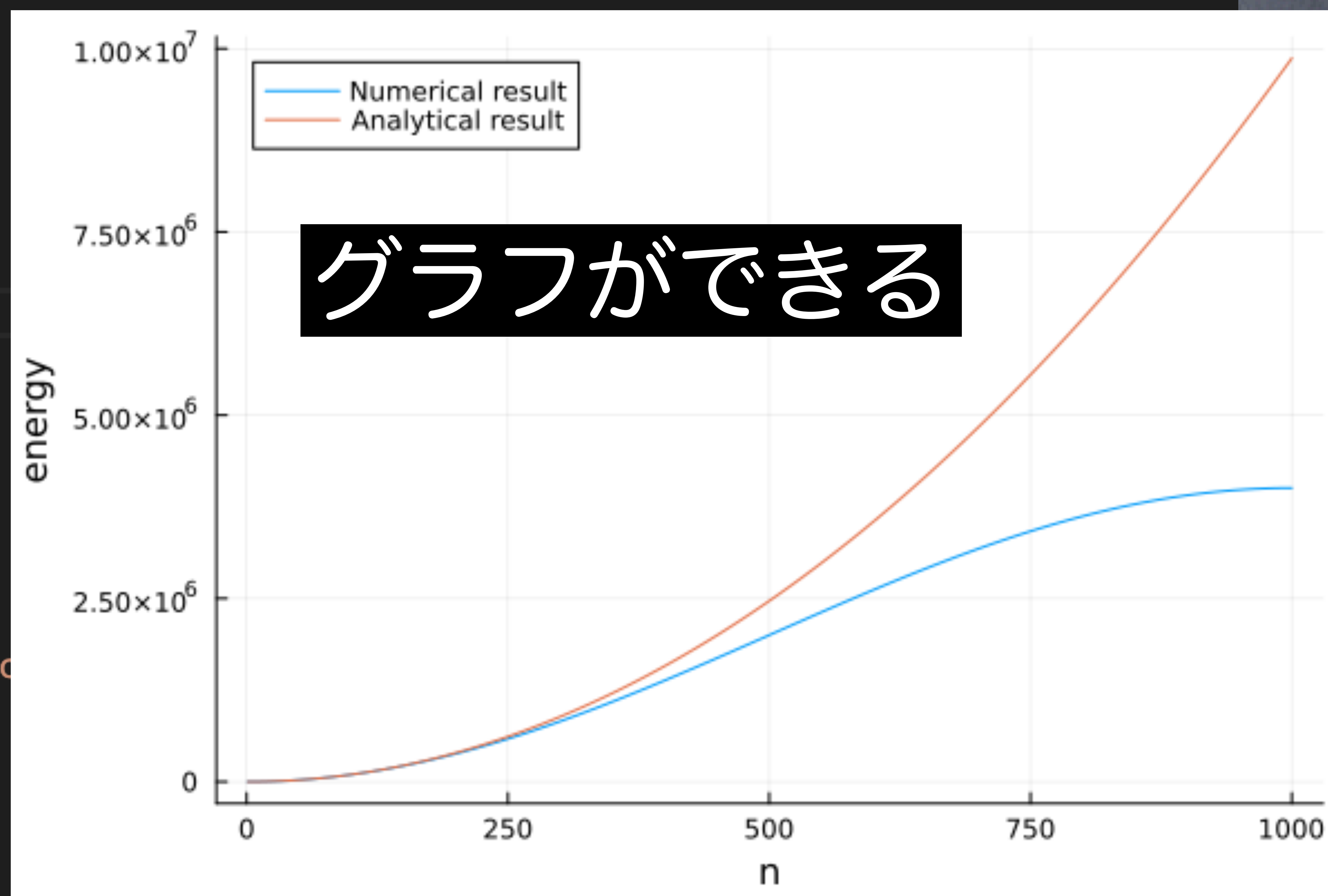
$$\hat{H}\psi_n(x) = E_n\psi_n(x) \qquad \hat{H} = -\frac{\hbar}{2m}\frac{\partial^2}{\partial x^2} + V(x, y, z)$$

差分化して解いて、V=0の解析解と比較してみる

```
using LinearAlgebra

using Plots
function test()
    e0 = zeros(Float64,N)
    for n=1:N
        e0[n]=n^2*π^2/L^2
    end
    plot(1:N,[e,e0],labels=["Numeric
    savefig("eigen.png")
    println(e0[1],"\t",e[1])
end
test()
```

julia qm.jl
と実行すると



Juliaはどんな言語か

トポロジカル超伝導体のハミルトニアン

$$H_{BdG} = \begin{pmatrix} H_N(k_x,k_y,k_z) & \Delta \\ \Delta^\dagger & -H_N(-k_x,-k_y,-k_z)^* \end{pmatrix}$$

$$H_N = \begin{pmatrix} \epsilon(k_x,k_y,k_z)+M(k_x,k_y,k_z) & 0 & P_3(k_x,k_y,k_z) & P_1(k_x,k_y,k_z)-iP_2(k_x,k_y,k_z) \\ 0 & \epsilon(k_x,k_y,k_z)+M(k_x,k_y,k_z) & P_1(k_x,k_y,k_z)+iP_2(k_x,k_y,k_z) & -P_3(k_x,k_y,k_z) \\ P_3(k_x,k_y,k_z) & P_1(k_x,k_y,k_z)-iP_2(k_x,k_y,k_z) & \epsilon(k_x,k_y,k_z)-M(k_x,k_y,k_z) & 0 \\ P_1(k_x,k_y,k_z)+iP_2(k_x,k_y,k_z) & -P_3(k_x,k_y,k_z) & 0 & \epsilon(k_x,k_y,k_z)-M(k_x,k_y,k_z) \end{pmatrix}$$

```
function calc_HBdG(kx,ky,kz,M0,μ,Δ0)
    HN = calc_HN(kx,ky,kz,M0,μ)
    HNast = conj.(calc_HN(-kx,-ky,-kz,M0,μ))
    Δ =  Δ0*Δ4
    HBdG = [
        HN Δ
        Δ' -HNast
    ]

end
```

```
function calc_HN(kx,ky,kz,M0,μ)
    ϵc = 2 - 2*cos(kz)
    ϵperp =  3 - 2*cos(sqrt(3)*kx/2)*cos(ky/2) - cos(ky)
    ϵ = -μ + D1* ϵc +(4/3)*D2*ϵperp
    M = M0 - B1*ϵc - (4/3)*B2*ϵperp
    P1 = (2/3)*A2*sqrt(3)*sin(sqrt(3)*kx/2)*cos(ky/2)
    P2 = (2/3)*A2*(cos(sqrt(3)*kx/2)*sin(ky/2) + sin(ky))
    P3 = A1*sin(kz)
    HN = [ ϵ + M           0          P3        P1 - im*P2
        0                ϵ + M     P1 + im*P2      -P3
        P3            P1 - im*P2       ϵ-M          0
        P1 + im*P2      -P3            0           ϵ-M
    ]
    return HN
end
```

ギリシャ文字（日本語も）使える

行列が直感的に書ける

Juliaはどんな言語か

例：“トポロジカル超伝導体の励起スペクトルの波数空間依存性を詳細に見たい”

適当な範囲で3次元k空間での固有値を求めて

```
kxs = range(-a,a,length=nkx)
kys = range(-a,a,length=nky)
kzs = range(-a,a,length=nkz)
energies = zeros(8,nkx,nky,nkz)
for ikx=1:nkx
    kx = kxs[ikx]
    for iky=1:nky
        ky = kys[iky]
        for ikz=1:nkz
            kz = kzs[ikz]
            HBdG =  calc_HBdG(kx,ky,kz,M0,μ,Δ0)
            e,v = eigen(HBdG)
            energies[:,ikx,iky,ikz] = e[:]
        end
    end
end
```

行列の対角化は1行で書ける

3次元スプライン補間を行なって

```
interp_cubics = []
for i=1:8
    push!( interp_cubics,cubic_spline_interpolation((kxs, kys,kzs),  energies[i,:,:,:]))
end
```

```
nkx_hi = 128*2
nky_hi = 128*2
kxs = range(-0.3,0.3,length=nkx_hi)
kys = range(-0.3,0.3,length=nky_hi)
kz = 0.0
fp = open("energy.txt","w")
for kx in kxs
    for ky in kys
        print(fp,"$kx $ky ")
        for i=1:8
            ei = interp_cubics[i](kx,ky,kz)
            print(fp,"$ei ")
        end
        println(fp,"\t")
    end
end
close(fp)
```

より細かな波数空間のメッシュを切ってファイルに書き出す

Juliaはどんな言語か

“量子力学を機械学習で解く最新論文の結果(の一部)を再現したい”

$$\hat{H}\psi_n(x) = E_n\psi_n(x) \qquad \hat{H} = -\frac{\hbar}{2m}\frac{\partial^2}{\partial x^2} + V(x,y,z) \qquad E_n = \frac{\int dx \psi_n(x)^\dagger \hat{H}\psi_n(x)}{\int dx \psi_n(x)^\dagger \psi_n(x)}$$

エネルギー固有値Enが最小となるような波動関数を機械学習で見つける

$\psi_n(x_i) = f(x_i)$　f(x):ニューラルネットワーク

A simple method for multi-body wave function of ground and low-lying excited states using deep neural network

Tomoya Naito, Hisashi Naito, Koji Hashimoto, arXiv:2302.08965

やること

1. ハミルトニアンの微分演算子を差分化
2. エネルギーが最小となるような波動関数を探索

論文では、Pythonの機械学習フレームワークTensorFlowを使用しているらしい

Juliaはどんな言語か

“量子力学を機械学習で解く最新論文の結果(の一部)を再現したい”

ハミルトニアンの微分演算子を差分化

```
using SparseArrays
using LinearAlgebra
function make_T(M)
    T = spzeros(Float64,M-1,M-1)
    for i=1:M-1
        j = i -1
        if 1 <= j <= M-1
            T[i,j] = 1
        end
        j = i + 1
        if 1 <= j <= M-1
            T[i,j] = 1
        end
        T[i,i] = -2
    end
    return T
end
```

運動エネルギー項

```
function make_V(M,ω,xs)
    V = spzeros(Float64,M-1,M-1)
    for i=1:M-1
        x = xs[i]
        V[i,i] = (1/2)*ω^2*x^2
    end
    return V
end
```

調和振動子ポテンシャル項

```
using Flux
n1 = 4
model = Chain(x ->
[x],Dense(1,n1,Flux.softplus),Dense(n1,1,
Flux.softplus),x -> sum(x)) |> Flux.f64
```

ニューラルネットワークを構築

隠れ層1層、活性化関数softplusのニューラルネット

```
function energy(model,xs)
    ψ = model.(xs)
    c = ψ'*ψ
    E = ψ'*H*ψ/c
    return E
end
```

$$E_n = \frac{\int dx \psi_n(x)^\dagger \hat{H} \psi_n(x)}{\int dx \psi_n(x)^\dagger \psi_n(x)}$$

エネルギーの定義

あとは訓練するだけ

PyTorchやTensorFlowと違い、
テンソルにする必要がない

Juliaはどんな言語か

“量子力学を機械学習で解く最新論文の結果(の一部)を再現したい”

Adamというオプティマイザーを使用

```
opt = Flux.setup(Adam(), model)
for i=1:50000
    Flux.train!(energy, model,(xs,), opt)
    e = energy(model,xs)
    if i % 1000  == 0
        println("i = $i energy = $e")
    end
end
```

energyを最小化するようにmodelを訓練

modelを使ったenergyの計算

訓練はこれだけ

これだけで最新の物理学with機械学習の結果を再現できる

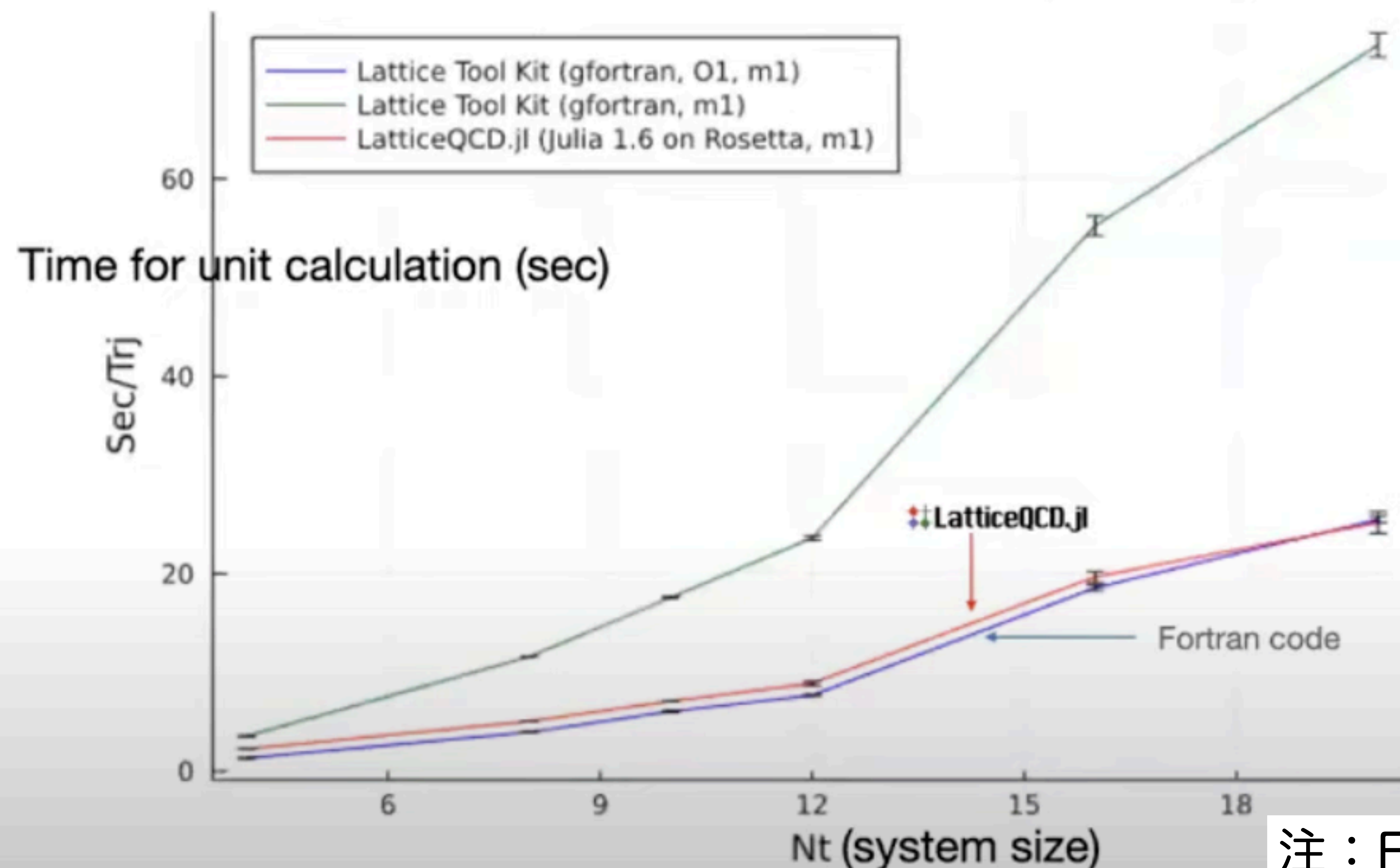
Juliaはどんな言語か

初学者がとっつきやすいシンプルな文法 and FortranやC言語並に高速

重要な点：複雑なコードでも高速！

格子量子色力学（格子QCD）のJuliaコード、LatticeQCD.jl :富谷氏の個別講義

Same algorithm, same parameter



Machine:
Mac mini (M1, 2020), memory 8GB
- Julia 1.6.1 + **Rosetta2**

Parameters
$L=4^3 \times Lt$
Lt = 4, 8, 10, 12, 16, 20
kappa = 0.141139
beta = 5.5
Nd = 10
CG eps = 10-8

Compare with Lattice tool kit (2002)
- gfortran11 (w/ & wo O1)
(O2 and O3 do not work)

Fortranコードに匹敵する速度

注：Fortranコードはガチチューニング済みコードではない

Julia言語

2018年にバージョン1がリリースされた比較的新しい言語

特徴：数式に近い形で計算ができる　電卓のような手軽さ and 高速！

無料：インストールも簡単　https://julialang.org/downloads/

Current stable release: v1.8.5 (January 8, 2023)

Checksums for this release are available in both MD5 and SHA256 formats.

Windows [help]	64-bit (installer), 64-bit (portable)		32-bit (installer), 32-bit (portable)
macOS x86 (Intel or Rosetta) [help]	64-bit (.dmg), 64-bit (.tar.gz)		
macOS ARM (M-series Processor) [help]	64-bit (.dmg), 64-bit (.tar.gz)		
Generic Linux on x86 [help]	64-bit (glibc) (GPG), 64-bit (musl)[1] (GPG)		32-bit (GPG)
Generic Linux on ARM [help]	64-bit (AArch64) (GPG)		
Generic Linux on PowerPC [help]	64-bit (little endian) (GPG)		
Generic FreeBSD on x86 [help]	64-bit (GPG)		
Source	Tarball (GPG)	Tarball with dependencies (GPG)	GitHub

自分のOSに合わせて
ダウンロードする

JAEA CCSE RIKEN AIP

Julia言語の使い方

大きく分けて三種類の方法がある

1. アプリを起動する（ダブルクリックで開く）

    手軽：電卓のように使える

2. Jupyter notebookを使う

    ブラウザで使える。Pythonのnotebookと似た操作感。Mathematicaに似ている

3. 端末で実行する

```
julia findnodes.jl
```

    重たい計算をしたい時やクラスターやスパコンで使いたい時
    複数の計算を投げたい時

Julia言語の使い方

大きく分けて三種類の方法がある

1. アプリを起動する（ダブルクリックで開く）

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.8.5 (2023-01-08)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

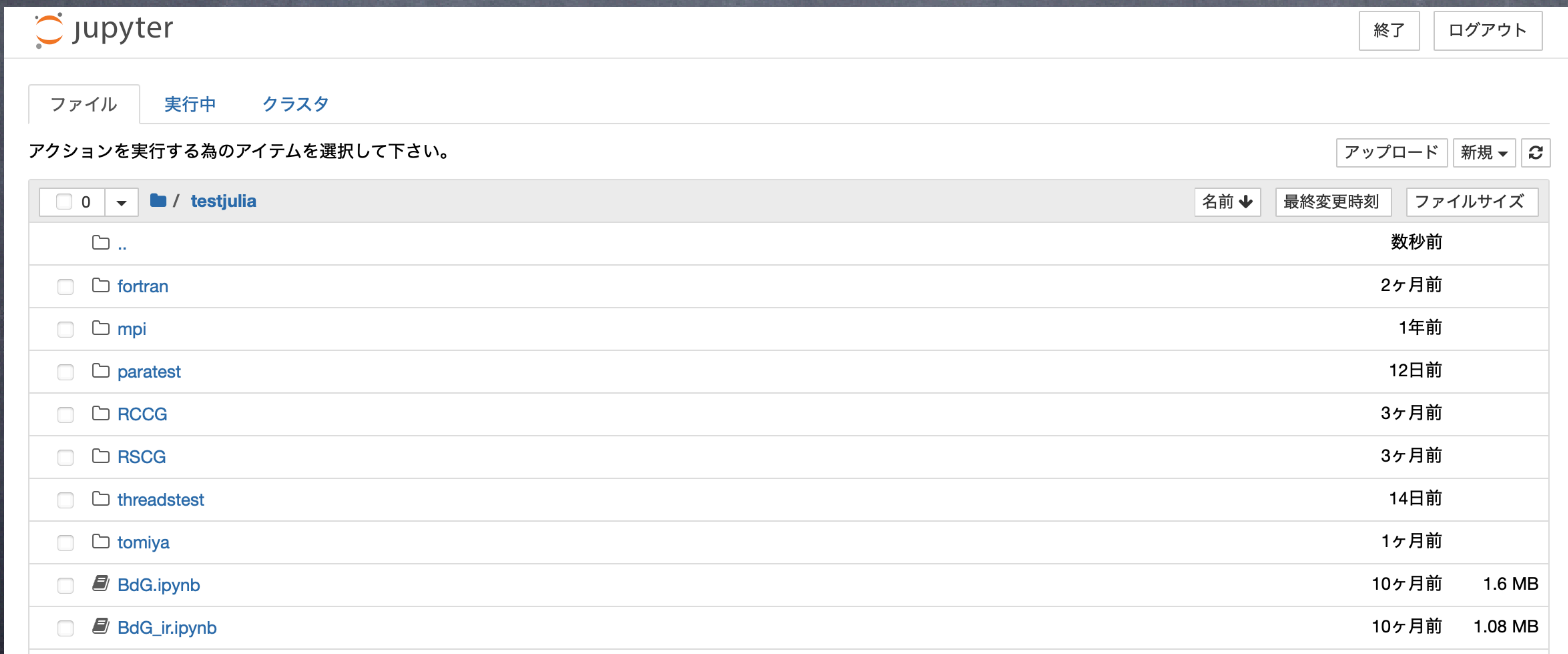
REPL(Read Eval Print Loop)と呼ばれている

ファイルに書いたコードも呼べる

Julia言語の使い方

大きく分けて三種類の方法がある

2. Jupyter notebookを使う　ブラウザで使える。Pythonのnotebookと似た操作感。Mathematicaに似ている

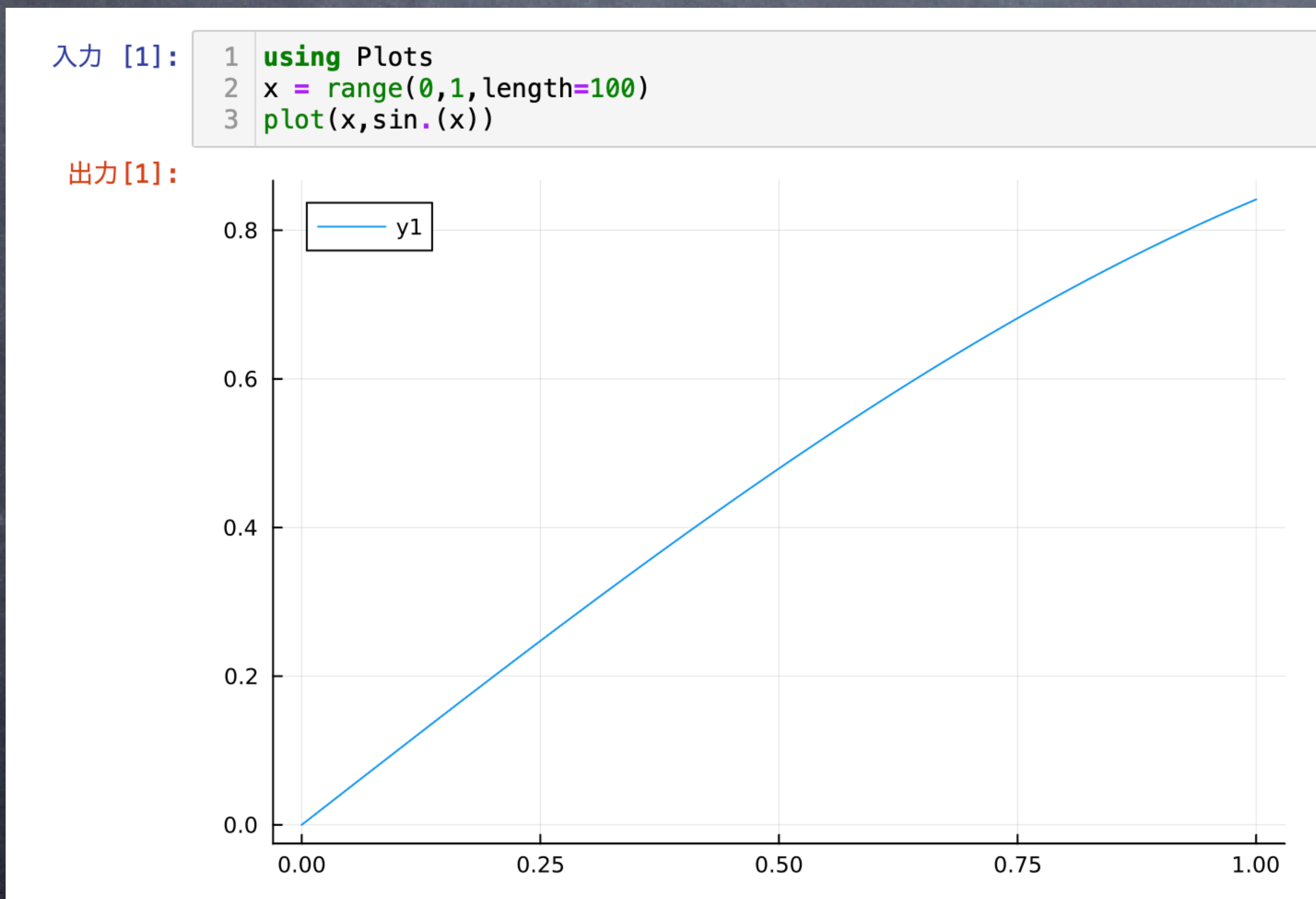


Julia言語の使い方

大きく分けて三種類の方法がある

2. Jupyter notebookを使う

ブラウザで使える。Pythonのnotebookと似た操作感。Mathematicaに似ている



入力をしてshift + enter
で結果が下に表示される

数式も扱えるので、一つの
計算ノートが出来上がる

Julia言語の使い方

大きく分けて三種類の方法がある

3. 端末で実行する

```
julia findnodes.jl
```

重たい計算をしたい時やクラスターやスパコンで使いたい時

```
julia findnodes.jl > output.txt
```

FortranやCのようなコンパイル作業がいらない

どの方法でやっても同じ。
“ある（誰かが作った）コードを走らせたいのに、コンパイルが通らなくて動かせない” ということがない

Juliaの文法

詳しい事は

数値計算に特化した

Juliaの解説書

量子力学や統計力学等の問題を解く

```
julia> println("hello world!")
hello world!

julia> 1 + 2
3

julia> 2 * 4
8

julia> (1+2im)*(1-2im)
5 + 0im

julia> exp(im*π)
-1.0 + 1.2246467991473532e-16im

julia> exp(im*π^2)
-0.9026853619330714 - 0.4303012170000917im
```

電卓的に使える

虚数はim

2*imじゃなくて2imで良い

三角関数や指数関数はそのまま使える

円周率はπをそのまま使える

Juliaの文法

詳しい事は

数値計算に特化した

Juliaの解説書

量子力学や統計力学等の問題を解く

```
julia> a = "沖縄といえば"
"沖縄といえば"

julia> b = "ルートビア"
"ルートビア"

julia> a*b
"沖縄といえばルートビア"
```

文字列は非可換なので+じゃなくて*

```
julia> a = [
       1
       im]
2-element Vector{Complex{Int64}}:
 1 + 0im
 0 + 1im

julia> a'*a
2 + 0im
```

aの転置複素共役はa'

```
julia> A =
       [ 1 2
         3 4 ]

2x2 Matrix{Int64}:
 1  2
 3  4
```

行列の定義

```
julia> B = [
       1
       2]
2-element Vector{Int64}:
 1
 2
```

ベクトルの定義

```
julia> A*B
2-element Vector{Int64}:
  5
 11
```

行列とベクトルの積

Juliaの文法

詳しい事は

数値計算に特化した

Juliaの解説書

量子力学や統計力学等の問題を解く

物理屋にとって良い点：ギリシャ文字が使える

シンプルな関数の定義

functionとして関数を定義してもよい

```
1     α(x) = x^2
2   ∨ function β(x,y)
3         return x + y
4     end
5     x = 2
6     y = 4
7     δ = α(x) + β(x,y)
```

線形代数も特殊関数もシンプルに使える

```
using LinearAlgebra
A = rand(10,10) #10x10行列の各要素に乱数が入った行列を定義
B = A'*A #行列同士の積
e,v = eigen(B) #固有値計算
u, s, v = svd(B) #特異値分解
using SpecialFunctions
n = 3
x = 0.4
Jnx = besselj(n,x) #n次のベッセル関数
ψ = digamma(x) #ディガンマ関数
```

行列AのエルミートIn共役はA'で計算できる

Julia言語の特徴：速さ

Juliaは、”コンパイル”という作業をユーザーで行わない Pythonのようなスクリプト言語みたい

書いてすぐ実行という手軽さが実現

Juliaは型を指定しない

```
1    α(x) = x^2
2  ∨ function β(x,y)
3        return x + y
4    end
5    x = 2
6    y = 4
7    δ = α(x) + β(x,y)
```

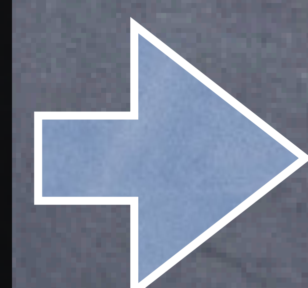
型を指定する煩雑さから解放されている

しかし、FortranやC言語に匹敵する速さ

実行時コンパイル(JITコンパイル)という技術を採用しているため

初めて関数が呼ばれた時、動的にコンパイルが実行されている

```
julia> function test(x)
         a = 1+2
         y = 3*x
         return y+a
       end
```



```
julia> @code_typed test(2)
CodeInfo(
1 ─ %1 = Base.mul_int(3, x)::Int64
│   %2 = Base.add_int(%1, 3)::Int64
└──      return %2
) => Int64
```

1. 型の推論
2. 高度な最適化
3. コンパイル

1.整数が入り整数が返ることが特定されている

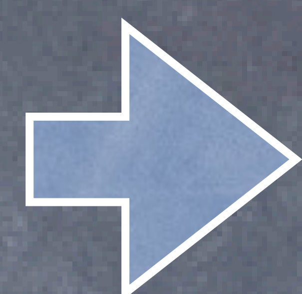
2. a=1+2はこの時点で3が代入されているし、aという変数は代入で消去されている

Julia言語の特徴：速さ

初めて関数が呼ばれた時、動的にコンパイルが実行されている

```
julia> function test(x)
         a = 1+2
         y = 3*x
         return y+a
       end
```

test(2)



```
julia> @code_typed test(2)
CodeInfo(
1 ─ %1 = Base.mul_int(3, x)::Int64
│   %2 = Base.add_int(%1, 3)::Int64
└──      return %2
) => Int64
```

test(2.1)

```
julia> @code_typed test(2.1)
CodeInfo(
1 ─ %1 = Base.mul_float(3.0, x)::Float64
│   %2 = Base.add_float(%1, 3.0)::Float64
└──      return %2
) => Float64
```

1. 型の推論
2. 高度な最適化
3. コンパイル

引数が2.1と整数ではないので、倍精度実数用のコードが作られる。倍精度用の和と積が用意されている

引数にどんな型が入っているかで、その都度最適なコードがコンパイルして生成される

Juliaの特徴：パッケージ管理システム

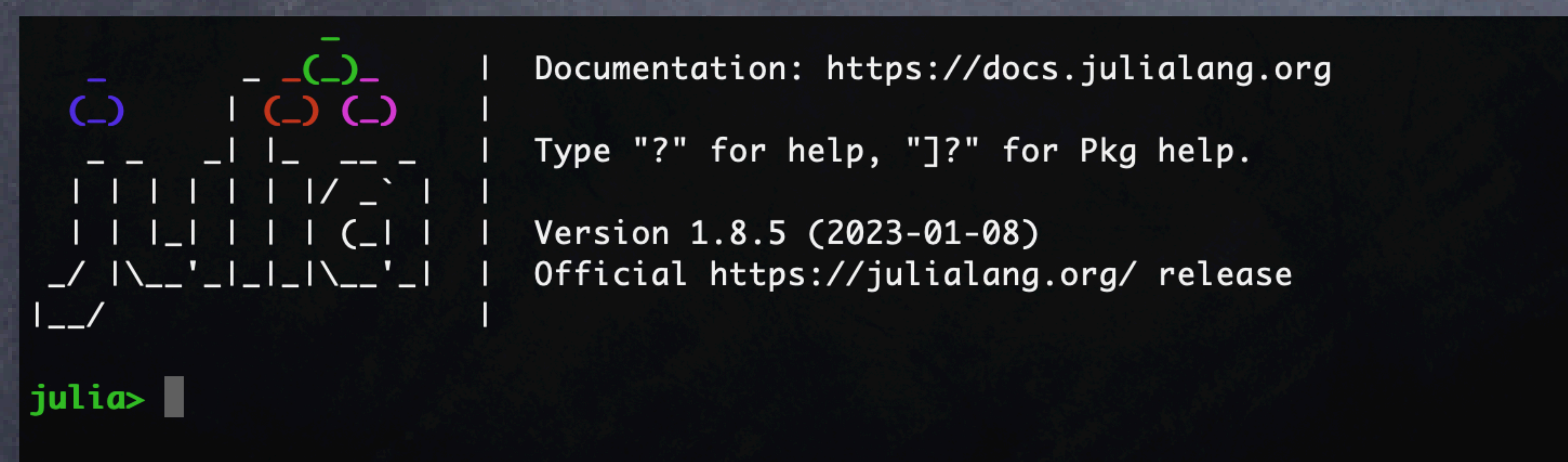
他の人が作ってくれているライブラリを簡単に導入することができる

FotranやC:自分の環境でコンパイルできるかどうかがわからない

それを動かすために別のライブラリが必要かもしれない

そもそもライブラリが置かれている共通スペースがない

WindowsでもMacでもLinuxでも同じようにインストールできる



REPL上で]キーを押す

add Flux

などとする

依存パッケージも含めて自動的にインストールされる

数値計算を行う上で、他の人が開発したパッケージを使えるというのは極めて重要

「車輪の再発明」を避ける

他の人が動作検証している（はず）なので安心して使える

Juliaの特徴：パッケージ管理システム

例：SparseArrays.jl

疎行列（要素がほとんどゼロの行列）を簡単に扱える

```
using SparseArrays
using LinearAlgebra
function make_T(M)
    T = spzeros(Float64,M-1,M-1)
    for i=1:M-1
        j = i -1
        if 1 <= j <= M-1
            T[i,j] = 1
        end
        j = i + 1
        if 1 <= j <= M-1
            T[i,j] = 1
        end
        T[i,i] = -2
    end
    return T
end
```

SpecialFunctions.jl

様々な特殊関数が扱える

```
using LinearAlgebra
A = rand(10,10) #10x10行列の各要素に乱数が入った行列を定義
B = A'*A #行列同士の積
e,v = eigen(B) #固有値計算
u, s, v = svd(B) #特異値分解
using SpecialFunctions
n = 3
x = 0.4
Jnx = besselj(n,x) #n次のベッセル関数
ψ = digamma(x) #ディガンマ関数
```

QuadGK.jl

1次元数値積分ができる

```
using QuadGK
function orthoij(n,i,j)
    αi = besselj_zero(n,i)
    αj = besselj_zero(n,j)
    f(x) = x*besselj(0,αi*x)*besselj(0,αj*x)
    dij,err = quadgk(f,0,1)
    return dij
end
```

ベッセル関数の直交性の確認

TightBinding.jl

自作。公式ライブラリ化

```
using TightBinding
Ax = 1
Ay = 1
m2x = 1
m2y = m2x
m0 = -2*m2x
m(k) = m0 + 2m2x*(1-cos(k[1]))+2m2y*(1-cos(k[2]))
Hk(k)
```

```
using
using
nkx =
kxs =
mat_e
for i=
    kx
    ma
    #p
    e,
    #p
    ma
end
plot(kxs,mat_e,labels="")
savefig("tes1.png")
```

Juliaの特徴：パッケージ管理システム

"Juliaって新しい言語だからパッケージがPythonより少ないんでしょ…"

大丈夫！Pythonのコードも簡単に呼べるよ

Pythonのライブラリ、TensorFlowのKerasを使ってニューラルネットワークを定義してみる

```
using PyCall
using Plots
const tensorflow = pyimport("tensorflow")
const keras =tensorflow.keras
const layers =keras.layers
const optimizers =keras.optimizers

function build_model(d_input,d_middle)
    inputs = keras.Input(shape=(d_input,))
    x = layers.Dense(d_middle, activation="relu")(inputs)
    y = layers.Dense(1)(x)
    adam = optimizers.Adam()
    model =  keras.Model(inputs=inputs, outputs=y)
    model.compile(optimizer=adam,
                        loss="mean_squared_error")
    return model
end
```

Juliaで作った配列をそのまま扱える

numpy配列としてインプットされる

c言語も呼べるしFortranも呼べる

JAEA CCSE RIKEN AIP

Julia言語の特徴:多重ディスパッチ

オブジェクト指向言語ではないが、ほとんど等価なことが可能

多重ディスパッチという機能で実装されている

FORTRAN,C	オブジェクト指向 C++, Python等	多重ディスパッチ
犬が鳴く(10回)	犬.鳴く(10回)	鳴く(犬,10回)
蛙が鳴く(3回)	蛙.鳴く(3回)	鳴く(蛙,3回)
異なる関数を呼ぶ必要がある	一つ目の引数を特別扱い	一つ目と二つ目に何が入るかで挙動が変わる

ユーザーは、犬か蛙かを気にせずに、「鳴く」を選べば良い

Julia言語の特徴：多重ディスパッチ

オブジェクト指向

“動物”クラスに
「鳴く」を定義

動物クラスを継承した
“犬”クラスで「鳴く」
をオーバーライド:
ワンワン！

動物クラスを継承した
“蛙”クラスで「鳴く」
をオーバーライド:
ケロケロ！

多重ディスパッチ

クラスの代わりに「型」を使う

“動物”という抽象型を定義

“動物”を引数とする「鳴く」を定義

鳴く(a::動物,回数)
を定義

動物型の下部タイプで
ある「犬」型を定義

鳴く(a::犬,回数)
ワンワン！ を定義

動物型の下部タイプで
ある「蛙」型を定義

鳴く(a::蛙,回数)
ケロケロ！ を定義

犬と蛙が同時に鳴く場合のみゲゲゲワンと鳴く
シチュエーションの場合は？

鳴く(a::犬,b::蛙,回数)
を定義

Juliaでのライブコーディング

万有引力に従う天体の運動

2つの物体同士に働く万有引力

$$m\vec{a} = \vec{F} \qquad \vec{F} = -\frac{GMm}{r^3}\vec{r}$$

2本の微分方程式となる

$$\frac{d\vec{v}}{dt} = -\frac{GM}{r^3}\vec{r} \qquad \frac{d\vec{r}}{dt} = \vec{v}$$

これを解けば2つの物体の運動がわかる

これをJuliaでコーディングしてみる

無次元化しておく

$$\dot{x} = u, \dot{u} = -\frac{x}{|\boldsymbol{r}|^3}$$

$$\dot{y} = v, \dot{v} = -\frac{y}{|\boldsymbol{r}|^3}$$

時間を離散化して、各時刻での天体の位置をシミュレートする

簡単のため、z方向には動かないとする

原点に天体がいて、その周りを回っている

$$\dot{x} = u, \dot{u} = \Phi(x)$$

時間に関する微分方程式を逐次的に解くことにする

エネルギーが保存するような手法を使ってみる

-> リープフロッグ法を採用

万有引力に従う天体の運動

時間に関する微分方程式を逐次的に解くことにする

$$\dot{x} = u, \dot{u} = -\frac{x}{|\boldsymbol{r}|^3}$$

$$\dot{y} = v, \dot{v} = -\frac{y}{|\boldsymbol{r}|^3}$$

$$\dot{\boldsymbol{x}} = \boldsymbol{u}, \dot{\boldsymbol{u}} = \Phi(\boldsymbol{x})$$

$$E = \frac{1}{2}(u^2 + v^2) - \frac{1}{r}$$

エネルギーが保存するような手法を使ってみる

-> リープフロッグ法を採用

リープフロッグ法：　速度と位置を交互に更新する

$$u(t + \Delta t/2) = u(t - \Delta/2) + \Phi(x)\Delta t$$

$$x(t + \Delta t) = x(t) + u(t + \Delta t/2)\Delta t$$

時刻tでの運動量は

$$u(t) = (u(t - \Delta/2) + u(t + \Delta/2))/2$$

最初に必要なt=-Δt/2の時のuは

$$u(-\Delta t/2) = u(0) - \Phi(x)\Delta t/2$$

Juliaでのライブコーディング

万有引力に従う天体の運動

nprint:何回に1回書き出すか

u(t): u

u(t+Δt/2): u_p

u(t-Δt/2): u_m

v(t): v

v(t+Δt/2): v_p

v(t-Δt/2): v_m

Δt = tmax/n

n:離散点の数

tmax:最終時刻

E0: 最初のエネルギー

E: 最初のエネルギー

$$\dot{x} = u, \dot{u} = -\frac{x}{|\boldsymbol{r}|^3}$$

$$\dot{y} = v, \dot{v} = -\frac{y}{|\boldsymbol{r}|^3}$$

$$u(t+\Delta t/2) = u(t-\Delta/2) + \Phi(x)\Delta t$$

$$x(t+\Delta t) = x(t) + u(t+\Delta t/2)\Delta t$$

$$u(t) = (u(t-\Delta/2) + u(t+\Delta/2))/2$$

$$E = \frac{u^2+v^2}{2} - \frac{1}{\sqrt{x^2+y^2}}$$

$$u(-\Delta t/2) = u(0) - \Phi(x)\Delta t/2$$

アニメーションのためのパーツ

```
using Plots
ENV["GKSwstype"] = "nul"
anim = Animation()
```

```
plt = scatter([0,x],[0,y],label="t = $t",xlims=(-2,2),ylims=(-2,2),aspect_ratio = 1)
frame(anim,plt)
```

```
gif(anim, filename, fps = 30)
```

初期パラメータ

```
x = 1.0
y = 0.0
u = 0.0
v = 1.0
n = 10000
tmax = 20
Δt = tmax/n
t = 0.0
filename = "anim_v$v.gif"
nprint = 50
```

これらの材料でライブコーディング

3体問題：太陽、地球、月

それぞれが万有引力に従う

-> r1,r2,r3,v1,v2,v3みたいにすると大変

$$F_i = -G \sum_j \frac{m_i m_j}{|\vec{r}_i - \vec{r}_j|^2}$$

Juliaっぽく書くには？

例えば

```
struct Body
    r::Vector{Float64}
    v::Vector{Float64}
    f::Vector{Float64}
    mass::Float64
    function Body(r,v,mass)
        f = zero(r)
        return new(r,v,f,mass)
    end
end
```

天体の情報を格納する型

```
mutable struct SolarSystem
    bodies::Vector{Body}
    nbody::Int64
    function SolarSystem()
        bodies = Body[]
        nbody = 0
        return new(bodies,nbody)
    end
end
```

太陽系全体の情報を格納する型

これらを使って微分方程式を解く

3体問題：太陽、地球、月

それぞれが万有引力に従う

-> r1,r2,r3,v1,v2,v3みたいにすると大変

$$F_i = -G\sum_j \frac{m_i m_j}{|\vec{r}_i - \vec{r}_j|^2}$$

Juliaっぽく書くには？

標準のpush!をSolarSystemに対応させる

```
function Base.push!(s::SolarSystem,body::Body)
    push!(s.bodies,body)
    s.nbody = length(s.bodies)
end
```

```
function Base.getindex(s::SolarSystem,i)
    return s.bodies[i]
end
```

s[1]やs[2]で天体の情報を取り出せるようにする

多重ディスパッチの利用の例

```
function calc_force!(s::SolarSystem)
    nbody = s.nbody
    for i=1:nbody
        s[i].f .= 0
    end
    for i=1:nbody
        r_i = s[i].r
        mass_i = s[i].mass
        for j=i+1:nbody
            r_j = s[j].r
            mass_j = s[j].mass
            rij = r_i - r_j
            s[i].f .+= -G*mass_i*mass_j*rij/norm(rij)^3
            s[j].f .+= -s[i].f
        end
    end
end
```

万有引力の計算

3体問題：太陽、地球、月

それぞれが万有引力に従う

-> r1,r2,r3,v1,v2,v3みたいにすると大変

$$F_i = -G \sum_j \frac{m_i m_j}{|\vec{r}_i - \vec{r}_j|^2}$$

Juliaっぽく書くには？

```
function update!(s::SolarSystem,Δt)
    nbody = s.nbody
    for i=1:nbody
        mass = s[i].mass
        s[i].v .+= s[i].f*Δt/(2*mass)
        s[i].r .+= s[i].v*Δt
    end

    calc_force!(s)

    for i=1:nbody
        mass = s[i].mass
        s[i].v .+= s[i].f*Δt/(2*mass)
    end

end
```

速度ベレル法で時間発展

```
function calc_force!(s::SolarSystem)
    nbody = s.nbody
    for i=1:nbody
        s[i].f .= 0
    end
    for i=1:nbody
        r_i = s[i].r
        mass_i = s[i].mass
        for j=i+1:nbody
            r_j = s[j].r
            mass_j = s[j].mass
            rij = r_i - r_j
            s[i].f .+= -G*mass_i*mass_j*rij/norm(rij)^3
            s[j].f .+= -s[i].f
        end
    end
end
```

万有引力の計算

3体問題：太陽、地球、月

それぞれが万有引力に従う

-> r1,r2,r3,v1,v2,v3みたいにすると大変

$$F_i = -G \sum_j \frac{m_i m_j}{|\vec{r}_i - \vec{r}_j|^2}$$

Juliaっぽく書くには？

```
solarsystem = SolarSystem()
sun = Body([0,0],[0,0],mass_Sun )
push!(solarsystem,sun)

r_Earth = [radius_Earth,0]
v_Earth = [0,velocity_Earth]
earth = Body(r_Earth,v_Earth ,mass_Earth )
push!(solarsystem,earth)

r_Moon = [radius_Earth+radius_Moon ,0]
v_Moon = [0,velocity_Earth+velocity_Moon]
moon = Body(r_Moon,v_Moon ,mass_Moon )
push!(solarsystem,moon)
```

初期速度や位置を入れる

push!で太陽系に天体を追加している

```
    for i = 2:n
        update!(solarsystem,Δt)
        t += Δt
    end
```

本体コード　（可視化部分は省略）

この方法だと、pushで別の天体をいくらでも追加できる

天体を追加しても本体コードの変更はない

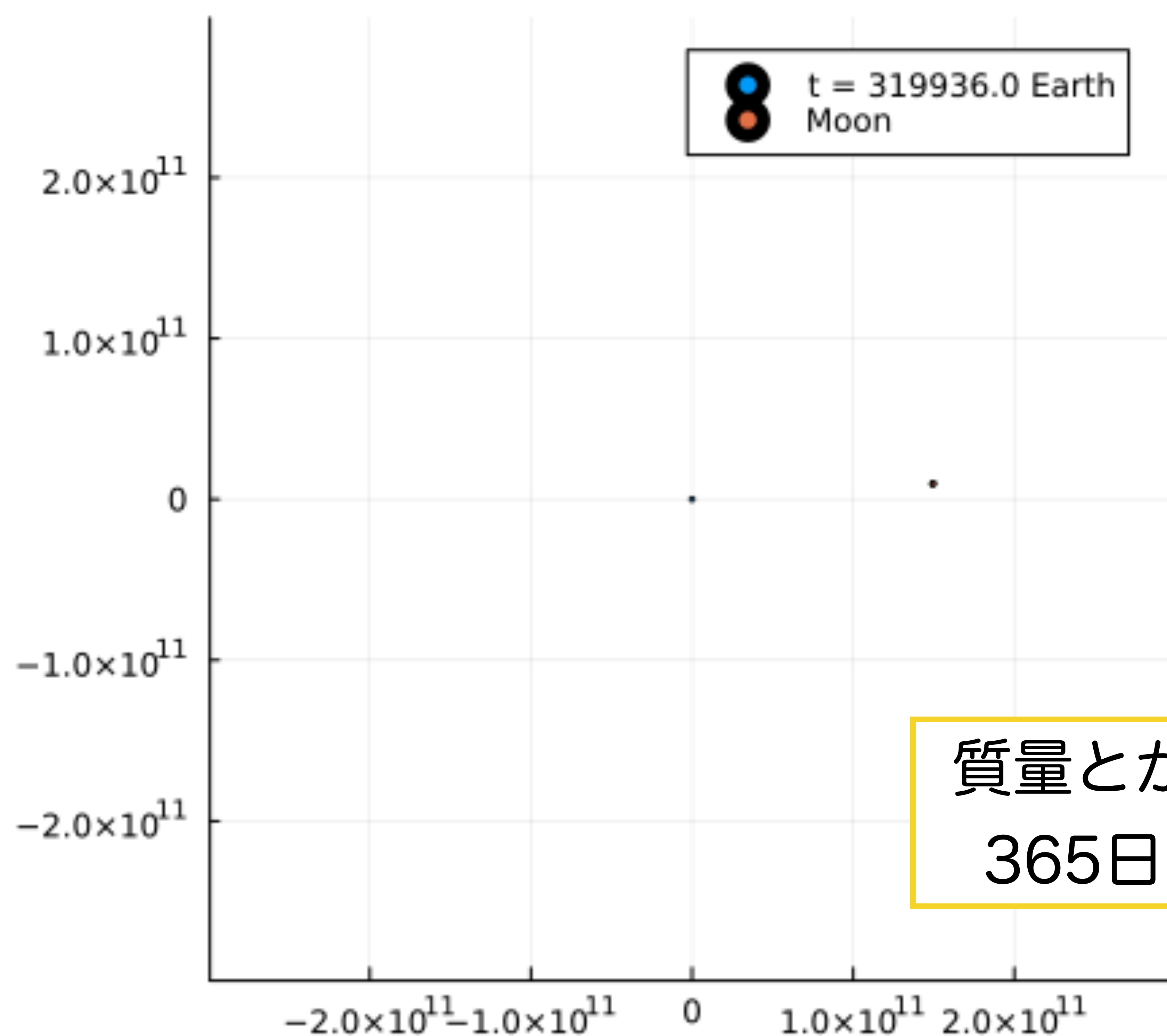
3体問題：太陽、地球、月

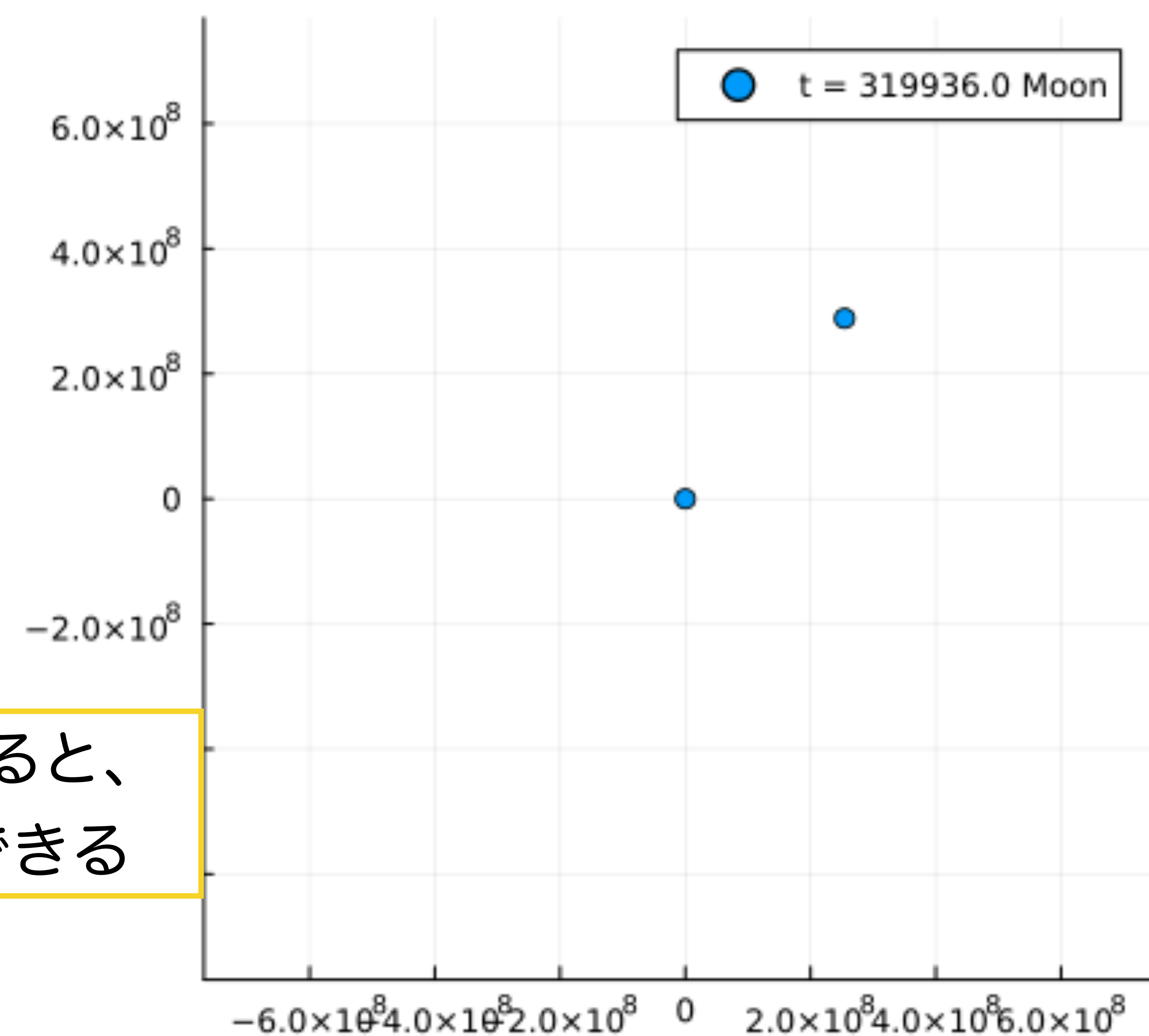
それぞれが万有引力に従う

$$F_i = -G \sum_j \frac{m_i m_j}{|\vec{r}_i - \vec{r}_j|^2}$$

-> r1,r2,r3,v1,v2,v3みたいにすると大変

Juliaっぽく書くには？





質量とか速度をちゃんと入れると、
365日で一周するのを確認できる

まとめ

Julia言語とは

- 数値計算に最適な言語
    - Pythonのようなシンプルさ
    - C,Fortranに匹敵する速さ
    - matlabのような線形代数操作
    - Mathematicaのように積分したりできる

新しい言語なので、数値計算における”痒いところに手が届く”ような言語設計になっている

この春の学校中、Juliaに関する質問はいつでも受け付けます

Julia 研究会
「数学と物理におけるJuliaの活用」
開催決定!

アカデミア内外でのJuliaの利用を議論します!

- 開催時期: 7月前半
- 開催場所: 九州大学伊都キャンパス
- 世話人: 富谷昭夫 (大阪国際工科専門職大学)、横山俊一 (東京都立大学)、 永井佑紀 (日本原子力研究開発機構) (順不同)
- ホームページ: https://akio-tomiya.github.io/julia_imi_workshop2023/
- 主催: 九州大学マス・フォア・インダストリ研究所
  ※ IMI 共同利用・研究集会 (II) として実施
- 共催: 科学研究費補助金学術変革領域研究(A)「学習物理学の創成」
  科研基盤研究(C)「Julia言語を用いた新しい計算機数論システムの開発とその応用」

Julia言語とは

数値計算に最適な言語

- Pythonのようなシンプルさ
- C,Fortranに匹敵する速さ
- matlabのような線形代数操作
- Mathematicaのように積分したりできる

数値計算しようぜ

新しい言語なので、数値計算における”痒いところに手が届く”ような言語設計になっている

この春の学校中、Juliaに関する質問はいつでも受け付けます