

# Juliaによる科学技術計算: 大規模並列計算について

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

永井佑紀

# アウトライン

- 自分野(物性物理学分野)におけるコード開発の状況：みんな大好きFortran
- 大規模並列計算
- 機械学習
- まとめ

# 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

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：機械学習を物理学を使う

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

# 自分野(物性物理学分野)に おけるコード開発の状況

みんな大好きFortran

# 物性物理学とは

物質の性質を理論的に研究する分野：固体における電子状態の研究が多い

量子力学と統計力学、場の量子論などを駆使する

超伝導体、磁性体、半導体、量子コンピュータetc

現実に存在しうるあらゆるもののが対象

more is different (多は異なり)

フィリップ・アンダーソン(1977年、ノーベル物理学賞)

物質中には電子が沢山いる->相互作用によって新しい性質が創発される (超伝導)

電子の数は $10^{23}$ 個 ->そもそもまとめて計算できない

いかに本質を抽出して計算可能にするかが大事

多彩すぎるため「一つの良い共通コード」がない

# 使われている言語

計算物理学：計算機とともに進展してきた分野      歴史が古い

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！

## 数値計算でいまだに使われているプログラミング言語

Fortran90がほとんど。FORTRAN77もたまに見かける  
大学学部のプログラミング実習でFortranを習う方も多い？

(汎用化できる領域での)汎用パッケージの多くはFortranで書かれている

例：密度汎関数理論に基づく第一原理計算パッケージ

Quantum Espresso      原子の数と種類と位置が与えられれば物性が計算できる

VASP

WIEN2k

個別の研究でもFortranが多く使われている

# Fortranが使われている実例

少し古いが C. A. Perroni et al., Phys. Rev. B, 75, 045125 (2007)

“We employ the routine MINIMIZE provided in Ref. 1.”

[1] A. Georges et al., Rev. Mod. Phys. 68, 13 (1996)

動的平均場理論(DMFT)のレビュー論文。FORTRANコードが公開されている

## MINIMIZEのコメント欄

```
C=====+$  
C PROGRAM: minimize  
C TYPE    : subroutine  
C PURPOSE: conjugent gradient search  
C I/O     :  
C VERSION: 30-Sep-95  
C COMMENT: This is a most reliable conjugent gradient routine! It has  
C           served us well for many years, and is capable to cope with  
C           a very large number of variables. Unfortunately, we don't  
C           know who wrote this routine (original name: 'va10a'), and  
C           we find it very obscure.  
C           Don't worry, it works just fine.  
Cnoprint=====
```

(意訳)”このコードは長年使つ  
てきた。残念ながら誰が書い  
たかわからない。でも気にし  
ないで、ちゃんと動くよ”

調べてみた

R.Fletcher, "FORTRAN SUBROUTINES FOR MINIMIZATION BY QUASI-  
NEWTON METHODS", Research group report(United Kingdom Atomic Energy  
Authority)

1972年4月のコード

# この分野での最近の流れ

物性理論分野

ここ10年でC++の公開コードが増えてきた

小さめのコードはPythonで書かれている場合も

近年流行している機械学習+物理学ではFortranではきつい

機械学習系ライブラリはPythonが優位

少数でコードを書いて研究するグループは公開せずにFortranを使っている?

MPI並列計算する場合にはFortranはやはり便利

# なぜFortranが(まだ)使われているのか

数百から数万のCPUコアを使ったMPI+Openmpのハイブリッド並列計算をスーパーコンピュータで行いたい場合

スパコンごとに特別にチューニングされたコンパイラ：FortranとC++しかない

(C++11以降であればまだ良いが)C++よりFortranの方が配列の扱いがしやすい

キャッシュを有効活用し、MPI通信の待ち時間も利用したカリッカリのチューニングが可能

もちろんC++でも可能だが、スパコンではFortran人口が多い

C++と比べるとFortranの方が学習障壁が低い（個人の感想です）

数値計算ではオブジェクト指向はなくても良い

プログラミングよりも物理に着目したい

Fortranは数式に近い書き方ができる

先生がFortran(FORTRAN)使い

研究室秘伝のコードがFortran

最初の約束事さえ覚えればすぐに研究に取り掛かれる（と予想される）

色々な理由があるが、結局「速い」が重要

「適当に書いても速い」もポイント

# Fortranに思うところ

確かに速い

基本的には全てのパートを自分で作らなければならない

LAPACKとBLASが使えるので固有値計算などは”簡単”にできる

例

“2次元シュレーディンガー方程式を円筒座標系で解きたいからBessel関数を使って基底を展開して、Bessel関数の零点を使って境界条件を設定して対角化して固有値を求める”

「Bessel関数 $J_n(x)$ はどうやって計算するの？」

「LAPACKはどうやって使うの？」

“動的平均場理論で松原振動数表示のGreen関数を再現するような有効ハミルトニアンのパラメータを決めるために、非線形関数を最小化したい”

「非線形最小化問題ってどうやって解くの？」

「非線形最小化問題を解くコードを書きたいんじゃなくて物理の問題を解きたいのに」

「持ってきたコードがFORTRANだけど自分のFortranコードとうまくリンクできない」

スパコンやクラスター計算機で並列計算をするならFortranは使う価値がある

でも、もう少し「軽い」問題なら？

JuliaならPython規模からFortran規模までカバーできる

# 大規模並列計算

# 大規模並列計算

並列計算とは

一度に複数の計算を行うこと

昨今のCPUであればマルチコアが当たり前

同時に様々なタスクが走っている

大規模並列計算とは

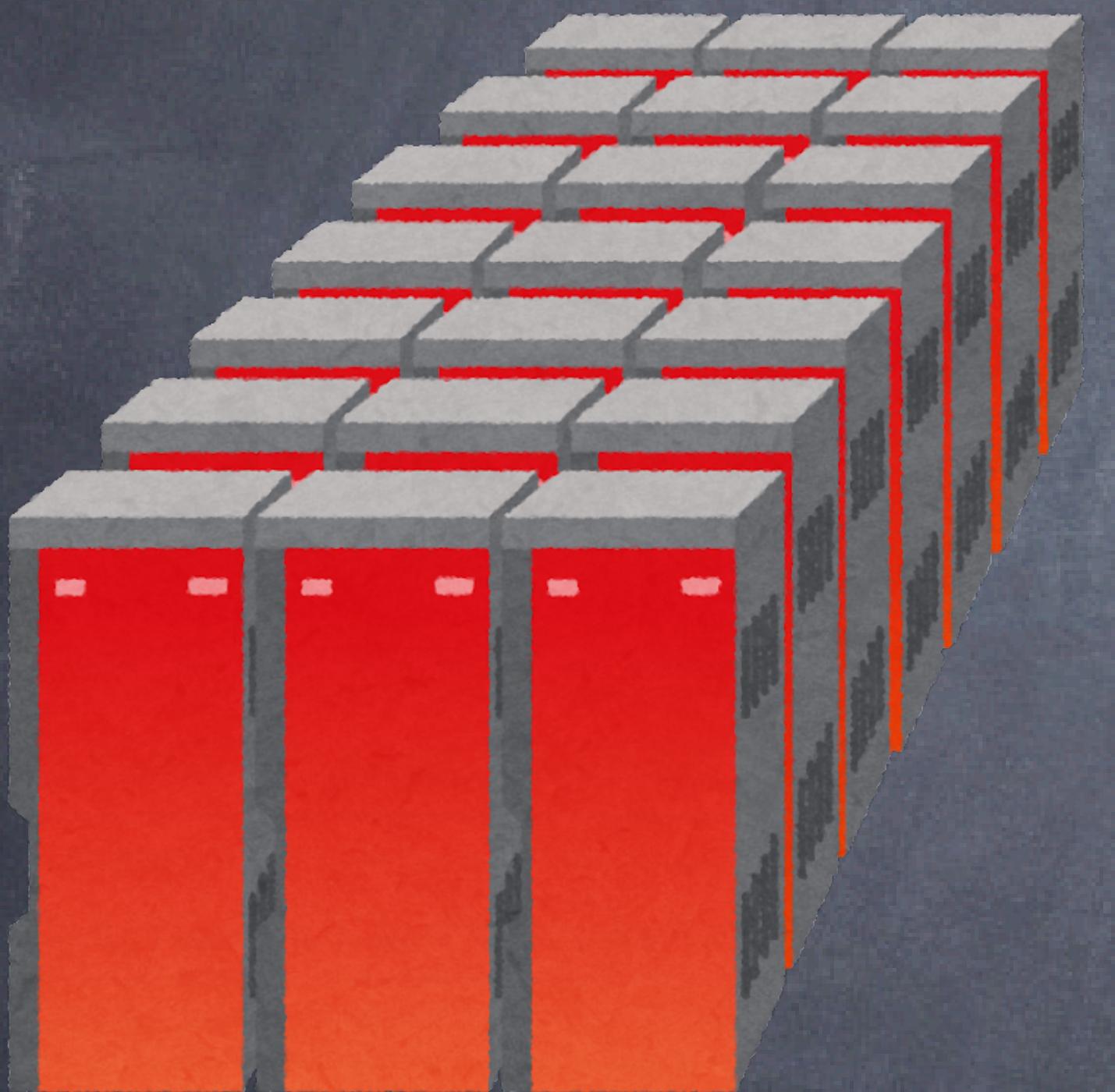
一度に大量の計算を行うこと

科学技術計算分野では

同時に”似た”計算を大量に並列に行う

非常に大量の計算を行なっているので、少しのロスも大きなロスになる

大規模並列計算特有の問題がある



# 並列計算の種類

大きく分けて二種類ある：スレッド並列とプロセス並列

スレッド並列



手が多い計算  
共通の脳（メモリ）  
にアクセスする

高速に行うには：  
いかに手がぶつからないよう  
に手を動かすか。脳を同時書  
き換えしないようにするか

メモリを共有する  
並列計算

手が多い計算

プロセス並列



人が多い計算



それぞれがそれぞれ  
の仕事をする

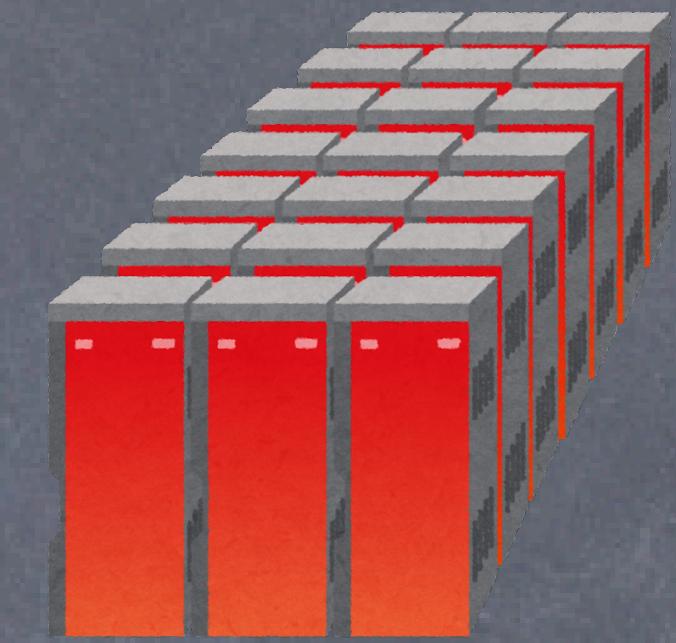
高速に行うには：  
それぞれのデータのやり取  
りは時間がかかるので、通  
信を減らす

メモリを共有しない  
並列計算

# 並列計算の種類

ハイブリッド並列

スレッド並列+プロセス並列



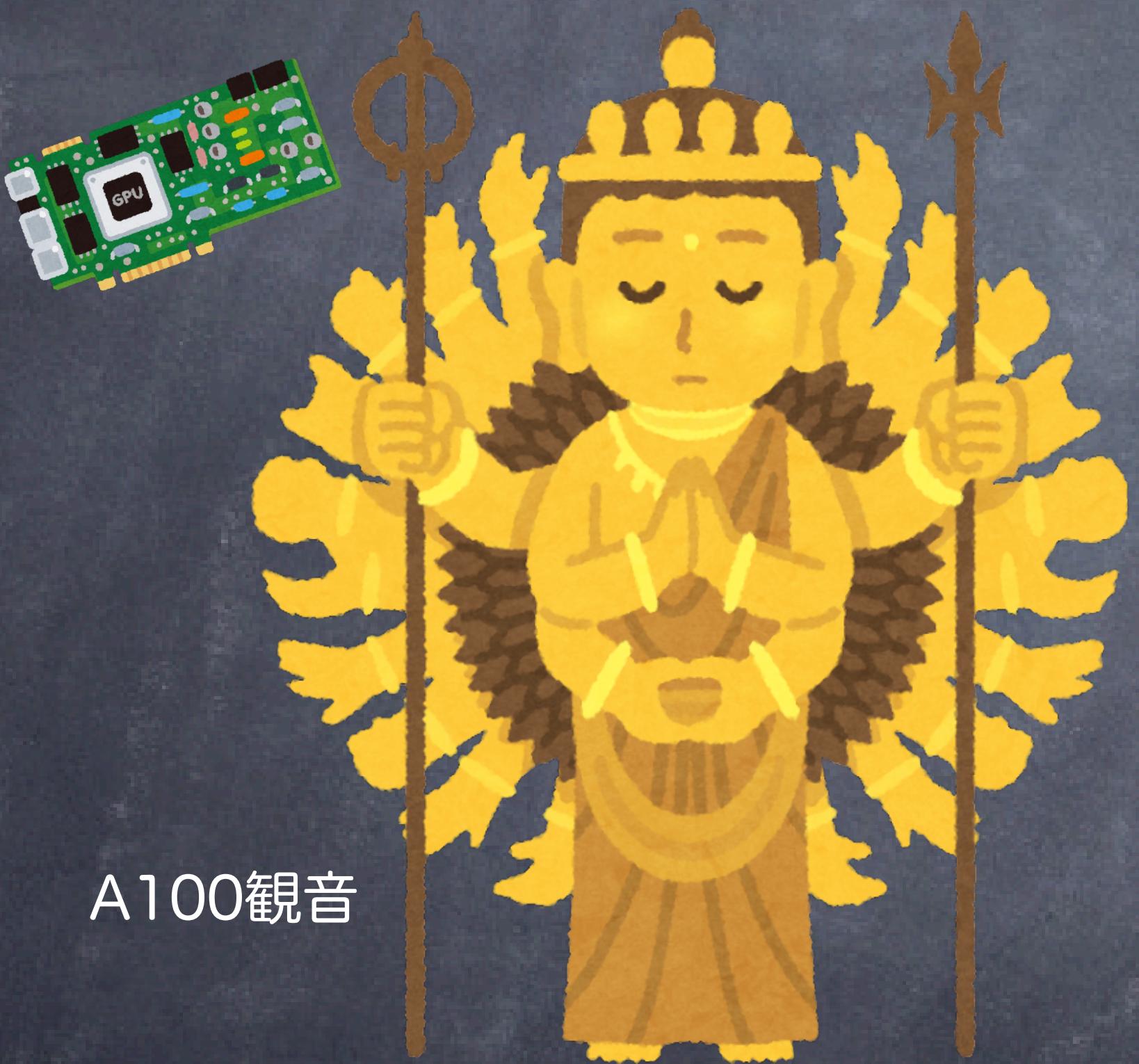
スーパーコンピュータ：  
複数の物理サーバーが高速なLAN  
ケーブルでつながっている

サーバー内：スレッド並列  
サーバー間：プロセス並列

最先端物理学研究：スパコン上でFortranやC++でハイブリッド並列計算が行われている

# その他、並列計算の種類

## GPU計算



A100觀音

GPU:シンプルな演算器  
がすごくたくさんある

## マルチGPU並列

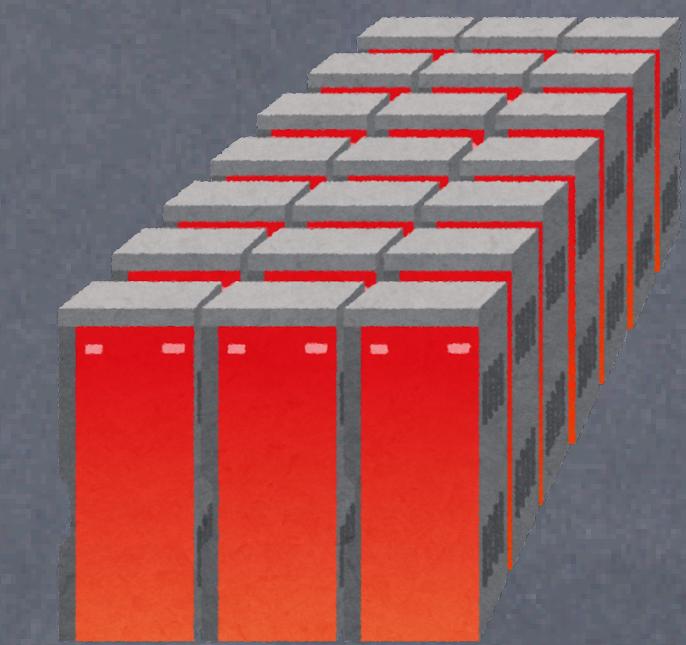


特定のタスクでは  
非常に強い

if文も含まない同じ計算の集まり  
(行列・ベクトル積等)

機械学習などと相性が良い

JuliaでマルチGPU: 気になるトピックスですが、  
自分もよくわかっていないので触れません



# Juliaでの並列計算

スレッド並列:後藤さんのtalk



物理の数値計算で「ハイパフォーマンスに」行う方法は正直よくわからない…

先行研究でOpenMPで出たパフォーマンスと同じくらい出したい

プロセス並列:このtalk



MPI.jlを使うのが無難

他言語のMPI使いはシンプルに移行できる

ハイブリッド並列:今のところJuliaで「ハイパフォーマンスに」行う方法を見つけられていない

->知りたい

# Juliaでのプロセス並列

Juliaでのプロセス並列の方法は複数種類ある

## 1. Distributed.jl

Juliaの標準のプロセス並列用パッケージ

単純で一つ一つが重たい並列計算ならこれが楽

各プロセスに投げたあと、通信はせず、後で結果  
を回収するような数値計算

-> pmapを使う

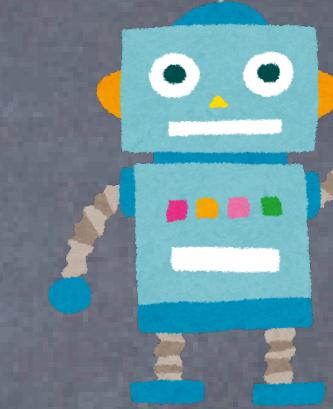
## 2. MPI.jl

科学技術計算において“業界標準”であるMPIを  
使えるパッケージ

スパコンが専用のMPIライブラリを持つ場合も使える

既存コードや論文と”同じように”Juliaで実装できる

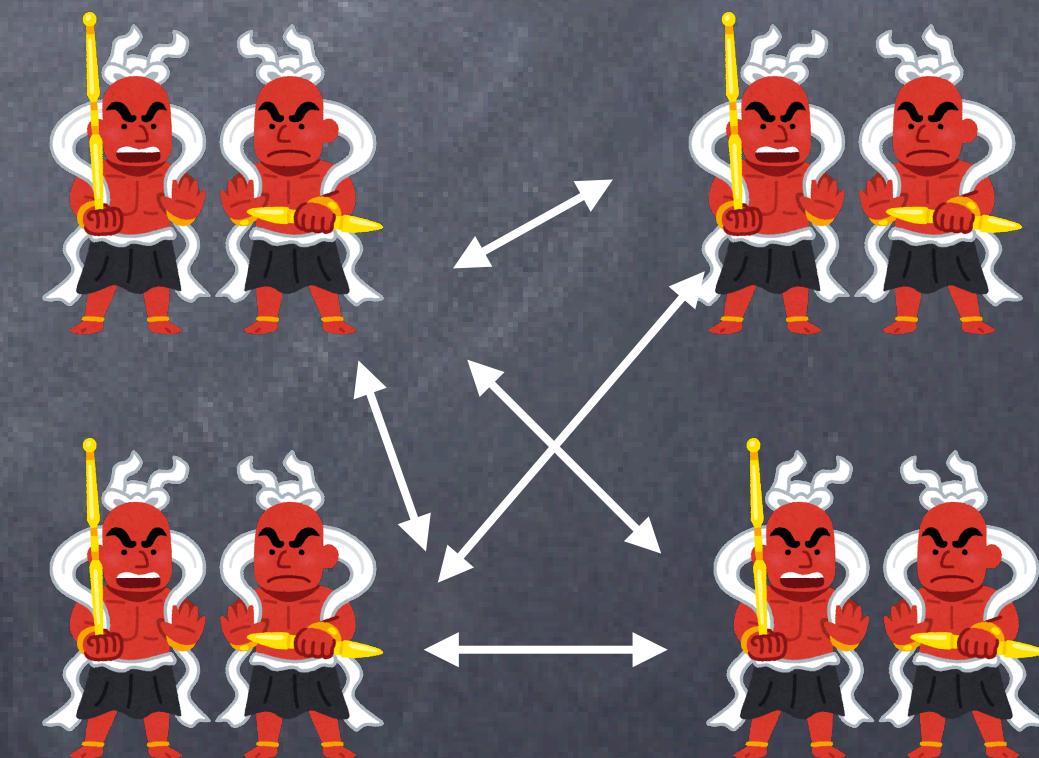
やれ！



はい！

masterとworkerの並列モデル

指示する人とやる人が別



同時に走り、それぞれを通信させる

# Distributed.jlが効果的な例

Krylov部分空間法を用いた手法（共役勾配法等）

Krylov部分空間  $K(A, x) : x, Ax, A^2x, A^3x, \dots$

初期ベクトル  $x$  が異なると異なる Krylov 部分空間ができる

例 :  $A x = b_i$  を  $i$  を変化させて解きたい

異なる  $b_i$  は別々に計算できる

`a= pmap(i -> goodfunc(i), 1:N)` のような形

`a` にはそれぞれの  $i$  で計算した値が代入される(要素数  $N$  の配列)

もし各  $i$  での計算が軽い場合は、スレッド並列の方が効率的（速い）

# 例：RSCG法(縮約シフト共役勾配法)

グリーン関数の行列要素 $G_{ii}$ は、 $e_i^T (z - H)^{-1} e_i$ で計算できるので、連立方程式

$(z-H)x = e_i$ さえ解ければよい

$e_i$ が異なるので異なる $G_{ii}$ はpmapで並列可能

Julia パッケージ RSCG.jl <https://github.com/cometscome/RSCG.jl>

**Table III.** Reduced-shifted CG for  $(\sigma_j I + A)x(\sigma_j) = b$  with a hermitian matrix  $A \in \mathbb{C}^{n \times n}$ ,  $V \in \mathbb{C}^{m \times n}$ ,  $x_k, r_k, p_k \in \mathbb{C}^n$ ,  $\alpha_k(\sigma_j), \beta_k(\sigma_j), \rho_k(\sigma_j), \Xi_k(\sigma_j)$ ,  $\Pi_k(\sigma_j), \Sigma_k \in \mathbb{C}^m$ , and  $\alpha_k, \beta_k \in \mathbb{C}$ .  $\Xi_k(\sigma_j) \equiv Vx_k(\sigma_j)$ .  $V^T \equiv (v_1, v_2, \dots, v_m)$ .  $v_i \in \mathbb{C}^n$ . Here, the symbol T represents transposition.

1. Input  $\sigma_j$  ( $j = 1, 2, \dots, N$ )
2. Set  $x_0 = 0, r_0 = p_0 = b, \alpha_{-1} = 1, \beta_{-1} = 0$
3. Compute  $\Sigma_0 = Vb$
4. Set  $\Xi_0(\sigma_j) = 0, \Pi_0(\sigma_j) = \Sigma_0, \rho_{-1}(\sigma_j) = \rho_0(\sigma_j) = 1$  ( $j = 1, 2, \dots, N$ )
5. For  $k = 0, 1, \dots$  until convergence Do:
  6.  $\alpha_k = (r_k, r_k)/(p_k, Ap_k)$
  7.  $x_{k+1} = x_k + \alpha_k p_k$
  8.  $r_{k+1} = r_k - \alpha_k Ap_k$
  9.  $\beta_k = (r_{k+1}, r_{k+1})/(r_k, r_k)$
  10.  $p_{k+1} = r_{k+1} + \beta_k p_k$
  11. Compute  $\Sigma_{k+1} = Vr_{k+1}$
  12. For  $j = 1, 2, \dots, N$  Do:
    13.  $\rho_{k+1}(\sigma_j) = \frac{\rho_k(\sigma_j)\rho_{k-1}(\sigma_j)\alpha_{k-1}}{\rho_{k-1}(\sigma_j)\alpha_{k-1}(1 + \alpha_k\sigma_j) + \alpha_k\beta_{k-1}(\rho_{k-1}(\sigma_j) - \rho_k(\sigma_j))}$
    14.  $\alpha_k(\sigma_j) = \frac{\rho_{k+1}(\sigma_j)}{\rho_k(\sigma_j)}\alpha_k$
    15.  $\Xi_{k+1}(\sigma_j) = \Xi_k(\sigma_j) + \alpha_k(\sigma_j)\Pi_k(\sigma_j)$
    16.  $\beta_k(\sigma_j) = \left(\frac{\rho_{k+1}(\sigma_j)}{\rho_k(\sigma_j)}\right)^2 \beta_k$
    17.  $\Pi_{k+1}(\sigma_j) = \rho_{k+1}(\sigma_j)\Sigma_{k+1} + \beta_k(\sigma_j)\Pi_k(\sigma_j)$
    18. End Do
    19. End Do

```

for k=0:maximumsteps
    mul!(Ap,A,-p)
    #A_mul_B!(Ap,A,-p)
    rnorm = dot(r,r)
    α = rnorm/dot(p,Ap)
    @. x += α*p #Line 7
    @. r += -α*Ap #Line 8
    β = r'*r/rnorm #Line9
    @. p = r + β*p #Line 10
    Σ[1] = r[i] #Line 11
    for j = 1:M
        if abs(pk[j]) > eps
            pkp[j] = pk[j]*pkm[j]*αm/(pkm[j]*αm*(1.0+α*σ[j])+α*βm*(pkm[j]-pk[j]))#Line 13
            αkj = α*pkp[j]/pk[j]#Line 14
            θ[j] += αkj*Π[j,1] #Line 15
            βkj = ((pkp[j]/pk[j])^2)*β #Line 16
            Π[j,:] = pkp[j]*Σ+ βkj*Π[j,:]
            pkm[j] = pk[j]
            pk[j] = pkp[j]
        end
    end
    αm = α
    βm = β
    hi = real(rnorm)*maximum(abs.(pk))
    if hi < eps
        return θ
    end
end

```

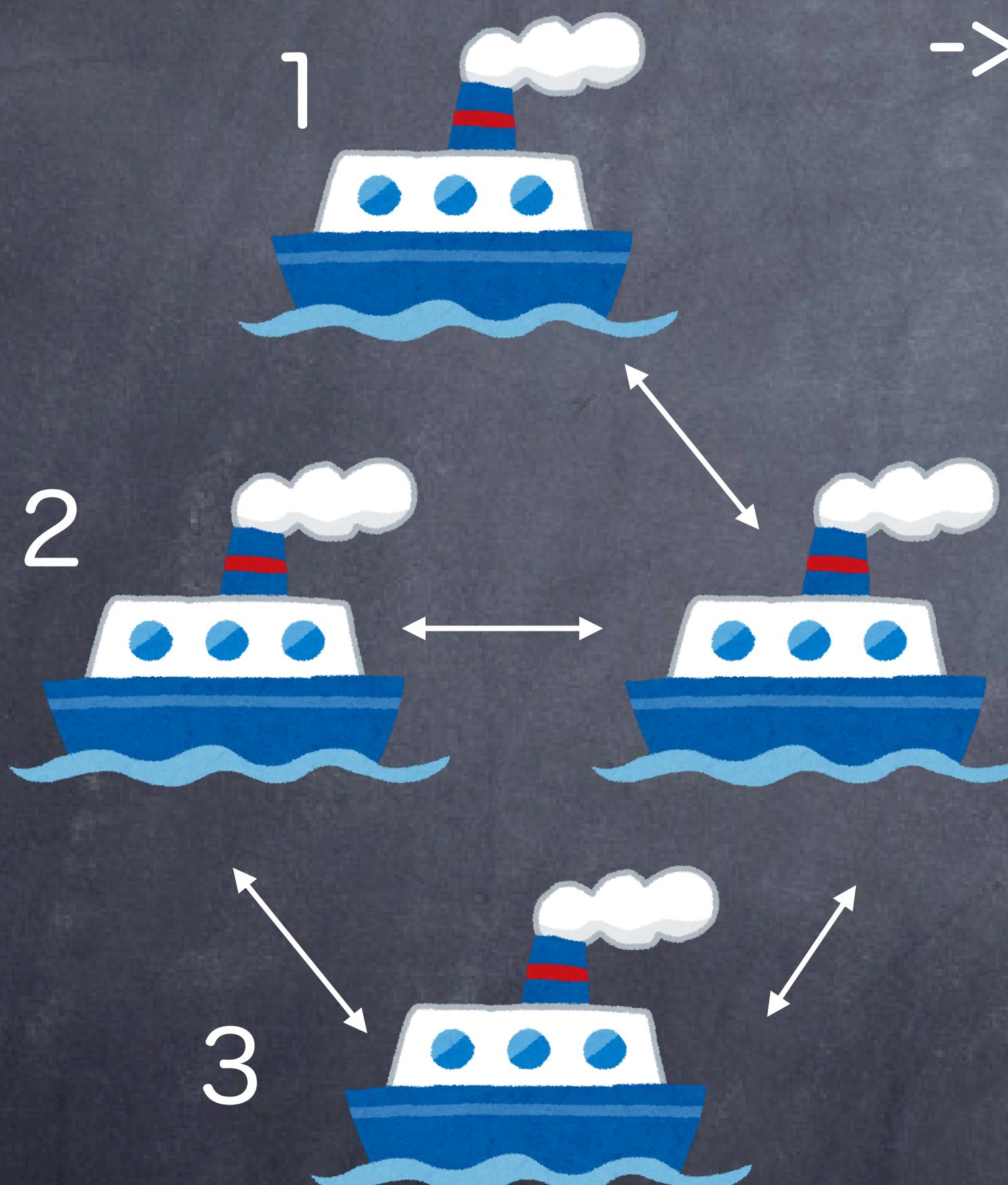
ポイント  
なるべくJuliaの基本機能を使う  
mul!とdot

これによって、mul!とdotが定義されている型ならどんなものでも使える

疎行列を使う

# MPI並列

## MPI(Message Passing Interface)



母艦を指定してもよいしみんな計算してもよい

-> CPUもメモリもそれぞれ別に持っている  
それぞれのプロセスがそれぞれ自分の仕事をする  
時々、他の人の結果をやりとりする  
それぞれの船が時々手旗信号で情報を送り合うような  
例  
プロセス0番からプロセス1番にデータを渡せ  
プロセス0番に全プロセスがデータaを集めろ  
プロセス0番から全プロセスにデータaを渡せ  
全プロセスのデータaを和をとり、結果を全プロセスに戻せ

# JuliaでのMPI



MPIはそれぞれのプロセスは独立に走るため、高速化をするには通信部分を気をつける必要がある

-> 大規模並列計算の場合、並列化技法に関して、それぞれの専門分野で大量の論文が存在する。それらを使うことができる

Distributed.jlでは、並列モデルが異なるため、自分で一から高速化技法を考える必要がある -> 計算科学者でない人にとっては辛い

並列化手法を編み出して実装しても実績（論文）にならない…

「大規模並列計算可能な既存の良いFortranコードがあるなら、それを使えばいいじゃない」

->それはそう。

その道のプロならそのまま使えばいい。でも新規参入者はコンパイルできなかったり…

大規模コードの大規模並列化ではなく、自前の 小規模コードの大規模並列化に向いている？

コードの拡張がしたい場合は、Juliaの方が読みやすい？

# MPI.jlのインストール

1. 何らかの方法でMPIをインストールする

しなくともMPI.jlが自動で入れてくれる

macならbrew install openmpiのようなもので入れられる

2. MPI.jlのドキュメントを見ながら入れる

通常はadd MPIで入るはず

- Open MPI
- MPICH (v3.1 or later)
- Intel MPI
- Microsoft MPI
- IBM Spectrum MPI
- MVAPICH
- Cray MPICH
- Fujitsu MPI
- HPE MPT/HMPT

## クラスターやスパコンの場合

add MPIPreferencesでMPIPreferences.jlをインストール  
using MPIPreferences  
MPIPreferences.use\_system\_binary()

これで自分のMPIの環境を自動検出して設定してくれる

あとはadd MPIをするだけ

間違いなく対応しているMPI SGIなど対応していない場合は、OpenMPIを使用すると良いかもしない

# JuliaでのMPI

```
MPI.Init() # MPI初期化  
test()  
MPI.Finalize() #MPI終了
```

MPIの初期化と終了

```
comm = MPI.COMM_WORLD  
nprocs = MPI.Comm_size(comm)  
myrank = MPI.Comm_rank(comm)
```

MPIでは、自分が何番目のプロセスなのか、  
何個プロセスがあるか、を知る必要がある

“全プロセスのデータwaを和をとり、結果を全プロセスに戻せ”

```
wa = MPI.Allreduce(wa,MPI.SUM,comm)
```

FortranやCと異なり、waがどんな型なのか、何個の要素を持つのか、ということを指示する必要がない

```
mpirun -np 4 julia sample.jl 実行もFortranコードと同じように行えばよい
```

# 実行例

```
using MPI
MPI.Init()

comm = MPI.COMM_WORLD
println("Hello world, I am $(MPI.Comm_rank(comm)) of $(MPI.Comm_size(comm))")
MPI.Barrier(comm)
```

日本原子力研究開発機構

HPE SGI8600



GPU演算部 : 9.739PFLOPS

NVIDIA V100 1080基

Xeon Gold 24コア×2 ×272-> 13056コア

CPU演算部:2.801PFLOPS

Xeon Gold 20コア×2 ×706-> 28240コア

module loadでOpenMPIをロードし、MPIPreferencesで環境設定

```
mpirun julia test.jl
Hello world, I am 0 of 40
Hello world, I am 1 of 40
Hello world, I am 2 of 40
Hello world, I am 3 of 40
Hello world, I am 4 of 40
Hello world, I am 5 of 40
Hello world, I am 6 of 40
Hello world, I am 7 of 40
Hello world, I am 8 of 40
Hello world, I am 9 of 40
Hello world, I am 10 of 40
Hello world, I am 11 of 40
Hello world, I am 12 of 40
Hello world, I am 13 of 40
Hello world, I am 14 of 40
Hello world, I am 15 of 40
Hello world, I am 16 of 40
Hello world, I am 17 of 40
Hello world, I am 18 of 40
Hello world, I am 19 of 40
Hello world, I am 21 of 40
Hello world, I am 22 of 40
Hello world, I am 23 of 40
Hello world, I am 24 of 40
Hello world, I am 25 of 40
Hello world, I am 26 of 40
Hello world, I am 27 of 40
Hello world, I am 28 of 40
Hello world, I am 29 of 40
Hello world, I am 30 of 40
Hello world, I am 31 of 40
Hello world, I am 32 of 40
Hello world, I am 33 of 40
Hello world, I am 34 of 40
Hello world, I am 35 of 40
Hello world, I am 36 of 40
Hello world, I am 37 of 40
Hello world, I am 38 of 40
Hello world, I am 39 of 40
Hello world, I am 20 of 40
```

# 実行例

## 1次元強束縛模型

```
function make_mat(n,n0,V0)
A = spzeros(Float64,n,n)
t = -1.0
μ = -0.5
for i=1:n
    dx = 1
    j = i+dx
    if 1 <= j <= n
        A[i,j] = t
    end
    dx = -1
    j = i+dx
    if 1 <= j <= n
        A[i,j] = t
    end
    A[i,i] = -μ
    if i == n0
        A[i,i] += V0
    end
end
return A
end
```

## 差分化された2階微分演算子D+対角要素V

```
function main()
MPI.Init()
stime = time_ns()*10^(-9)
comm = MPI.COMM_WORLD
n=1000
n0 = 500
V0 = 3
A = make_mat(n,n0,V0)
nene = 100
energies = range(-2.5,3,length=nene)
η=0.05
σ = energies .+ im*η
ista,iend,nbun = start_and_end(n,comm)

ldos = zeros(nene,nbun)
count = 0
for i=ista:iend
    count += 1
    Gii = greensfunctions(i,i,σ,A)
    ldos[:,i] = -(1/π)*imag(Gii)
end
ldosall = reshape(MPI.Allgather(ldos,comm),(nene,n))
myrank = MPI.Comm_rank(comm)
```

$H = D + V$

$e_i^T (z - H)^{-1} e_i$ をRSCG.jlで計算

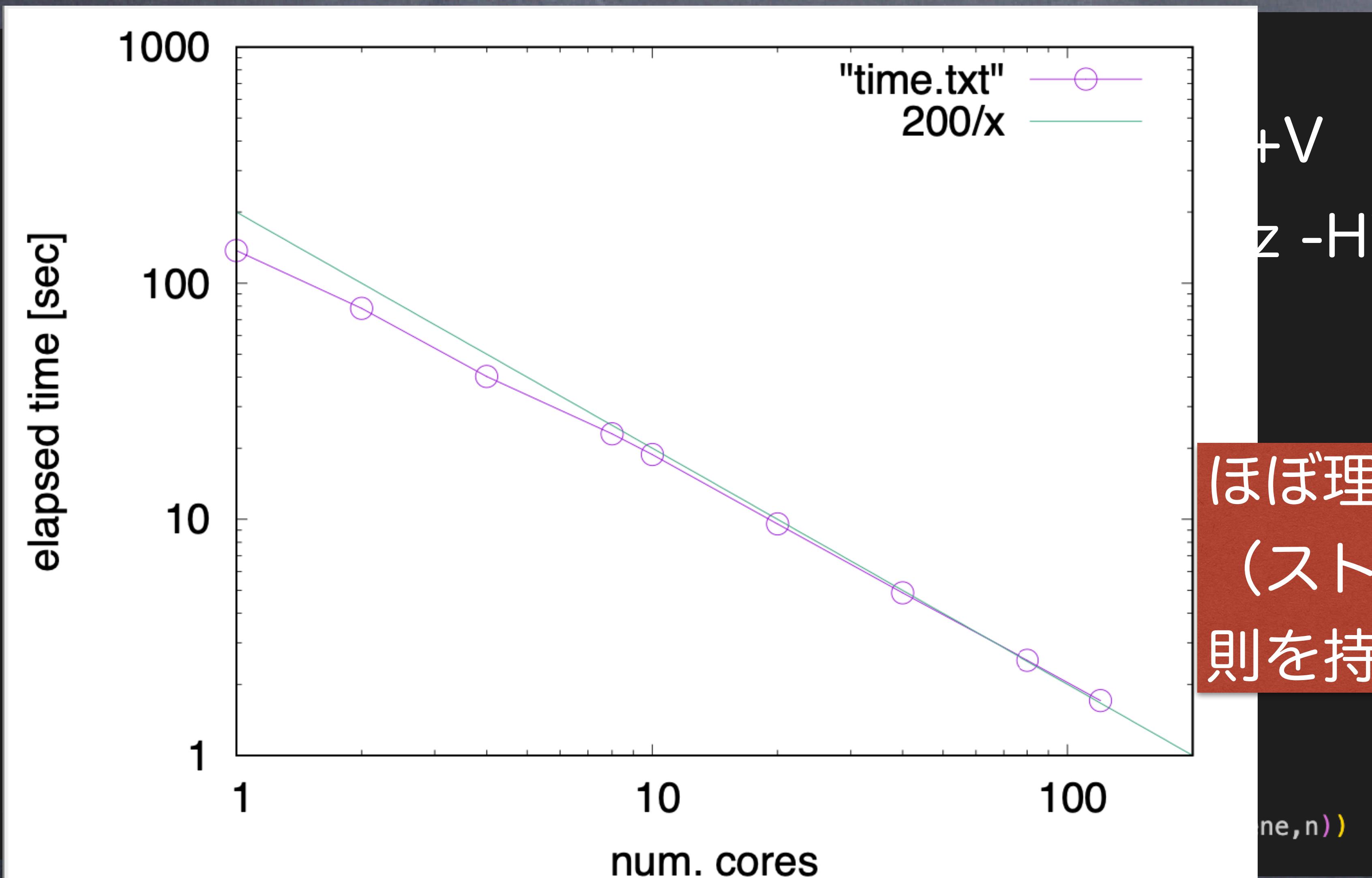
iのループを並列化

原子力機構のスパコン上で  
MPI.jlを使用してみる

Julia 1.9.1で実行

# 実行例

1次元強束縛模型 差分化された2階微分演算子D+対角要素V



ほぼ理想的なスケーリング  
(ストロングスケーリング)  
則を持つ

ne, n)

+ V  
 $Z - H)^{-1} e_i$  を RSCG.jl で 計算  
i の ループを並列化

# 並列計算について

## MPI.jlがこれまでの並列計算の知見を使えるため有益



スレッド並列+プロセス並列の効率的な計算方法  
はどうすればいいのか？

スレッド並列+プロセス並列をするときに何が問題か

スレッド並列はforループを並列化できる

-> 一つのforループを並列化する時に立ち上がりに若干のオーバーヘッドが存在する

-> 無数のforループを回す場合、無視できないほどのロスになる

FortranでのOpenMPでも同じ問題がある

-> OpenMPの場合、スレッドを立ち上げたままにすることで回避できる

Juliaでのやり方はわからない……

OpenMPみたいにスレッド並列ができる  
ば先行研究の技法が使えるのになあ……

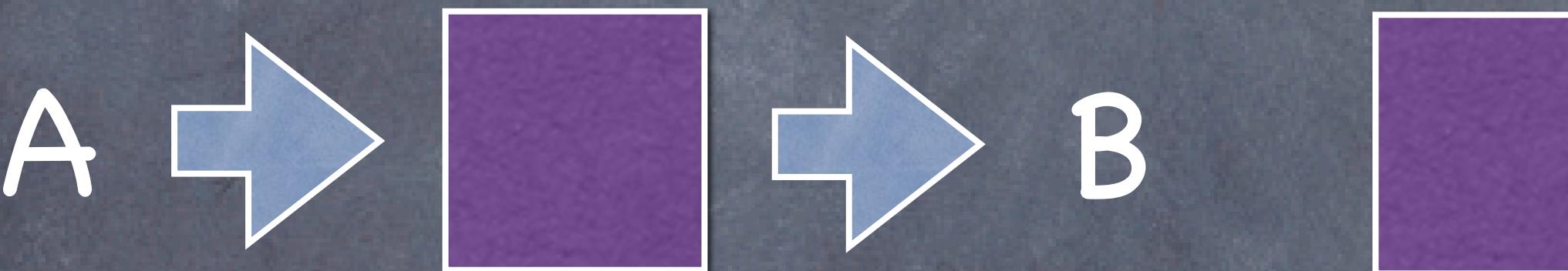
# 機械學習

# 機械学習

そもそも機械学習とは何か？

猫画像判別、自動運転、囲碁、自動お絵描きetc

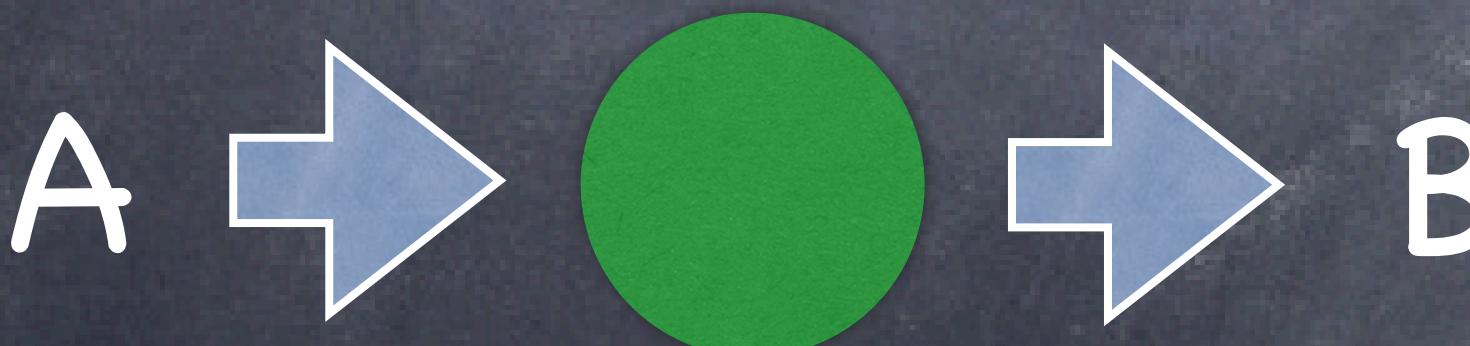
何かを判断したり、推測したりを機械が行う -> 代わりにやってもらう



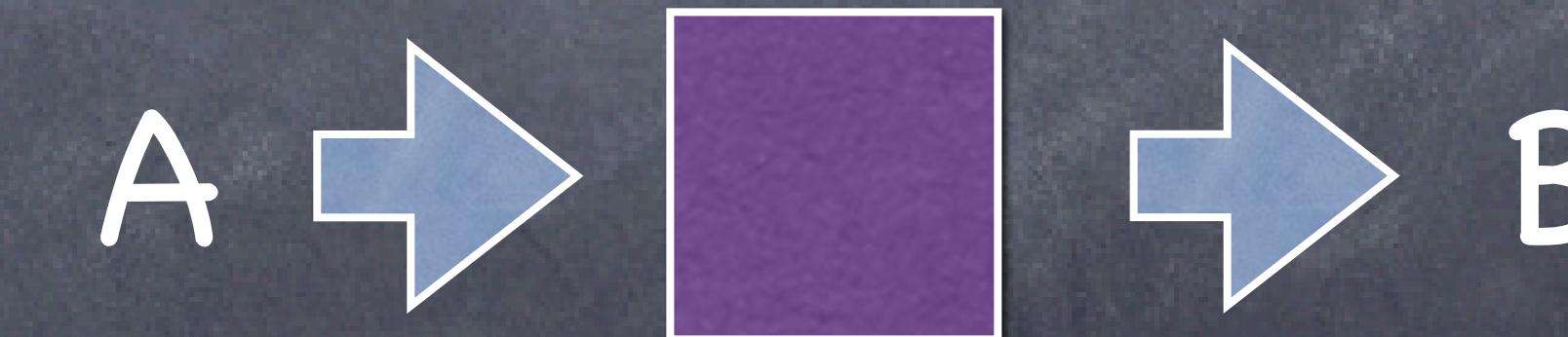
:人間には定式化できない何か

シミュレーション分野での機械学習の使用例

これまで計算が大変だった部分を機械が行う



定式化済みだけど計算コスト大

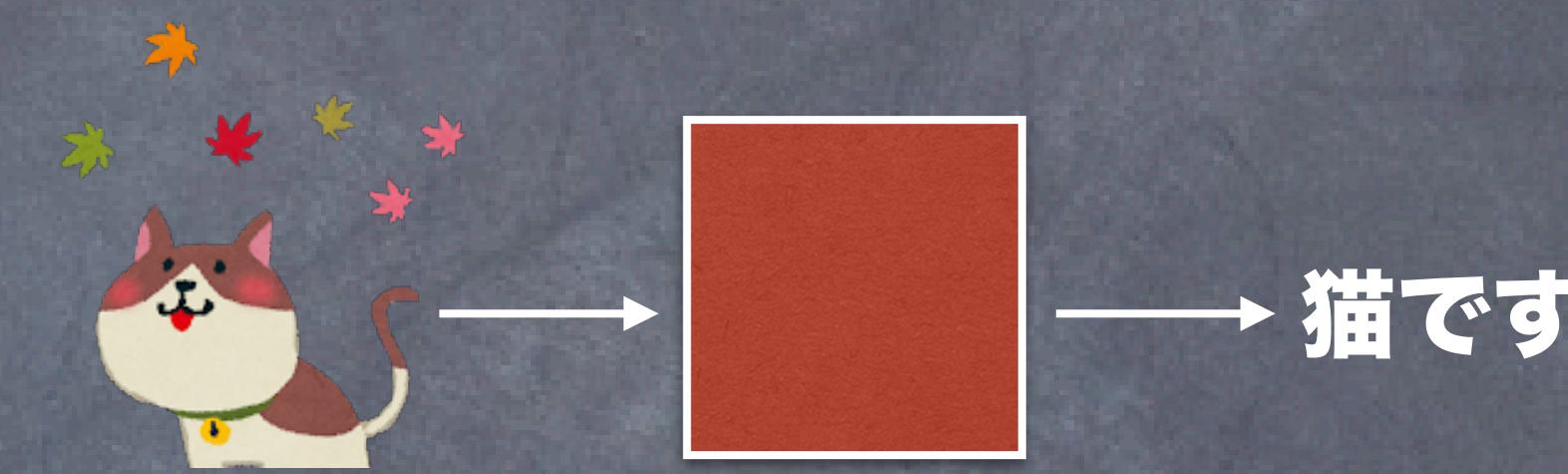
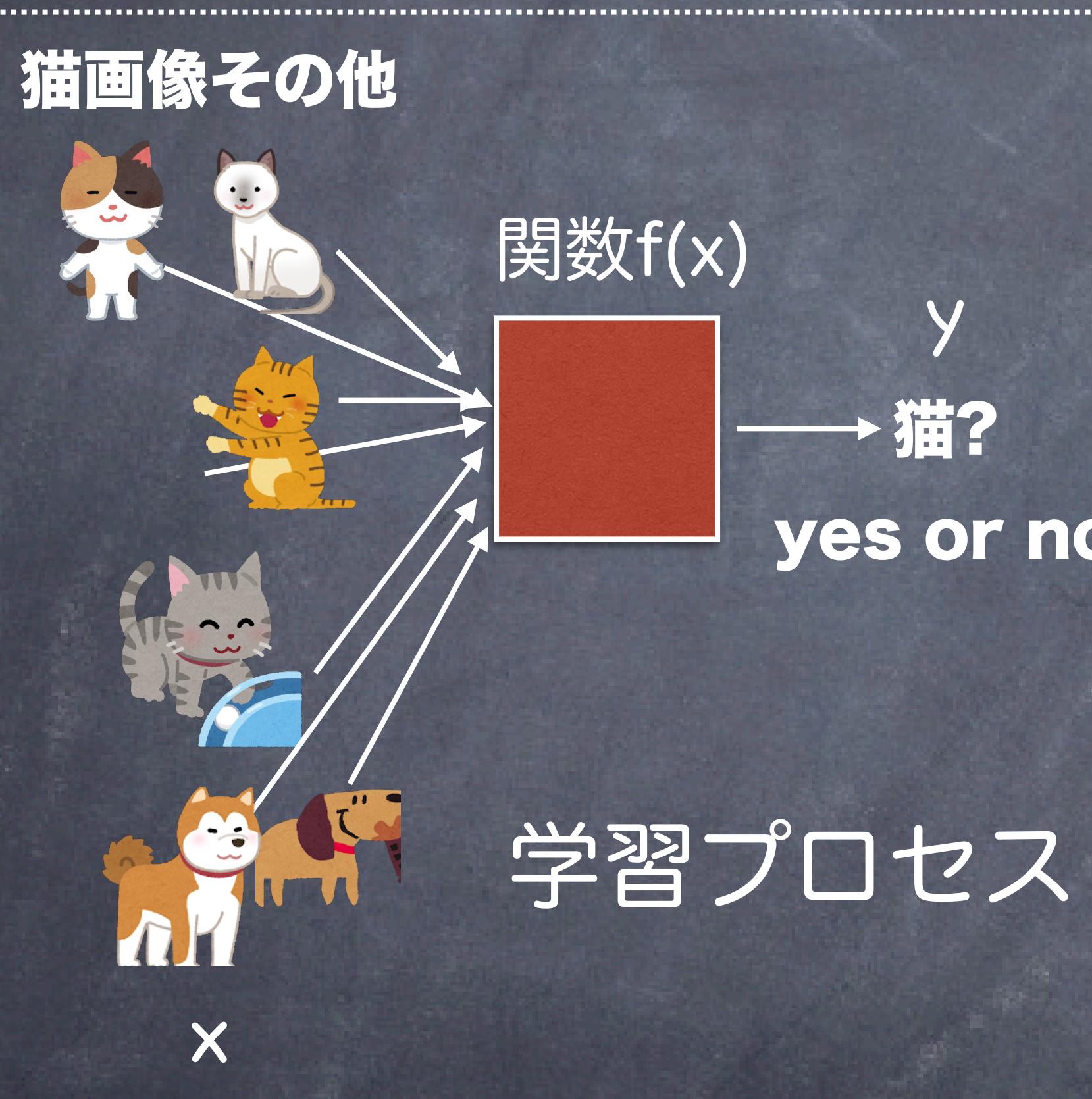


計算コストの軽い機械

シミュレーションの高速化に機械学習を用いる

# 機械学習とは

## 教師あり学習



大量の入力データ  $x$  を用いて、  
 $f(x)=y$  を満たす関数  $f(x)$  を決める

例：囲碁棋譜データ → 勝利の方程式

囲碁プロ棋士に勝利

# 機械学習とは

## 教師あり学習

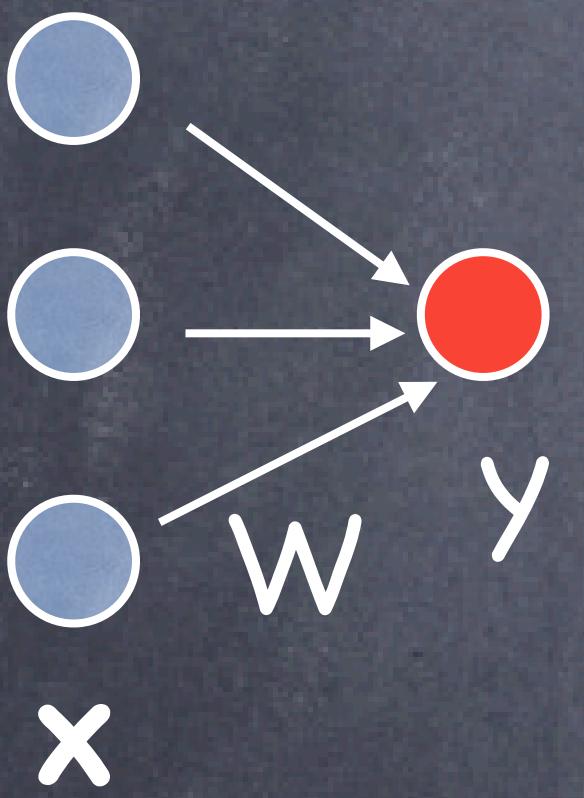
大量の入力データ $x$ を用いて、 $f(x)=y$ を満たす関数 $f(x)$ を決める

一番シンプルな例

$$y = ax + b \quad \text{直線で近似 (線形回帰)}$$

インプットが複数→ $x$ がベクトル

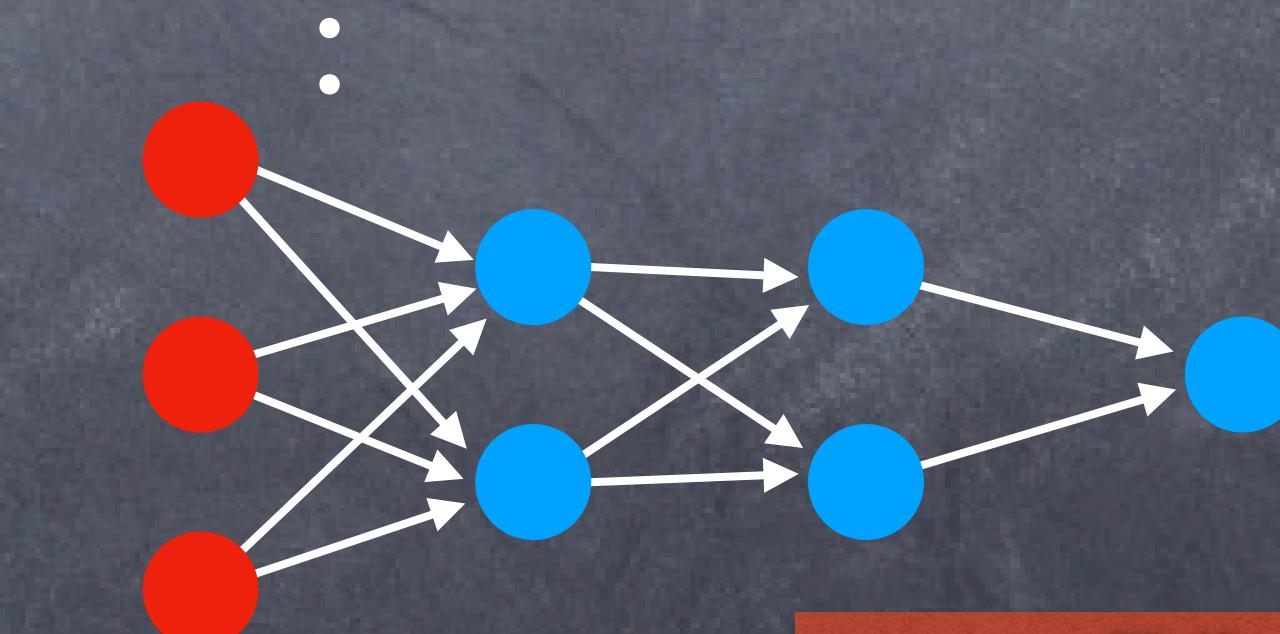
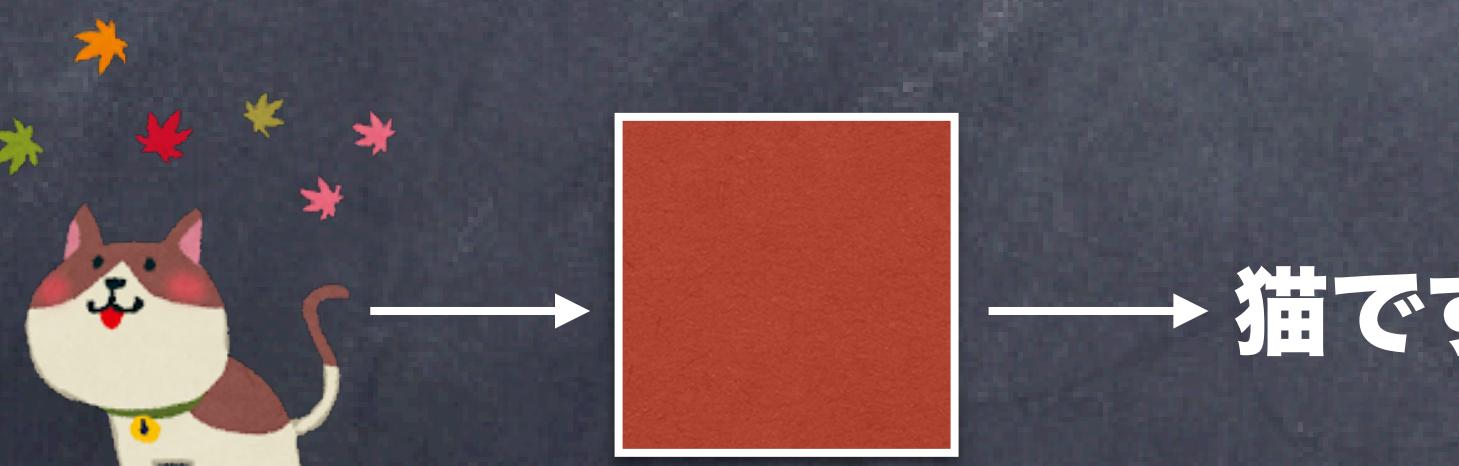
$$y = Wx + b$$



もっと表現力を高めたい

$$y = W_2 f(W_1 x + b_1) + b \quad f: \text{非線形関数}$$

$$y = W_3 f(W_2 f(W_1 x + b_1) + b_2) + b$$



深層学習

ニューラルネットワーク

# 昨今の機械学習

ChatGPT:生成AI

チャットの文章を入れるといい感じに返答してくれる

お絵描き系AI

文章で絵を説明すると、その文章にあった絵を描いてくれる

さまざまな形のネットワークが提案され続けている

進展が非常に速い…

-> 自分の研究に使いたかったらどうする？

# 機械学習とは

パラメータ $\theta$ とインプット $x$ の関数 $f$   $y = f(\theta, x)$

“loss関数” $L(\theta, f)$ を最小化する $\theta$ を見つける

最小化するには、パラメータ微分が必要

複雑な非線形関数をどうやって微分する？

-> 自動微分という技術がある

Fortranで機械学習をすることは可能か？

->可能。実際、機械学習分子動力学パッケージaenet等はFortranで書かれている

次々と出てくる新種の構造のネットワークの自動微分をその都度Fortranで実装する？？

# Juliaで機械学習

自動微分の詳細について見る前に、Juliaで機械学習をする方法について

いくつか機械学習用パッケージがある:Flux.jlやLux.jlなど

->Flux.jlが一番有名

```
using Flux
function main()
    model = Chain(Dense(2,10,relu),Dense(10,10,σ),Dense(10,1,sum))
    x = Float32[0.2,0.3]
    println(model(x))                                隠れ層2層の全結合ニューラルネットワーク
    dfdx = gradient(model,x)[1]
    println(dfdx)
end
main()
```

これだけで微分可能なニューラルネットワークが出来上がっている

Chainで次々とレイヤーを繋げていく。普通の関数を間に挟んでもよい  
グラディエントの計算ができる

# カスタムレイヤー

全結合ではない、自分独自のネットワークを作りたい

```
using Flux
struct Dense2
    W
    b
    activation
end
Dense2(in::Integer,out::Integer) = Dense2(randn(out,in),randn(out),x -> x)
Dense2(in::Integer,out::Integer,activation) =
    Dense2(randn(out,in),randn(out),activation)
(m::Dense2)(x) = m.activation.(m.W*x .+ m.b) インプットに対してどう応答するか定義する
Flux.trainable(a::Dense2) = (weight=a.W,bias=a.b) 訓練用パラメータを登録
Flux.@functor Dense2 Fluxでレイヤーとして使えるように登録
```

レイヤー内のそれぞれの計算が自動微分できるならば、標準のレイヤーと同等に使える  
自動微分できない関数がある場合には、その関数用に自動微分を定義する（後述）

# Flux.jlの使用例

“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”

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

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

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

**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

# Flux.jlの使用例

“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

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

エネルギーの定義

```
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

あとは訓練するだけ

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

PyTorchやTensorFlowと違い、  
テンソルにする必要がない

# Flux.jlの使用例

“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”

ハミルトンのAdamというオプティマイザーを使用

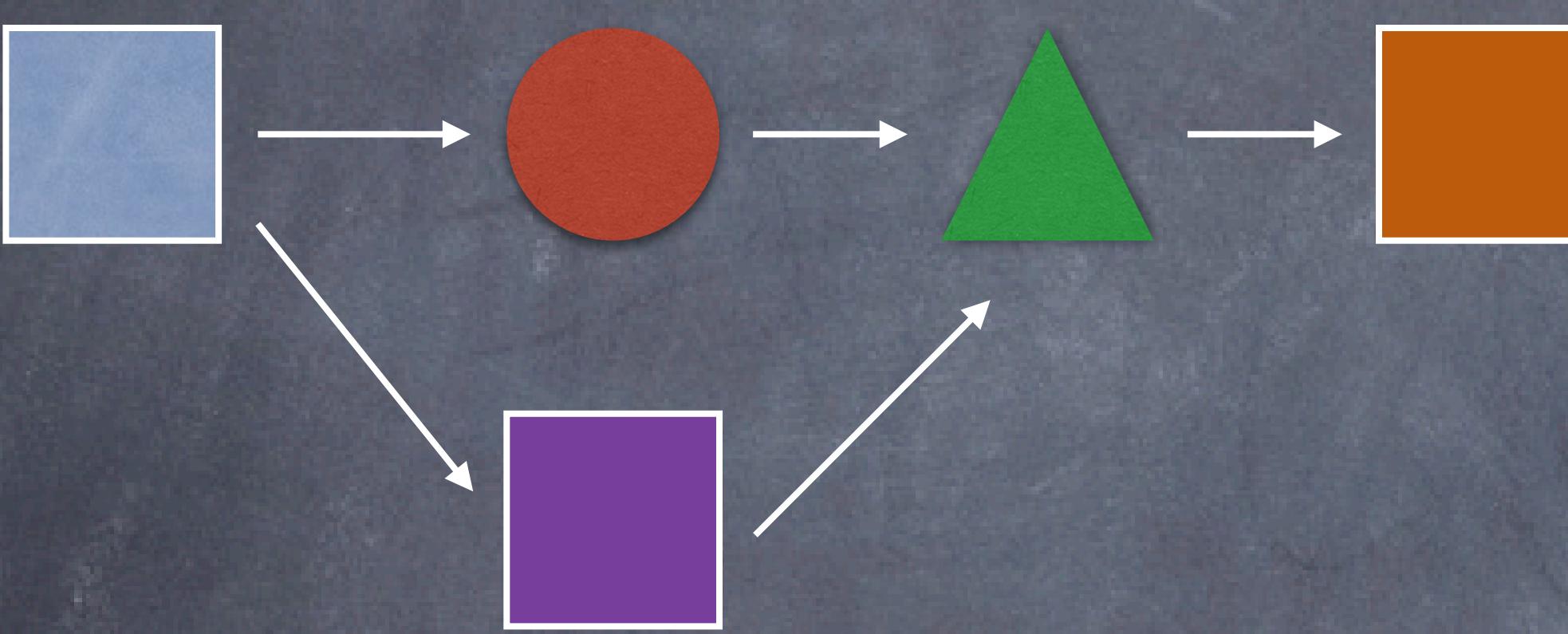
```
opt = Flux.setup(Adam(), model)
for i=1:50000
    Flux.train!(energy, model, (xs,), opt)
    e = energy(model, xs)
    if i % 1000 == 0
        println("i = $i energy = $e")
    end
end
```

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

# 自動微分とは？

機械学習で登場する非線形関数の特徴：ブロック化されている



$$\begin{aligned}
 x_1 &= f_1(x) \\
 x_2 &= f_2(x_1) + g_2(x_1) \\
 x_3 &= f_3(x_2) \\
 y &= f_4(x_3)
 \end{aligned}$$

インプットを順番に計算して  
いくことで出力が得られる

$$y = f_4(f_3(f_2(f_1(x)) + g_2(f_1(x))))$$

例  $df/dx$ を計算したい

$$a(x) = \sin(x)$$

連鎖律を使う

$$b(a) = 0.2 + a$$

$$c(b) = \sqrt{1 + b}$$

$$f(x) = c(b(a(x)))$$

$$\frac{\partial f}{\partial x} = \frac{\partial c}{\partial x} = \frac{\partial a}{\partial x} \frac{\partial c}{\partial a} \quad \frac{\partial c}{\partial a} = \frac{\partial b}{\partial a} \frac{\partial c}{\partial b} \quad \frac{\partial c}{\partial b} = \frac{\partial c}{\partial b} \frac{\partial c}{\partial c}$$

↑                      ↑                      ←

誤差逆伝播法と呼ばれている

# 自動微分とは？

例  $df/dx$ を計算したい

$$a(x) = \sin(x)$$

連鎖律を使う

$$b(a) = 0.2 + a$$

$$c(b) = \sqrt{1 + b}$$

$$f(x) = c(b(a(x)))$$

誤差逆伝播と呼ばれている

$$\frac{\partial f}{\partial x} = \frac{\partial c}{\partial x} = \frac{\partial a}{\partial x} \frac{\partial c}{\partial a} \quad \frac{\partial c}{\partial a} = \frac{\partial b}{\partial a} \frac{\partial c}{\partial b} \quad \frac{\partial c}{\partial b} = \frac{\partial c}{\partial b} \frac{\partial c}{\partial c}$$



微分を手で書ける

それぞれのブロック（あるいはレイヤー）の微分がわかれれば計算できる

プログラミングでどうやって実現するの？

そのブロックがどのような計算をしているかを記録し、  
その微分も計算しておけば良い？

# 自動微分とは？

プログラミングでどうやって実現するの？

そのブロックがどのような計算をしているかを記録し、  
その微分も計算しておけば良い？

Python:機械学習分野ではデファクトスタンダードなプログラミング言語

機械学習パッケージ: Tensorflow、PyTorch etc.

**Define by Run**という技術で自動微分を実現

Tensorという型を導入し、その型を使った計算を記録しておく

-> 微分したい計算は、このTensorという型で表現しなければならない

一方、Juliaはどんな型も自動微分できる。なぜ？？

# Juliaでの自動微分

Juliaでは、コードは動的にコンパイルされて実行されている

コンパイル：構文を解析し、最適なコードを生成する

Juliaは常に構文を解析している！

-> どのような計算が行われているか、把握している

計算の方法を記録する必要はない！ ->Tensorとかいらない！

だから、Juliaで書いたコードに対して、普通に微分できる

```
function samplefunc(x)
    a = sin(x)
    b = 0.2+a
    c = sqrt(1+b)
    return c
end
```

Zygoteという  
パッケージを  
使えば

```
x = 0.5
dfdx = gradient(samplefunc,x)[1]
```

これだけで微分できてしまう

# Juliaでの自動微分

計算したい関数

$$\begin{aligned} a_1 &= f_1(x) \\ a_2 &= f_2(a_1) \\ &\vdots \\ a_i &= f_i(a_{i-1}) \\ &\vdots \\ a_{N-1} &= f_{N-1}(a_{N-2}) \\ a_N &= f_N(a_{N-1}) \\ f(x) &= a_N \end{aligned}$$

その微分

$$\begin{aligned} \frac{\partial a_N}{\partial a_N} &= 1 \\ \frac{\partial a_N}{\partial a_{N-1}} &= \frac{\partial a_N}{\partial a_{N-1}} \frac{\partial a_N}{\partial a_N} = \left. \frac{\partial f_N(x)}{\partial x} \right|_{a_{N-1}} \frac{\partial a_N}{\partial a_N} \\ \frac{\partial a_N}{\partial a_{N-2}} &= \frac{\partial a_{N-1}}{\partial a_{N-2}} \frac{\partial a_N}{\partial a_{N-1}} = \left. \frac{\partial f_{N-1}(x)}{\partial x} \right|_{a_{N-2}} \frac{\partial a_N}{\partial a_{N-1}} \\ &\vdots \\ \frac{\partial a_N}{\partial a_{i-1}} &= \left. \frac{\partial f_i(x)}{\partial x} \right|_{a_{i-1}} \frac{\partial a_N}{\partial a_i} \\ &\vdots \\ \frac{\partial a_N}{\partial a_1} &= \left. \frac{\partial f_2(x)}{\partial x} \right|_{a_1} \frac{\partial a_N}{\partial a_2} \\ \frac{\partial a_N}{\partial x} &= \left. \frac{\partial f_1(x)}{\partial x} \right|_x \frac{\partial a_N}{\partial a_1} \end{aligned}$$

微分計算に必要なもの

$$\frac{\partial f_i}{\partial x} \frac{\partial a_N}{\partial a_i}$$

$\frac{\partial f_i}{\partial x}$  は  $x$  の関数  $\frac{\partial a_N}{\partial a_i}$  は伝播してくる数字

$$B_y(\bar{y}, x) \equiv \frac{\partial y}{\partial x} \bar{y} \text{ 用意すると}$$

$$\frac{\partial a_N}{\partial a_{i-1}} = B_y\left(\frac{\partial a_N}{\partial a_i}, a_{i-1}\right)$$

と書ける

# Juliaでの自動微分

計算したい関数

$$a_1 = f_1(x)$$

$$a_2 = f_2(a_1)$$

⋮

$$a_i = f_i(a_{i-1})$$

⋮

$$a_{N-1} = f_{N-1}(a_{N-2})$$

$$a_N = f_N(a_{N-1})$$

$$f(x) = a_N$$

$$\frac{\partial a_N}{\partial a_{i-1}} = B_{f_i}\left(\frac{\partial a_N}{\partial a_i}, a_{i-1}\right)$$

各レイヤー*i*で  $B_{f_i}(\bar{y}, x)$

がわかれば自動微分できる

$$y = \sin(x) \text{ なら } B_y(\bar{y}, x) = \cos(x)\bar{y}$$

->このBを「pullback」と呼ぶ

Juliaは構文解析によって基本的な演算の集まりに分割できている

`cos`や`sin`、足し算や掛け算など、基本的な演算にBを定義しておけばよい

多重ディスパッチにより`rrule(::typeof(sin), x)`のようなメソッドを定義する

`ChainRulesCore`というパッケージにこれらの演算が定義されている

自作の型の場合は、自分で`rrule`を作れば微分できるようになる！

# 独自モデルの自動微分

インプットSに対してHを計算する関数

$$\check{S}_{ki} = \sum_l \alpha_l (S_{ki+l} + S_{ki-l})$$

$$H(S) = J \sum_i \sum_k \check{S}_{ki} (\check{S}_{ki+1} + \check{S}_{ki-1})$$

レイヤーを定義

```
struct Smearing
    α::Vector{Float64}
end
Smearing(n) = Smearing(randn(n))
(m::Smearing)(S) = calc_smearing(S,m.α)
Flux.trainable(a::Smearing) = (α=a.α,)
Flux.@functor Smearing
```

```
function calc_smearing(S,α)
    nspin,nsite = size(S)
    n = length(α)
    Snew = deepcopy(S)
    for i=1:nsite
        for l=1:n
            j = i+l
            j += ifelse(j > nsite,-nsite,0)
            for k=1:nspin
                Snew[k,i] += α[l]*S[k,j]
            end
            j = i-l
            j += ifelse(j < 1,nsite,0)
            for k=1:nspin
                Snew[k,i] += α[l]*S[k,j]
            end
        end
    end
    return Snew
end
```

# 独自モデルの自動微分

モデルを定義

```
model = Chain(Smearing(n), S -> calc_hamiltonian(S,J))
```

```
function calc_hamiltonian(S,J)
    nspin,nsite = size(S)
    ham = 0.0
    for i=1:nsite
        j = i+1
        j += ifelse(j > nsite,-nsite,0)
        for k=1:nspin
            ham += J*S[k,i]*S[k,j]
        end
        j = i-1
        j += ifelse(j < 1,nsite,0)
        for k=1:nspin
            ham += J*S[k,i]*S[k,j]
        end
    end
    return ham
end
```

$$\check{S}_{ki} = \sum_l \alpha_l (S_{ki+l} + S_{ki-l})$$

$$H(S) = J \sum_i \sum_k \check{S}_{ki} (\check{S}_{ki+1} + \check{S}_{ki-1})$$

モデルをパラメータとモデルに分ける

```
θ, re = Flux.destructure(model)
H = re(θ)(S)
```

パラメータ微分を計算したい

```
dHdθ_a = gradient(θ -> re(θ)(S), θ)[1]
for i=1:length(θ)
    println("$i dHdθ = ", dHdθ_a[i])
end
```

# 独自モデルの自動微分

$\check{S}_{ki} = \sum_l \alpha_l (S_{ki+l} + S_{ki-l})$  をパラメータ  $\alpha$  で微分できるようにしたい

$B_y(\bar{y}, x) \equiv \frac{\partial y}{\partial x} \bar{y}$  を定義すればよい

ポイント：Bの型はxの型と同じ

$B_{\check{S}_{ki}}(\bar{S}_{ki}, S_{ki})$  と  $B_{\bar{S}_{ki}}(\bar{S}_{ki}, \alpha_i)$  が必要

$$B_{\check{S}_{ki}}(\bar{S}_{ki}, S_{ki}) = \sum_{nj} \frac{\partial f}{\partial \check{S}_{nj}} \frac{\partial \check{S}_{nj}}{\partial S_{ki}}$$

$$B_{\check{S}_{ki}}(\bar{S}_{ki}, \alpha_i) = \sum_{nj} \frac{\partial f}{\partial \check{S}_{nj}} \frac{\partial \check{S}_{nj}}{\partial \alpha_i} \quad \text{を定義する}$$

```

function ChainRulesCore.rrule(::typeof(calc_smearing), S, α)
    nspin, nsite = size(S)
    n = length(α)
    y = calc_smearing(S, α)
    function pullback(ybar)
        sbar = NoTangent()
        fSbar = zero(S)
        fabar = zero(α)

        for i=1:nsite
            for l=1:n
                j = i+l
                j += ifelse(j > nsite, -nsite, 0)
                for k=1:nspin
                    fSbar[k, i] += ybar[k, j] * α[l]
                    fabar[l] += ybar[k, i]*S[k,j]
                end
                j = i-l
                j += ifelse(j < 1, nsite, 0)
                for k=1:nspin
                    fSbar[k, i] += ybar[k, j] * α[l]
                    fabar[l] += ybar[k, i]*S[k,j]
                end
            end
        end
        return sbar, fSbar, fabar
    end
    return y, pullback
end

```

$$B_{\check{S}_{ki}}(\bar{S}_{ki}, S_{ki}) = \sum_n \sum_l \frac{\partial f}{\partial \check{S}_{nj}} \alpha_l$$

$$B_{\check{S}_{ki}}(\bar{S}_{ki}, \alpha_i) = \sum_n \sum_l \frac{\partial f}{\partial \check{S}_{nj}} S_{ki}$$

```

function ChainRulesCore.rrule(::typeof(calc_smearing),S,α)
    nspin,nsite = size(S)
    n = length(α)
    y = calc_smearing(S,α)
    function pullback(ybar)
        sbar = NoTangent()
        fSbar = zero(S)
        fabar = zero(α)

        for i=1:nsite
            for l=1:n
                j = i+l
                j += ifelse(j > nsite,-nsite,0)
                for k=1:nspin
                    fSbar[k, i] += ybar[k, j] * α[l]
                    fabar[l] += ybar[k, i]*S[k,j]
                end
                j = i-l
                j += ifelse(j <1,nsite,0)
                for k=1:nspin
                    fSbar[k, i] += ybar[k, j] * α[l]
                    fabar[l] += ybar[k, i]*S[k,j]
                end
            end
        end
        return sbar,fSbar,fabar
    end
    return y, pullback
end

```

$$\check{S}_{ki} = \sum_l \alpha_l (S_{ki+l} + S_{ki-l})$$

$$B_{\check{S}_{ki}}(\bar{S}_{ki}, S_{ki}) = \sum_n \sum_l \frac{\partial f}{\partial \check{S}_{nj}} \alpha_l$$

$$B_{\check{S}_{ki}}(\bar{S}_{ki}, S_{ki}) = \sum_n \sum_l \frac{\partial f}{\partial \check{S}_{ni-l}} S_{ki}$$

rruleにpullbackを返す関数  
を実装すればよい

```

dHdθ_a = gradient(θ -> re(θ)(S),θ) [1]
for i=1:length(θ)
    println("$i dHdθ = ",dHdθ_a[i])
end

```

あとはgradientで自動微分できる

# 独自モデルの自動微分

インプットSに対してHを計算する関数

$$\check{S}_{ki} = \sum_l \alpha_l (S_{ki+l} + S_{ki-l})$$

$$H(S) = J \sum_i \sum_k \check{S}_{ki} (\check{S}_{ki+1} + \check{S}_{ki-1})$$

先程のrruleを定義するだけで、

$\frac{\partial H(S)}{\partial S_{ik}}$  も  $\frac{\partial H(S)}{\partial \alpha_i}$  も計算できる

レイヤーを複数重ねても自動微分ができる

```
model = Chain(Smearing(2), Smearing(3), Smearing(2), S -> calc_hamiltonian(S, J))
```

## 応用例

Attentionレイヤーを  
Flux.jlで実装し同変  
transformerを実装した

### Self-learning Monte Carlo with equivariant Transformer

Yuki Nagai\*

CCSE, Japan Atomic Energy Agency, 178-4-4, Wakashiba, Kashiwa, Chiba 277-0871, Japan

Akio Tomiya†

Faculty of Technology and Science, International Professional University of Technology,  
3-3-1, Umeda, Kita-ku, Osaka, 530-0001, Osaka, Japan

YN and A. Tomiya, arXiv:2306.11527

# Juliaで機械学習

シンプルに書いて機械学習ができる

どんな型でも自動微分できる -> あとで自動微分を実装することすらできる

rruleはある関数の微分を計算できるようになる

->あえて書くことで高速に計算することも可能

(連鎖律を短くすることができる)

GPUも使える

AMD GPUやMacのMetalもできるようになってきている

```
θ, re = Flux.destructure(model)  
H = re(θ)(S)
```

パラメータとモデルにバラすと、色々なパッケージで最適化可能

まとめ

# Juliaでの科学技術計算

並列計算：MPI.jlを使うとよい

他の言語でMPIを使っていた人は簡単に移行可能

MPIはじめてでも、他の言語よりやりやすい

FortranやCのMPIの資料を見ながら勉強し、そのままJuliaで実装できる

機械学習はFlux.jlで便利に使える

Juliaは構文を解析し、コンパイルしている

構文を解析してくれているため、自動微分との相性がよい

Tensorがいらない。どんな型でも自動微分できる

