

マス・フォア・インダストリ研究 No.27

数学と物理における Juliaの活用

編集 富谷 昭夫
横山 俊一
永井 佑紀
寺崎 敏志

Institute of Mathematics for Industry
Kyushu University

About the Mathematics for Industry Research

The Mathematics for Industry Research was founded on the occasion of the certification of the Institute of Mathematics for Industry (IMI), established in April 2011, as a MEXT Joint Usage/Research Center – the Joint Research Center for Advanced and Fundamental Mathematics for Industry – by the Ministry of Education, Culture, Sports, Science and Technology (MEXT) in April 2013. This series publishes mainly proceedings of workshops and conferences on Mathematics for Industry (MfI). Each volume includes surveys and reviews of MfI from new viewpoints as well as up-to-date research studies to support the development of MfI.

October 2022

Kenji Kajiwara

Director

Institute of Mathematics for Industry

Use of Julia in Mathematics and Physics

Mathematics for Industry Research No.26, Institute of Mathematics for Industry, Kyushu University
ISSN 2188-286X

Editors: Akio Tomiya, Shun'ichi Yokoyama, Yuki Nagai, Satoshi Terasaki

Date of issue: 18 January 2024

Publisher:

Institute of Mathematics for Industry, Kyushu University

Motooka 744, Nishi-ku, Fukuoka, 819-0395, JAPAN

Tel +81-(0)92-802-4402, Fax +81-(0)92-802-4405

URL <https://www.imi.kyushu-u.ac.jp/>

Printed by

Kijima Printing, Inc.

Shirogane 2-9-6, Chuo-ku, Fukuoka, 810-0012, Japan

TEL +81-(0)92-531-7102 FAX +81-(0)92-524-4411

数学と物理における Julia の活用

編集 : 富谷 昭夫
横山 俊一
永井 佑紀
寺崎 敏志

巻頭言:

Julia 言語は 2018 年にバージョン 1 が公開されたオープンソースの科学技術計算言語であり、一言でまとめると Fortran の様に高速でかつ Python の様に生産性の高い言語である。とはいえ、Fortran や Python、Mathematica 言語や MATLAB 言語など科学技術計算が可能なプログラミング言語はすでに数多ある。なぜ今 Julia 言語なのだろうか。Julia 言語は後発であるがゆえ、様々な言語の良いところを集め、また反省点が生かされている。後発ゆえの利点は数多いが例えば JIT (Just in time) コンパイルシステムは、Python でも Numba や JAX などを通じて実装が進んでいるが Julia 言語では言語の開発段階から想定されている。また多重ディスパッチによる関数の多重定義は、様々なデータ型に対しての操作を自然な実装で実現することができる枠組みとなっている。LLVM を用いたコンパイルも後発ゆえの選択だろうと思う。加えて開発コミュニティがユーザーに近いためユーザーの声が届きやすく、多くのパッケージ開発もユーザーコミュニティで進んでいる。Julia 言語の難点といえば、様々な分野において活用が始まっているが、2023 年時点では、Python や C/C++ の様に専門家が各機関にいるということもなく、最新の情報はいくつかの書籍かネットを参照するしかないのが現状である。そこで本研究会は、産学を含め、数学・物理学の最前線で研究を行っている研究者に講演を頂いた。数学・物理学を含んだ広い分野における様々な利用例が全国的に共有され、議論も活発に行われた。本研究会に参加した方、資料を参照した方の中から Julia 言語を用いた数学・物理学の研究が生まれれば至極幸せである。

富谷昭夫

材料科学においては、近年、急速に必要なとなっている幾何・代数の材料科学への適応に関わる研究の更なる加速が期待される。本研究会はその礎・足場として期待に応えるものとなったと考え、ここに報告をする。

組織委員代表 富谷昭夫

2024年 1月18日

組織委員

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

寺崎敏志 AtelierArith

Contents

巻頭言	i
Julia 入門	
後藤俊介 (来栖川電算), 寺崎敏志 (AtelierArith)	1
BasicBSpline.jl で始める B-spline	
堀川由人 (大阪大学)	89
応用数学の概念を用いた物理モデルをいかに Julia でプログラミングを行うか	
降旗大介 (大阪大学)	123
Julia と量子多体計算: スパースモデリングから Quantics Tensor Train まで	
品岡寛 (埼玉大学)	133
数論における Julia の援用	
横山俊一 (東京都立大学)	151
Julia による科学技術計算: 大規模並列計算について	
永井佑紀 (日本原子力研究開発機構)	169
素粒子物理における Julia の活用～格子 QCD の大規模計算に向けて～	
富谷昭夫 (大阪国際工科専門職大学)	195

概要:

題目: Julia 入門

講演者: 後藤俊介 (来栖川電算), 寺崎敏志 (AtelierArith)

概要: ライトユースにも使いたい、けれどもそれで高速性を犠牲にしたくない。総合的に開発できるものにしたい。超初心者にも習得は容易でありながら、ハッカーの満足にも応えられるものがほしい。」そういった願いからプログラミング言語 Julia が誕生しました。直近では v1.9 がリリースされ、長年課題であったパッケージの読み込み時間の短縮など、開発体験の改善が行われました。さらに和書の入門書も充実してきており、学習を始めるには絶好のタイミングといえるでしょう。本チュートリアル講演では Julia の入門として 1. インストールと基本的な使い方, 2. 型と多重ディスパッチ, 3. スレッド並列の基礎の 3 つのトピックを解説します。この機会に Julia に入門し、新しい体験を始めてみませんか。

題目: BasicBSpline.jl で始める B-spline

講演者: 堀川由人 (大阪大学)

概要: B-spline は滑らかさを保証した区分多項式を作るための道具であり、幾何形状表現や数値解析など広範な分野で活用されている。Julia における B-spline のパッケージは幾つか存在するが、その中でも BasicBSpline.jl を本講演では扱う。BasicBSpline.jl パッケージの使用方法を交えながら B-spline に関する理論を解説し、その応用例についても触れる。

題目: 応用数学の概念を用いた物理モデルをいかに Julia でプログラミングを行うか

講演者: 降旗大介 (大阪大学)

概要: 物理上の相分離現象に対するモデル方程式としては Cahn-Hilliard 方程式が有名かつ有用だが、この数値解析は計算量的に高コストである。そこでこの現象のごく初期過程を除いて状況を記述するようなシンプルなモデルを考案したい。この際、流体的挙動を模するために空間の Voronoi 分割という応用数学的手法をモデルに取り入れるのだが、これまでのコンピュータ言語ではこうしたモデルの数値解析にはそれなりに苦勞する面が多々あったのが現状である。これに対し、Julia を用いると比較的に数値解析が行えることを示すことで、微分方程式の数値解析に対して Julia がどのように用いられるかを紹介したい。

題目: Julia と量子多体計算: スパースモデリングから Quantics Tensor Train まで

講演者: 品岡寛 (埼玉大学)

概要:本講演では、Julia を用いた情報圧縮技術の量子多体計算への応用を紹介する。前半では、虚時間グリーン関数の「スパースモデリング」技術 [1]を実装した SparseIR.jl [2]を紹介する。本ライブラリでは、疑似四倍精度浮動小数点演算による特異値分解を活用し、悪条件の積分方程式を解くことで、虚時間グリーン関数の最適な正規直交基底を構成する。これにより、虚時間形式に基づく場の量子論計算を大幅に高速化することができる [2]。後半では、桁違いに異なる複数の長さスケールが共存する関数を圧縮可能な **quantics (quantized) tensor train (QTT)** [3]を紹介する。近年、QTT は、流体力学[4]や量子多体計算[5]など、物理学の様々な分野で応用されている。テンソルネットワーク、QTT を解説した後、品岡等が開発する実験的なライブラリのデモンストレーションを行う。このライブラリを用いていることで、QTT によって圧縮されたデータのフーリエ変換、畳み込みなどの演算が簡便に行える。量子物理学を超えた将来の応用についても議論したい。

[1] H. Shinaoka, J. Otsuki, M. Ohzeki, K. Yoshimi, PRB 96, 035147 (2017); H. Shinaoka et al., SciPost Physics Lecture Notes, 063 (2022). [2] M. Wallerberger, S. Badr, ..., H. Shinaoka, SoftwareX 21, 101266 (2023). [3] I. V. Oseledets, Doklady Math. 80, 653 (2009); B. N. Khoromskij, Constr. Approx. 34, 257 (2011). [4] N. Gourianov et al., Nat. Comput. Sci. 2, 30 (2022). [5] H. Shinaoka et al., Physical Review X 13, 021015 (2023).

題目: 数論における Julia の援用

講演者: 横山俊一 (東京都立大学)

概要: Julia 言語は現在急速に普及しているが、数学分野においては比較的幾何学・解析学の分野においての活用例が多く、代数学の分野においてはまだまだ少ないのが現状である。しかしながら近年では **AbstractAlgebra.jl** などに代表される、非常に汎用性の高いパッケージが充実してきている。本講演では代数系分野における Julia 活用の可能性について触れた後、とくに数論における **Julia native** の数式処理システム開発プロジェクト **NemoCas / OSCAR** について紹介する。

題目: Julia による科学技術計算:大規模並列計算について

講演者: 永井佑紀 (日本原子力研究開発機構)

概要: Julia を使った科学技術計算の方法について具体例を交えながら解説を行う。特に、現在 Fortran を用いて数値計算を行ってきた方が Julia に期待しているであろう点について言及する予定である。Fortran は、高速でありかつスーパーコンピュータ上での大規模並列計算が可能である。一方で、機械学習に関するパッケージはほとんどな

く、Fortran 利用者にとって、既存の数値計算と機械学習を組み合わせるのは非常に難しい。そこで、Julia である。Julia は高速で動作し、MPI 並列計算も Fortran（と同程度かそれ以上に楽に）実装できる。そして、最新の機械学習アルゴリズムを簡単に実装できる。本講演では、Julia による大規模並列計算の方法と、機械学習の方法について述べる予定である。

題目: 素粒子物理における Julia の活用～格子 QCD の大規模計算に向けて～

講演者: 富谷昭夫 (大阪国際工科専門職大学)

概要: 格子 QCD の数値計算周りの話と Julia の利用について話します。

チュートリアル講演

Julia 入門

Satoshi Terasaki (AtelierArith)

2023/07/10

1 / 87

この資料について

- [数学と物理におけるJuliaの活用](#) でのチュートリアル講演資料
- [piever/Remark.jl](#) を使って Markdown 原稿をスライドとして表示

更新日: 2023-07-10T03:59:43.607

本日の資料は [AtelierArith/julia_tutorial](#) にあります。

2 / 87

ここでの Julia とは

- プログラミング言語の一つ
- 2012 年 [Why We Created Julia](#) に発表. 開発は 2009 年ごろから
 - [Jeff Bezanson](#), [Stefan Karpinski](#), [Viral B. Shah](#), and [Alan Edelman](#)
 - 冒頭で In short, because we are greedy. とあるように既存のプログラミング言語の良いところを取り入れた言語
- 個人的に好きな箇所
 - We want the speed of C with the dynamism of Ruby
 - We want something as usable for general programming as Python
 - Something that is dirt simple to learn, yet keeps the most serious hackers happy.
 - (Did we mention it should be as fast as C?)
- 要するに
 - 高級言語のように使いやすく計算機の上で高速に動作する
 - 速く動くコードを早く書くことができる

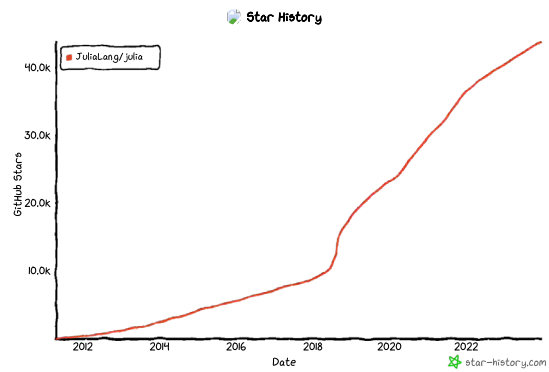
3 / 87

Why We Use Julia, 10 Years Later

- 2022 年は 10 周年記念 🎉 [Why We Use Julia, 10 Years Later](#) が公開
 - [JuliaLang/www.julialang.org](#) リポジトリで管理されている
 - [この原稿](#) にみんなが集まって書いたもの
 - 例: miguelraz さんの [Pull Request](#) など
- Julia というプログラミング言語に出会ったきっかけが紹介されている. 例えば以下のようなもの:
 - [Why We Created Julia](#) を読んでソースをビルドした
 - 同僚、職場、講義で出会った
 - C/C++/Fortran は難しい..., Python は簡単に使えるけれど遅い...
 - 気づいたら特定のパッケージのメンテナになっていた

4 / 87

Star History



5 / 87

直近の動き

- Julia 1.9 がリリース
 - TTFX 問題が改善される !!!
 - 可視化ツールの使用が捗る !
- デバッグまわりのツールが増えてきた
- 鈍器(褒め言葉)扱いの 実践Julia入門, Juliaプログラミング大全が登場してきた
- いろんな本が今年には出版されるらしい

6 / 87

時代は Julia なのでは？

そこでチュートリアル講演です

概要: ライトユースにも使いたいけれどもそれで高速性を犠牲にしたくない。総合的に開発できるものにしたい。超初心者にも習得は容易でありながらハッカーの満足にも応えられるものがほしい。」そういった願いからプログラミング言語 Julia が誕生しました。直近では v1.9 がリリースされ長年課題であったパッケージの読み込み時間の短縮など開発体験の改善が行われました。さらに和書の入門書も充実してきており学習を始めるには絶好のタイミングといえるでしょう。本チュートリアル講演では Julia の入門として

1. インストールと基本的な使い方 (← このスライドの担当箇所)
2. 型と多重ディスパッチ
3. スレッド並列の基礎

3つのトピックを解説します。この機会に Julia に入門し新しい体験を始めてみませんか。

7 / 87

Julia のインストール

8 / 87

Julia のインストール

- 要するに `julia` というコマンドが利用できれば良い

```
$ julia
┌───┐
│   │ Documentation: https://docs.julialang.org
│   │
│   │ Type "?>" for help, "?>" for Pkg help.
│   │
│   │ Version 1.9.1 (2023-06-07)
│   │ Official https://julialang.org/ release
│   │
└───┘
julia>
```

- すでに使える人はスキップする

素直な方法

- Julia の公式サイトから入手 <https://julialang.org/downloads/> 移動し [各自の環境に合わせて導入](#)

9 / 87

それはそうだが

- 任意の人間が「各自の環境に合わせて導入してください」ができれば苦労しない
- 各自の環境が何とか、環境変数とかパスを通すとか
- バージョンが上がる度に手動でインストールしたいか?

10 / 87

それはそうだが

- 任意の人間が「各自の環境に合わせて導入してください」ができれば苦労しない
- 各自の環境が何とか、環境変数とかパスを通すとか
- バージョンが上がる度に手動でインストールしたいか?

そんなあなたに Juliaup



11 / 87

[Juliaup - Julia version manager](#) を使う方法 (1)

Windows

パワーシェルを開く.

```
PS> winget install julia -s msstore
```

[Windows アプリケーションから](#) も入手ができる



12 / 87

Juliaup - Julia version manager を使う方法 (2)

[juliaup/deploy/shellscrip_t/juliaup-init.sh](#) を叩いている。

Mac and Linux

```
$ curl -fsSL https://install.julialang.org | sh
$ source ~/.bashrc
$ julia --version
julia version 1.9.1
```

--yes オプションを使えば途中の対話操作を省略し進めることができる。[command-line-arguments](#) をみよ。

```
$ curl -fsSL https://install.julialang.org | sh -s -- --yes
```

13 / 87

動作確認

インストールができたと仮定して進める。julia というコマンドを使うことができるか確認をする:

```
$ julia --version
julia version 1.9.1
```

何も指定しない場合 REPL(Read-Eval-Print Loop) が起動する:

```
$ julia
┌───┴───┐
┌───┴───┐ | Documentation: https://docs.julialang.org
┌───┴───┐ |
┌───┴───┐ | Type "?>" for help, "j?" for Pkg help.
┌───┴───┐ |
┌───┴───┐ | Version 1.9.1 (2023-06-07)
┌───┴───┐ | Official https://julialang.org/ release
┌───┴───┐ |
julia>
```

14 / 87

Appendix: Julia 自体のアップデート

- [Julia v1.9.2 has been released](#) というアナウンスが出た
- Juliaup 経由で入れていると次のような通知が出る

```
$ julia
The latest version of Julia in the `1.9` channel is 1.9.2+0.x64.apple.darwin14. You currently have `1.9.1+0.x64.apple.darwin14` installed. Run:
juliaup update
to install Julia 1.9.2+0.x64.apple.darwin14 and update the `1.9` channel to that version.
```

```
$ juliaup update
Installing Julia 1.9.2+0.x64.apple.darwin14
Downloading: [->          ] 1.35 MiB/114.18 MiB eta: 14m
```

15 / 87

Appendix: Juliaup 以外の選択肢

- [abelsiqueira/jill](#)
 - jill - Julia Installer 4 Linux - Light
- [johnnychen94/jill.py](#)
 - pip3 install jill && jill install
- [Docker](#) を用いて `docker run -it --rm julia:1.9.1` のようにしてコンテナを起動する

16 / 87

参考資料

- 1.9 系が現時点で最新安定版。もし 1.10 がリリースされると 1.9 系のサポート（バグ修正など）はしなくなる。
- 特定のバージョンを長く使いたい場合は LTS (1.6 系) を使うと良い
- [Julia's Release Process](#)

17 / 87

Julia を動かす (REPL)

18 / 87

なぜ REPL で動かすのか?

もちろんファイルにコードを記述し次のように動かしても良い。

```
$ julia script.jl
```

- Julia は JIT コンパイルによって動作する言語
- `julia script.jl` することにパッケージのロード関数のコンパイルが行われるそのためのコストは無視できない
 - コンパイル結果を使い回し効率よく作業する必要がある
- 試行錯誤時は REPL の上で作業するのがよく行われる
 - REPL の上でだけで学べることが多い
 - Python と異なりインデントに関してセンシティブではないのでコードをコピー&ペーストにより自由に行きわたらせることができる
- もちろん VS Code, Jupyter, Pluto などのリッチな環境でも良い

19 / 87

VS Code を使いたい場合

- [VS Code のホームページ](#)
- [エクステンションの導入](#)
- [いくつか便利なショートカットキー](#)
 - ファイルにフォーカスを当てて Shift + Enter を押す
 - Alt-J Alt-O で REPL を開く

Pluto.jl 使いたい場合

- [Pluto.jl 入門](#)
 - JuliaTokai で話した勉強会資料

20 / 87

Julia を動かす (REPL)

versioninfo() を使うと詳細な情報を得ることができる。バグレポートなどに添付すると良い。

```
$ julia
┌───┐
│   │ │   │ │ Documentation: https://docs.julialang.org
│   │ │   │ │
│   │ │   │ │ Type "?>" for help, "j?" for Pkg help.
│   │ │   │ │
│   │ │   │ │
│   │ │   │ │ Version 1.9.1 (2023-06-07)
│   │ │   │ │ Official https://julialang.org/ release
│   │ │   │ │
└───┘

julia> versioninfo()
Julia Version 1.9.1
Commit 147bdf428cd (2023-06-07 08:27 UTC)
Platform Info:
  OS: macOS (x86_64-apple-darwin22.4.0)
  CPU: 16 × Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-14.0.6 (ORCJIT, skylake)
  Threads: 1 on 16 virtual cores
```

21 / 87

Julia を動かす (REPL)

22 / 87

REPL を使う (Julian mode)

```
julia> 1 + 1 # 簡単な算数
2

julia> println("Hello World")
Hello World

julia> print("Hello"); print(" "); print("World") # セミコロンで繋げることもできる
Hello World
julia>
```

julia> の部分はプロンプトと呼ばれる。julia> の部分も含めてコピーしても REPL 側が適切に処理する。

```
julia> 1 == 2
true # 実際は false なので実際に動かすと false になる
julia> println("Hello")
World # 実際は Hello が出るはず
```

- Python と異なりインデントに関してセンシティブではない
- コードを適当にコピーして自由に実行させることができる

23 / 87

Example

正の整数 a, b を選ぶ、 $\text{gcd}(a, b)$ が 1 となる確率が $1/\zeta(2) = 6/\pi^2$ となることを数値計算で確認する様子

```
julia> function calcπ(N)
    cnt = 0
    for a in 1:N # @は \in と入力する、`in` でも良い
        for b in 1:N # `in` の代わりに `=` と書いても良い
            if gcd(a, b) == 1 # gcd(a, b) == 1 && (cnt += 1) とすることもできる
                cnt += 1
            end
        end
    end
    prob = cnt / N / N
    return √(6*prob) # √ は \sqrt と入力
end
calcπ (generic function with 1 method)
julia> @time calcπ(10^4)
1.791594 seconds
3.141534239016629 # 円周率に近い値
```

(上と同じコードを Python で書いてみると面白いかも。少なくとも数十秒かかる。)

24 / 87

REPL を使う (Help mode)

- 例えば x^y , x_x , $\hat{0}$, \surd のような文字も使うことができる
- どう入力すべきなのか? REPL が教えてくれる!!!

```
julia> # ここで?を押す
help?>  $\hat{0}$ 
" $\hat{0}$ " can be typed by '\theta<tab>\hat<tab>
```

通常の Julian mode で呼び出すこともできる。

```
julia> (Base.Docs.doc)((Base.Docs.Binding)(Main,  $\hat{0}$ ))
```

上記のコードの代わりに下記のようにして @doc マクロを使うこともできる：

```
julia> @doc  $\hat{0}$ 
```

25 / 87

寄り道(マクロについて)

マクロは呼ばれる式の断片を受け取り新しい式を生成する。

例えば @doc $\hat{0}$ は (Base.Docs.doc)((Base.Docs.Binding)(Main, $\hat{0}$)) を生成する。その様子は次のようにして確認できる：

```
julia> @macroexpand @doc  $\hat{0}$ 
:((Base.Docs.doc)((Base.Docs.Binding)(Main,  $\hat{0}$ )))
```

入力の仕方の他に、関数の使い方を知ることができる：

```
help?> ndims
ndims(A::AbstractArray) -> Integer
Return the number of dimensions of A.
See also: size, axes.
Examples
=====
julia> A = fill(1, (3,4,5));
julia> ndims(A)
3
```

26 / 87

寄り道(マクロについて)

1 の原始 3 乗根 $\omega = \exp(2\pi\sqrt{-1}/3)$ をそのままコードに落とし込むと次のようになる:

```
julia> ω = exp(im * 2π/3) # im は虚数単位
-0.4999999999999998 + 0.8660254037844387im
julia> @show ω
ω = -0.4999999999999998 + 0.8660254037844387im
-0.4999999999999998 + 0.8660254037844387im
julia> ω^3 # 3 乗するとほぼほぼ 1 になる。
0.9999999999999998 - 6.106226635438361e-16im
julia> @assert ω^3 ≈ 1 # "*" can be typed by \approx<tab>
```

マクロを使わないと次のようなコードを書いていることになる:

```
julia> ω = exp(im * 2π/3)
-0.4999999999999998 + 0.8660254037844387im
julia> println("ω = ", repr(begin dummyvariable = ω end)); dummyvariable
ω = -0.4999999999999998 + 0.8660254037844387im
-0.4999999999999998 + 0.8660254037844387im
julia> if ω^3 ≈ 1
nothing
else
Base.throw(Base.AssertionError("ω^3 ≈ 1"))
end
```

27 / 87

寄り道(フォントの問題)

文字化けする場合は入力に用いるフォントを変更するとよい。例えば次の候補がある:

- [cormullion/juliamono](#)
- [tohgarashi/JuGeM](#)
- [yuru7/juisee](#)
- [miton/Cica](#)

例えば macOS の場合 `~/Library/Fonts/` 直下に .ttf 拡張子のファイルを配置する。

```
julia> using Plots; plot(sin, label="三角関数", title="日本語", fontfamily="JuGeM-Regular")
```

28 / 87

Pkg mode

パッケージマネージャが付属している。

```
julia> using Pkg; Pkg.add("Example") # Julia.Lang/Example.jl をインストールする
julia> using Example; Example.hello("World")
julia> hello("Example")
julia> Pkg.rm("Example") # アンインストール
```

[Pkg mode](#) を用いて次のようにして書くこともできる:

```
julia> # ] を入力する
(@v1.9) pkg> add Example
(@v1.9) pkg> ^C # Ctrl と C を同時に押す
julia> # 元に戻る
```

29 / 87

寄り道(Example パッケージについて)

上記のコードは実際に動作する例。Example.jl の実装は概ね次のようになっている。詳細は [こちら](#) を参照せよ。

```
module Example
export hello, domath

hello(who::String) = "Hello, $who"

domath(x::Number) = x + 5

end
```

export hello と宣言しているので using Example によって hello を即時使うことができる。

```
julia> using Example
julia> hello("World")
"Hello, World"
julia> domath(3)
8
```

30 / 87

名前空間の話

- `export name` によってユーザが `name` を使うことができる
- 便利である一方「この関数はどこで定義しているのかぱっと見わからない」問題がある

```
julia> using Example: hello
julia> hello("World") # hello がロードされていることがわかる
julia> domath(3) # これはできない
julia> Example.domath(3) # このように使う
```

```
julia> using Example: Example
julia> import Example
julia> Example.hello("World")
```

その他下記のようにしてヒントを得ることができる

```
julia> @doc hello("world")
julia> @which hello("world")
julia> @less hello("world")
```

31 / 87

TerminalClock.jl

Example.jl は簡単すぎるかもしれない？

```
julia> using TerminalClock: clock()
```

- [JuliaPackaging/Preferences.jl](#) によって時計の文字盤を制御することができる:

```
$ julia -q
julia> using TerminalClock
julia> tomfile = joinpath(dirname(pathof(TerminalClock)), "dials", "UnicodeBox.toml");
julia> TerminalClock.set_dials(tomfile)
julia> exit() # REPL を再起動
$ julia -q
julia> using TerminalClock: clock()
```

32 / 87

Search mode

- REPL で入力した過去の履歴を検索できる。~/julia/logs/repl_history.jl に残っている
- 各々 Ctrl-R, Ctrl-S でサーチができる
- 入力途中の式 + Ctrl-P で以前入力したものを補完してくれる

[OhMyREPL.jl](#) を使う場合

[KristofferC/OhMyREPL.jl](#) を使うと直感的に探すことができる:

```
julia> using OhMyREPL
julia> 1+1 # シンタックスハイライトされる
julia> # Ctrl-R で直感的な入力履歴を参照できる
```

33 / 87

Julia で開発する際のワークフロー

34 / 87

Julia で開発する際のワークフロー

- REPL の機能に触れたので Julia での開発ワークフローについて触れる
- 知るところで人生が豊かになる (かも)

35 / 87

REPL 周り学習リソース

2022 Workshop

Julia REPL Mastery Workshop | Miguel Raz Guzmá...



Revise.jl の使い方v1.9 で追加された機能についての解説も

Julia: Workflow tips, and what is new in v1.9 | 2nd D...



36 / 87

Project.toml/Manifest.toml

- 大抵のプログラムは何かしらのパッケージ(ライブラリ)に依存
 - ユーザは何をインストールすればいいかを知りたい
- Project.toml にて使用するパッケージを記述
 - Manifest.toml はより詳細な依存関係の情報を格納 (Project.toml から自動生成される)
- Pkg.activate("path/to/Project.toml")
 - プロジェクト(環境)をアクティベート(活性化)する。
- Pkg.instantiate()
 - Project.toml から Manifest.toml を作る。
 - Manifest.toml から依存関係を解決
- JuliaProject.toml と書くこともできる (混乱がなければ Project.toml でよい) .

37 / 87

例: 研究会の日程, 講演者のアブストを確認する (1)

```
$ cd julia_tutorial/playground/code/mdtable
```

table.jl を動かすための依存関係を解決

```
$ julia -e 'using Pkg; Pkg.activate("."); Pkg.instantiate()'
```

--project, または --project=@. で現在のディレクトリまたは親のディレクトリにある Project.toml から定まるプロジェクトをアクティベートする。 [Stack Overflow](#) での解説も参考

```
$ julia --project=@. -e 'using Pkg; Pkg.instantiate()'
```

38 / 87

例: 研究会の日程, 講演者のアブストを確認する (2)

特定の環境で実行する際にも `--project` または `--project=@` が必要.

```
$ julia --project=@. table.jl 0
```

`--project=@` を指定するのは面倒なので環境変数を設定するファイル (`.bashrc` や `.zshrc`) のなか

```
export JULIA_PROJECT=@.
```

を入れておくと幸せになれる.

39 / 87

試行錯誤の方法

```
script.jl を書く  
julia script.jl を実行する # ( ` ` ) 起動に時間がかかる 😞  
script.jl を更新する  
julia script.jl を実行する # ( ` ` ) 起動に時間がかかる 😞
```

JIT コンパイルが毎回走るので (人間にとって) 効率が悪い, コンパイル結果を使い回す運用が必要.

`mylib.jl` 内部に `main` 関数があるとする.

```
julia> include("mylib.jl")  
julia> main()  
# 別のターミナルで作業して mylib.jl を編集  
julia> include("mylib.jl")  
julia> main()  
# 別のターミナルで作業して mylib.jl を編集
```

これでも良いが `Revise.jl` を使うと良い (次のページへ)

40 / 87

Revise.jl

mylib.jl 内部に main 関数があるとする。

```
julia> using Revise
julia> # includet は Revise から export されている
julia> includet("mylib.jl")
julia> main()
```

これで mylib.jl を変更すれば変更後の main() が実行できる。パッケージの開発でも同様。

using Revise; includet("mylib.jl") のようにセミコロン ; で繋げて書くと 1 行でセットアップができるので作業効率が良い。

- VS Code にて Alt-J Alt-O にて REPL を開くと Revise がすでにロードされている
 - Alt-J Alt-W でウォッチするとわかる

41 / 87

Debugger.jl

- 文字通りデバッガ
- 1 行 1 行実行しその都度変数の状態を監視できる
- VS Code でもできるが REPL の上でもできる

```
julia> using Debugger
julia> include("mylib.jl")
julia> @enter main()
```

- [VS Code での使い方はこちら](#)

42 / 87

JET.jl

- 型不安定なコードや潜在的なエラーを検知ができる。
 - 開発のモチベーションは [aviatesk/grad-thesis](#) などから知ることができる
 - 日本語で読むことができる

43 / 87

型安定・型不安定の話 (1)

- Julia は JIT コンパイル方式で動作をする
 - Julia は関数の引数に渡された値の型の情報を元にコンパイルをする
 - 入力から出力の型が決定できれば型安定な実装となり効率の良いコードを生成することができる
 - 入力される値によって出力の型が変わると型不安定になる。速度が求められる箇所では型安定なコードを書くのが必須です。
- 型安定なコードを書くように意識すると
 - 型システムに親しめるようになる
- 型システムに親しめるようになる
 - 型安定なコードを書くように意識する
- 型システムについてはチュートリアル第二部で行います。（このスライドは第一部です）

44 / 87

型安定・型不安定の話 (2)

- [What does "type-stable" mean?](#)
- [Type annotation make JIT compile faster?](#)

上記の質問に対する Stefan Karpinski さんの回答:

No. You do not generally need type annotations on function arguments (except to control behavior via dispatch), nor do you need type annotations in local scope. The place that type annotations are essential for performance is on locations: the fields of structs and the element types of arrays and other data structures.

- 実際 [Argument-type declarations](#) があるように関数の引数に対して型アノテーションが必要な理由は Dispatch, Correctness, Clarity の3つであって、実行速度の理由で必要とするわけではない。

45 / 87

型安定・型不安定の話 (3)

- @code_XXX 系のマクロで Julia がどのようにコードを理解しているか観測ができる
- @code_XXX 系のマクロの説明は Stack Overflow での議論

[What is the difference between @code_native, @code_typed and @code_llvm in Julia?](#) の解説がわかりやすい。

46 / 87

型不安定な例 (1)

ReLU (rectified linear unit) の例が典型的

```
relu(x) = x > 0 ? x : 0
```

上記のコードは下記と等価

```
function relu2(x)
  if x > 0
    return x
  else
    return 0
  end
end
```

これは @code_lowered で確認することができる。 playground/code/code_comparison のツールを用いて比較ができる。

47 / 87

型不安定な例 (2)

```
relu(x) = x > 0 ? x : 0
```

x の型 T が Float64 の場合, x の値によって Int 型である 0 を返すかもしれないし T を型とする x 自身を返すかもしれない。この曖昧さが型不安定を引き起こす。

48 / 87

型不安定な例 (2)

```
relu1(x) = x > 0 ? x : 0
```

x の型 T が `Float64` の場合、 x の値によって `Int` 型である `0` を返すかもしれないし T を型とする x 自身を返すかもしれない。この曖昧さが型不安定を引き起こす。

処方箋

`0` の代わりに `zero(x)` とする。これは x の型に応じた適切なゼロ(加法単位元)を与えてくれる。

Get the additive identity element for the type of x

```
relu3(x) = x > 0 ? x : zero(x)
```

このようにすると x の入力型が T だった場合、常に T を型とする出力を与える実装になる(型安定になる)。

49 / 87

型安定な例 (1)

```
relu3(x) = x > 0 ? x : zero(x)
```

入力 x は数学的には実数 \mathbb{R} の要素であることに注意する。これは x の型 T が Julia の `Real` のサブタイプであることを要請する:

```
relu4(x::Real) = x > 0 ? x : zero(x)
relu5(x::T) where {T<:Real} = x > 0 ? x : T{0}
relu6(x::T) where {T<:Real} = x > 0 ? x : zero(T)
```

`relu4`, `relu5`, そして `relu6` は等価

Julia 内部では次のような実装になっている (@less zero(1.0) などで検証せよ):

```
zero(x::Number) = oftype(x, 0)
oftype(x, y) = convert(typeof(x), y)
zero(::Type{T}) where {T<:Number} = convert(T, 0)
convert(::Type{T}, x::Number) where {T<:Number} = T(x)::T
```

50 / 87

型安定な例 (2)

- @code_llvm relu4(rand()) の結果

```
: @ string:1 within `relu4`
define double @julia_relu4_2276(double %0) #0 {
top:
  %inv = fcmp ogt double %0, 0.000000e+00
  %l = select i1 %inv, double %0, double 0.000000e+00
  ret double %l
}
```

- @code_llvm relu5(rand()) の結果

```
: @ string:2 within `relu5`
define double @julia_relu5_2278(double %0) #0 {
top:
  %inv = fcmp ogt double %0, 0.000000e+00
  %l = select i1 %inv, double %0, double 0.000000e+00
  ret double %l
}
```

51 / 87

型不安定・型安定なコードの比較 (1)

```
using Random
***
型不安定なコード
***
function main1(N=10)
  rng = MersenneTwister(0)
  s = 0.0
  arr = [relu1(2rand(rng) - 1) for _ in 1:N] # arr は Vector{Real} になる
  for y in arr
    s += y
  end
  s
end
***
型安定なコード
***
function main3(N=10)
  rng = MersenneTwister(0)
  s = 0.0
  arr = [relu3(2rand(rng) - 1) for _ in 1:N] # arr は Vector{Float64} になる
  for y in arr
    s += y
  end
end
```

52 / 87

型不安定・型安定なコードの比較(2)

ほんのちょっとした気遣いで10倍コードが速くなる・ちょっとした怠けで10倍遅くなる。

```
julia> using BenchmarkTools
julia> N = 10000;
julia> @benchmark main1($N) # 型不安定
BenchmarkTools.Trial: 1391 samples with 1 evaluation.
Range (min ... max): 3.283 ms ... 7.546 ms | GC (min ... max): 0.00% ... 44.61%
Time (median): 3.403 ms | GC (median): 0.00%
Time (mean ± σ): 3.590 ms ± 595.631 μs | GC (mean ± σ): 4.67% ± 10.10%
Histogram: log(frequency) by time
Memory estimate: 3.83 MiB, allocs estimate: 149983.
julia> @benchmark main3($N) # 型安定
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
Range (min ... max): 214.528 μs ... 9.073 ms | GC (min ... max): 0.00% ... 96.93%
Time (median): 249.382 μs | GC (median): 0.00%
Time (mean ± σ): 335.062 μs ± 635.965 μs | GC (mean ± σ): 24.49% ± 12.41%
Histogram: log(frequency) by time
Memory estimate: 800.88 KiB, allocs estimate: 14.
```

53 / 87

型不安定・型安定なコードの比較(3)

code_warntype, JET.report_optなどで検出する。対応するマクロもある。

```
using JET

# 色々警告が出る。REPLだと警告は赤色で表示される
code_warntype(main1, (Int,))
@code_warntype main1(10) # (InteractiveUtils.code_warntype(main1, (Base.typosof(10))))
report_opt(main1, (Int,))
@report_opt main1(10) # JET.report_opt(main1, (Base.typosof(10)))

code_warntype(main3, (Int,))
@code_warntype main3(10)
report_opt(main3, (Int,))
@report_opt main3(10)
```

54 / 87

「型安定・型不安定の話 (2)」の続き

- [Type annotation make JIT compile faster?](#)

上記の質問に対する Stefan Karpinski さんの回答:

No. You do not generally need type annotations on function arguments (except to control behavior via dispatch), nor do you need type annotations in local scope. The place that type annotations are essential for performance is on locations: the fields of structs and the element types of arrays and other data structures.

要するに

- 構造体のフィールド
 - パフォーマンスの観点からは `Vector{Real}` ではなく `Vector{Float64}` とするのが良い
- 配列をはじめとするデータ構造に関する要素型

55 / 87

構造体: アフィン変換の例 (1)

アフィン変換 $f(x) = Wx + b$ を考える

```
julia> struct Affine
    W
    b
end

julia> (aff::Affine)(x) = aff.W * x + aff.b # Python での `def __call__` 相当

julia> aff = Affine(rand(2,2), rand(2))
Affine{0.05221655288726834 0.4531635136343529, 0.006660217186564732 0.32819491295200176}, [0.16834154818251978, 0.6642250801

julia> aff(x)
2-element Vector{Float64}:
 0.24022734713529842
 0.7082428857823445
```

動いてそう。

56 / 87

構造体: アフィン変換の例 (2)

人間のからすると W は行列, b はベクトルということはわかるが, Julia 側からはわからない:

```
julia> # 前のスライドからの続き
julia> typeof(aff)
Affine # フィールドの型情報が見えない

julia> @code_warntype aff(x)
MethodInstance for (::Affine){::Symbolics.Array{Num, 1}}
from (aff::Affine)(x) @ Main REPL[2]:1
Arguments
  aff::Affine
  x::Symbolics.Array{Num, 1}
Body::Any
┌── %1 = Base.getproperty(aff, :W)::Any
│   %2 = (%1 * x)::Any
│   %3 = Base.getproperty(aff, :b)::Any
│   %4 = (%2 + %3)::Any
└── return %4
```

フィールドの型情報が見えないので Any がいっぱい. (パフォーマンスの観点からよくない)

57 / 87

構造体: アフィン変換の例 (3)

- フィールドに型をつけたいじゃないか?
- 一旦 REPL を再起動して下記を動かす

```
julia> struct Affine
    W::Matrix{Float64}
    b::Vector{Float64}
end

julia> (aff::Affine)(x) = aff.W * x + aff.b

julia> aff = Affine(rand(2,2), rand(2))
Affine([0.6711341109848074 0.24382214292980786; 0.6036132680677607 0.13610672620163822], [0.2609069147126297, 0.4285554314320])

julia> aff(rand(2))
2-element Vector{Float64}:
 0.9884096898337165
 1.04891268376067
```

動いてそう

58 / 87

構造体: アフィン変換の例 (4)

良さそうに見える

```
julia> @code_warntype aff(rand(2))
MethodInstance for (::Affine)::Vector{Float64}
from (aff::Affine)(x) @ Main REPL[2]:1
Arguments
  aff::Affine
  x::Vector{Float64}
Body::Vector{Float64}
┌── %1 = Base.getproperty(aff, :W)::Matrix{Float64}
│   %2 = (%1 * x)::Vector{Float64}
│   %3 = Base.getproperty(aff, :b)::Vector{Float64}
│   %4 = (%2 + %3)::Vector{Float64}
└──      return %4
```

59 / 87

構造体: アフィン変換の例 (5)

次の使い方をユーザーには適用できない 😞

```
julia> struct Affine
    W::Matrix{Float64}
    b::Vector{Float64}
end
julia> Wf32 = rand{Float32, 2, 2}; bf32 = rand{Float32, 2};
julia> Affine(Wf32, bf32)
W b typeof
Matrix{Float64} (alias for Array{Float64, 2}) # Float32 で計算したいのに...
julia> using CUDA
julia> Wcu = CUDA.rand{2,2}; bcu = CUDA.rand{2,2}
julia> Affine(Wcu, bcu)
W b typeof
julia> Matrix{Float64} (alias for Array{Float64, 2}) # GPU のリソース使いたかったのに...
julia> W = OffsetArray(rand{2,2}, 0:1, 0:1)
```

アフィン変換は汎用性が高いので多くの場面で"いい感じに"振る舞ってほしい!

60 / 87

構造体: アフィン変換の例 (6)

次のようにしてパラメトリック構造体([Parametric Composite Types](#)) によって定義する:

```
julia> struct Affine{T1, T2}
    W::T1
    b::T2
end
julia> (aff::Affine)(x) = aff.W * x + aff.b
julia> Wf32 = rand(Float32, 2, 2); bf32 = rand(Float32, 2);
julia> aff = Affine(Wf32, bf32);
julia> @code_warntype aff(rand(Float32, 2))
MethodInstance for (::Affine{Matrix{Float32}, Vector{Float32}})::Vector{Float32}
  from (aff::Affine){x} @ Main REPL[7]:1
Arguments
  aff::Affine{Matrix{Float32}, Vector{Float32}}
  x::Vector{Float32}
Body::Vector{Float32}
1 ─ %1 = Base.getproperty(aff, :W)::Matrix{Float32}
   │ %2 = (%1 * x)::Vector{Float32}
   │ %3 = Base.getproperty(aff, :b)::Vector{Float32}
   │ %4 = (%2 + %3)::Vector{Float32}
   └── return %4
```

61 / 87

構造体: アフィン変換の例 (7)

今回のように 2x2, 3x3 程度の小規模の行列では配列のサイズ情報も含めて最適なコードを生成する `StaticArrays.jl` を用いた方法を採用すると良い。

```
julia> using StaticArrays
julia> struct Affine{T1, T2}
    W::T1
    b::T2
end
julia> (aff::Affine)(x) = aff.W * x + aff.b
julia> W = @SMatrix rand(Float32, 2, 2)
2×2 SMatrix{2, 2, Float32, 4} with indices SOneTo(2)×SOneTo(2):
 0.106584  0.369698
 0.182812  0.803011
julia> b = @SVector rand(Float32, 2)
2-element SVector{2, Float32} with indices SOneTo(2):
 0.8716512
 0.4812594
julia> aff = Affine(W, b)
julia> x = @SVector rand(Float32, 2)
julia> aff(x)
```

62 / 87

寄り道 (StaticArrays.jl による計算の様子)

人間が直接 $W[1,1] * x[1] + W[1,2] * x[2]$ のように書き下したのとほぼ同じことをしていることがわかる:

```
julia> # 前のページの続き
julia> @code_typed aff(x)
CodeInfo(
  1 --- %1 = Base.getfield(aff, :W)::SMatrix{2, 2, Float32, 4} # W = aff.W
  | %2 = StaticArrays.getfield(%1, :data)::NTuple{4, Float32} # W.data <- 生のデータにアクセス
  | %3 = Base.getfield(%2, 3, false)::Float32 # W.data[3] == W[1,2]
  | %4 = StaticArrays.getfield(x, :data)::Tuple{Float32, Float32} # data = x.data
  | %5 = Base.getfield(%4, 2, false)::Float32 # x.data[2] == x[2]
  | %6 = StaticArrays.getfield(%1, :data)::NTuple{4, Float32} # data = W.data
  | %7 = Base.getfield(%6, 1, false)::Float32 # W.data[1] == W[1,1]
  | %8 = StaticArrays.getfield(x, :data)::Tuple{Float32, Float32} # x.data
  | %9 = Base.getfield(%8, 1, false)::Float32 # x.data[1]
  | %10 = Base.mul_float(%7, %9)::Float32 # W[1,1] * x[1] 相当の計算
  | %11 = Base.muladd_float(%3, %5, %10)::Float32 # W[1,2] * x[2] + W[1,1] * x[1]
  | %12 = StaticArrays.getfield(%1, :data)::NTuple{4, Float32}
  (中略)
  _____
  _____ return %30
  \s = SVector{2, Float32}
```

63 / 87

寄り道 (ConcreteStructs.jl でサボる)

```
julia> struct Affine{T1, T2}
  W::T1
  b::T2
end
```

における $T1, T2$ は形式的につけるものであって実装の部分としては本質的でない。実装時に人間が意識したくない。

```
julia> using ConcreteStructs # @concrete マクロを提供する
julia> @macroexpand @concrete struct Affine; W; b; end
```

下記のコードと等価

```
struct Affine{__T_W, __T_b} <: Any
  W::__T_W
  b::__T_b
  function Affine(W::__T_W, b::__T_b) where {__T_W, __T_b}
    return new{__T_W, __T_b}(W, b)
  end
end
```

64 / 87

寄り道 (サボり方)

型を意識しなくて済む。

```
julia> using ConcreteStructs; @concrete struct Affine; W; b; end
julia> using StaticArrays
julia> W = @SMatrix rand(2,2);
julia> b = @SVector rand(2)
2-element SVector{2, Float64} with indices SOneTo{2}:
 0.620866754177038
 0.5075755639223726
julia> aff = Affine(W, b)
Affine{SMatrix{2, 2, Float64, 4}, SVector{2, Float64}}{[0.21089312982809838 0.4062630292552363; 0.039518291001160444 0.6804761753...
```

今までの議論から分かるように、サボるにはそれなりの教養が必要。

65 / 87

JET.jl を用いた潜在的なエラーの発見 (1)

```
"""
sumevens(N::Integer)
1 から N までの範囲の偶数を集める。収集した値を全て足し、その値を返却する。
"""
function sumevens(N::Integer)
    N >= 1 || throw(DomainError{N}, "N cannot be less than 1.")
    arr = []
    for n in 1:N
        if iseven(n)
            push!(arr, n) # Python での `arr.append(n)` 相当
        end
    end
    return sum(arr)
end

julia> @assert sumevens(8) == 2 + 4 + 6 + 8 == 20
```

66 / 87

JET.jl を用いた潜在的なエラーの発見 (2)

```
julia> sumevens(1) # おっとっと？  
ERROR: MethodError: no method matching zero(::Type{Any})
```

- $N = 1$ の時は `arr = []` の次にあるループは実質何もしない。一番最後に `sum()` を実行することになる。
- `[]` は `Vector{Any}` を型とする要素数が 0 の配列
 - `Vector{Any}` は任意の値が格納できる。任意の型に対するゼロ元を定義するのができないので Julia ではエラーを返す
 - `no method matching zero(::Type{Any})` が出るのはこのため

```
# sum(arr) は大雑把に言えば下記のようなことをする
```

```
s = <初期値> # これをどう定義するか？  
for a in arr  
    s += a  
end
```

67 / 87

JET.jl を用いた潜在的なエラーの発見 (3)

JET.jl は `sumevens(1)` を実行しなくてもまずい部分を検出してくれる。

```
julia> using JET; report_call(sumevens, (Int,))  
===== 1 possible error found =====  
┌ sumevens(N::Int64) @ Main ./REPL[1]:13  
│ ┌ sum(a::Vector{Any}) @ Base ./reducedim.jl:994  
│ │ ┌ sum(a::Vector{Any}; dims::Colon, kw::Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}) @ Base ./reducedim.jl:994  
│ │ │ ┌ _sum(a::Vector{Any}, ::Colon) @ Base ./reducedim.jl:998  
│ │ │ │ ┌ _sum(a::Vector{Any}, ::Colon; kw::Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}) @ Base ./reducedim.jl:998  
│ │ │ │ │ ┌ _sum(f::typeof(identity), a::Vector{Any}, ::Colon) @ Base ./reducedim.jl:999  
│ │ │ │ │ │ ┌ _sum(f::typeof(identity), a::Vector{Any}, ::Colon; kw::Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}) @  
│ │ │ │ │ │ │ ┌ mapreduce(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{Any}) @ Base ./reducedim.jl:357  
│ │ │ │ │ │ │ │ ┌ mapreduce(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{Any}; dims::Colon, init::Base._InitialValue) @ Bas  
│ │ │ │ │ │ │ │ │ ┌ _mapreduce_dim(f::typeof(identity), op::typeof(Base.add_sum), ::Base._InitialValue, A::Vector{Any}, ::Colon) @ Bas  
│ │ │ │ │ │ │ │ │ │ ┌ _mapreduce(f::typeof(identity), op::typeof(Base.add_sum), ::IndexLinear, A::Vector{Any}) @ Base ./reduce.jl:427  
│ │ │ │ │ │ │ │ │ │ │ ┌ mapreduce_empty_iter(f::typeof(identity), op::typeof(Base.add_sum), itr::Vector{Any}, IterType::Base.HasEltyp  
│ │ │ │ │ │ │ │ │ │ │ │ ┌ reduce_empty_iter(op::Base.MappingRF{typeof(identity), typeof(Base.add_sum)}, itr::Vector{Any}, ::Base.H  
│ │ │ │ │ │ │ │ │ │ │ │ │ ┌ reduce_empty(op::Base.MappingRF{typeof(identity), typeof(Base.add_sum)}, ::Type{Any}) @ Base ./redu  
│ │ │ │ │ │ │ │ │ │ │ │ │ │ ┌ mapreduce_empty(f::typeof(identity), op::typeof(Base.add_sum), T::Tuple{Any}) @ Base ./reduce.jl:367
```

68 / 87

JET.jl を用いた潜在的なエラーの発見 (4)

処方箋は次のとおり、`arr = Int[]` とすればよい:

```
function sumevens(N::Integer)
    N >= 1 || throw(DomainError(N, "N cannot be less than 1."))
    arr = Int[] # この行を修正した
    for n in 1:N
        if iseven(n)
            push!(arr, n)
        end
    end
    return sum(arr)
end
```

`sum(arr)` は大雑把にいえば下記のようなことをする

```
s = 0 # Int 型のゼロ元
for a in arr
    s += a
end
```

すなわち、`arr` の値を総和する配列に対して、空の配列の `sum` は `0` となるのが自然だから `arr = Int[]` が `0` として計算できる

69 / 87

JET.jl 周り学習リソース (JuliaCon)

JET.jl の紹介

JET.jl: The next generation of code checker for Julia | Shuh...



レクチャー動画

Package development: improving engineering quality & late...



70 / 87

高速化周り学習リソース(JuliaCon)

佐藤さんによる Julia高速化の常識・非常識 @ Bio*Pack*athon2022#12

Julia高速化の常識・非常識 @ Bio*Pack*athon2022#12



71 / 87

Cthulhu.jl

- クトゥルフ と発音するらしい
- ローカル変数がどのような型になっているかを観察できる
- 誤解を恐れずに言えば Debugger.jl の型バージョン

Making Abstract Interpretation Less Abstract in Cthulhu.jl | ...



72 / 87

BenchmarkTools.jl

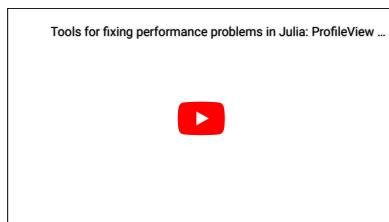
ベンチマークをとることができる

```
julia> @benchmark main1(SN) # 型不安定な実装
BenchmarkTools.Trial: 1410 samples with 1 evaluation.
Range (min ... max): 3.257 ms ... 6.721 ms | GC (min ... max): 0.00% ... 39.38%
Time (median): 3.367 ms | GC (median): 0.00%
Time (mean ± σ): 3.540 ms ± 588.671 μs | GC (mean ± σ): 4.80% ± 10.34%
Histogram: log(frequency) by time
3.26 ms Histogram: log(frequency) by time 5.97 ms <
Memory estimate: 3.83 MiB, allocs estimate: 149983.
julia> @benchmark main3(SN) # 型不安定な実装
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
Range (min ... max): 215.560 μs ... 10.235 ms | GC (min ... max): 0.00% ... 97.30%
Time (median): 249.598 μs | GC (median): 0.00%
Time (mean ± σ): 338.071 μs ± 647.998 μs | GC (mean ± σ): 24.88% ± 12.44%
Histogram: log(frequency) by time
216 μs Histogram: log(frequency) by time 4.59 ms <
Memory estimate: 800.88 KiB, allocs estimate: 14.
```

73 / 87

Profile.jl/ProfileView.jl/ProfileSVG.jl

- どの部分が時間がかかっているかを調べることができる。
- 先ほどの Cthulhu.jl と連携することもできる



74 / 87

Replay.jl

- REPL での作業を自動化する。
- [ドキュメントはこちら](#)
 - hydrodium さんありがとうございます

75 / 87

Pluto.jl 入門

76 / 87

Pluto.jl について

- Julia が動作するノートブック
 - ソースコード: <https://github.com/fonsp/Pluto.jl>
 - 使い方: <https://github.com/fonsp/Pluto.jl/wiki>
- ユーザーが書いたコードが更新されるとセル間の依存関係を自動で解決
- パッケージの依存関係が一つのファイル内に記録される
 - 動作の再現がしやすい(他者と共有しやすい)
- PlutoUI.jl と連携し簡易 UI を構築することができる
- 頑張ると [Introduction to Computational Thinking](#) のようなリッチな教育資源を構築することができる

77 / 87

Pluto.jl の使い方

インストール

```
$ julia -e 'using Pkg; Pkg.add("Pluto")'
```

起動

デフォルトのポート番号は 1234

```
$ julia -e 'using Pluto; Pluto.run()'
```

Docker の Julia 公式イメージから出発すると次のように起動することができる:

```
$ docker run --rm -it -v $PWD:/work -w /work \
-p 1234:1234 \
julia:1.9.2 \
julia -e 'using Pkg; Pkg.add("Pluto"); using Pluto; Pluto.run(host="0.0.0.0")'
```

78 / 87

Demo

- [ソースコードはこちら](#)
- [Lévy C curve](#)
- [反復関数系](#)
- [RandomLogos.jl](#)

79 / 87

豆知識

ローカルにある自作パッケージを Pluto で動かす場合

- `julia --project=@. -c 'using Pluto; Pluto.run()'` で Pluto を起動
- 下記のセルを追加

```
begin
  Pkg.activate(Base.active_project())
end
```

- [terasakisatoshi/Sacabambaspis.jl](#)
- 詳しいことは [Pluto's built-in package management](#) を読むと良い

80 / 87

Pluto.jl 周り学習リソース (JuliaCon)

Pluto.jl の紹介

JuliaCon 2020 | Interactive notebooks ~ Pluto.jl | Fons van ...



開発者らの最近のレクチャー動画

Julia for HPC webinar - Fons van der Plas - Pluto.jl - Reprod...



81 / 87

VS Code (1)

- [julia-vscode/julia-vscode](https://github.com/julia-vscode/julia-vscode)

Julia in VS Code - What's New | David Anthoff, Sebastian P...



82 / 87

VS Code (2)

VS Code 1.80.0 から VS Code の統合ターミナル上で画像を表示できるようになった:

```
using ImageInTerminal
using TestImages
m = testimage("mandril_color")
```

```
using FileIO, Sixel, Plots
gr()
buf = IOBuffer()
show(buf, MIME("image/png"), plot(sin, size=(500, 300)))
buf |> load |> sixel_encode
```

```
using OpenCV
using ImageCore: normedview, colorview, RGB
using ImageInTerminal

img_bgr = OpenCV.imread("sin.png")
img_rgb = OpenCV.cvtColor(img_bgr, OpenCV.COLOR_BGR2RGB)
# Julia のエコシステムが理解できるデータ構造に変換する
jimg = colorview{RGB, normedview}(img_rgb)
# メモリレイアウトの関係上、空間方向の座標が反転していることに注意. 転置する.
```

83 / 87

VS Code (3)

- @testitem によって部分的にテストを実行可能
 - 機能 A はテストが通った
 - 機能 B のデバッグとテストに集中したい
- [JuliaTesting/ReTestItems.jl](#)
- [terasakisatoshi/Sacabambaspis.jl](#)

84 / 87

まとめ

85 / 87

まとめ

- Julia言語の導入, REPL 周りのエコシステムを紹介した
- ここで紹介しきれなかったことはいっぱいある
- 思ったよりも便利な機能があると思います
 - 皆さんの感想を教えてください
- エコシステムが個人や集団からなるコミュニティベースで作られていることがわかる
- Julia言語を多くの人に試してもらい作る側の立場に立てる方が増えれば幸いです

86 / 87

以上

87 / 87

型と多重ディスパッチ

数学と物理におけるJuliaの活用 チュートリアル講演(2)
2023/07/10 後藤俊介

本講義の予定

1. はじめに(5分)
2. 型システムの基本(10分)
3. 複合型と抽象型(10分)
4. 多重ディスパッチの基本(10分)
5. 実習:Julia の多重ディスパッチと型定義(10分)
6. まとめ(5分)

はじめに

講義の目的と目標

- 型(の基本)の理解
 - 型とは何か？
 - Julia の型システム
 - 型の定義
 - 型パラメータ、型制約
- 多重ディスパッチ(の基本)の理解
 - 多重ディスパッチとは？
 - メソッドの選択の仕組み

実験コード

- 本講演資料に記載のサンプルコードを 以下のURLで公開しています。
<https://antimon2.github.io/julia-imi-workshop2023-tutorial/1st-example-type.md.html>
- チュートリアル(1) で Pluto.jl を導入済の方は、上記URL→[Edit or Run] ボタン→「On your computer」の手順に従って手元の環境で実行できます。
- 実習課題も編集可能な状態で用意してありますのでご利用ください。



型システムの基本

型とは何か(1)

プログラミング全般において…

- その値(オブジェクト)が『何』であるかを表すもの。
- (Julia の場合) `typeof()` 関数でその値の型が分かる

```
julia> typeof(1)
Int64
```

```
julia> typeof("文字列")
String
```

```
julia> typeof([1.0, 2.0, 3.0])
Vector{Float64} (alias for Array{Float64, 1})
```

型とは何か(2)

少し概念的な例えとして…

- 物理で言えば『単位』
- 物理:『単位』を意識することで現象や状況を理解
- プログラミング:変数や式の値の『型』を意識することでその式の意味や状況を理解

あなたは型理論と物理学のプロフェッショナルです。
以下の文章を学生にも分かるように分かりやすく説明してください。

プログラミングにおける『型による抽象』は、物理学において『式を単位で抽象的に見る』のと同じと見なせる。

まず、この文の内容を理解するためには、二つの概念、「型による抽象」と「式を単位で抽象的に見る」について理解する必要があります。

『型による抽象』は、プログラミングの世界でよく見かける概念です。プログラミング言語には「型」という概念があります。例えば「整数」、「文字列」、「真」などです。

法を使っています。これは、スーパーマーケットで商品を並べ替えるのと同じように、物の置きかたまりを整理するのと同じです。

もっと簡単に、箇条書きでまとめてください。

もちろんです。以下にその内容を箇条書きにまとめます。

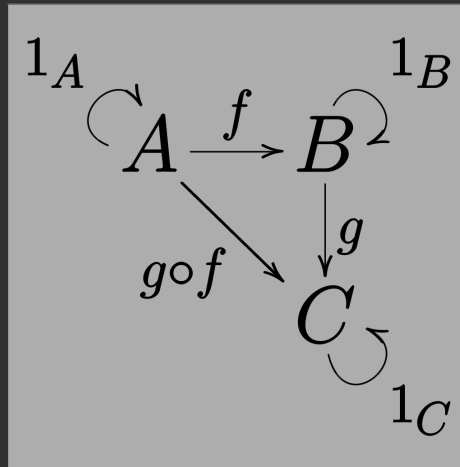
1. プログラミングでは、「型」を使ってデータを分類します。
 - * これは、色を分類するようなものと考えてください。
2. この「型」によって、何ができて何が得られるかを理解します。
3. 物理学では、「単位」を使って物理的現象を理解します。
 - * これは、物の置きかたまりのようなものと考えてください。
4. この「単位」によって、どのような現象が起こり、どのような結果が得られるかを理解します。
5. つまり、プログラミングも物理学も、「ラベル」を使って物事を理解し、予測する方法を使っています。

な状況に対応できるようになります。具体的な事例や問題に立ち向かうための一般的な進め方やフレームワークを提供することで、より効率的で柔軟な解決策を見つけ出すことができます。

型とは何か(3)

さらに抽象的な例えとして…

- 数学(圏論)で言う『対象』



画像は [Wikipediaより引用](#)
A, B, C が『対象』、 f , g , $g \circ f$ は『射』(Julia における関数に対応)

型とは何か(4)

要するに…

- 値(オブジェクト)を分類(もしくはラベル付け)したもの
- その値(オブジェクト)が『どのような性質を持つのか』を表す
- その値に対して『どのような操作ができるのか』を規定する

Julia の型システム(1)

Julia の型システムの基本…

- 公称型システム (Nominative Type System)
 - 名前で型が決まる!
- 公称的サブタイピング (Nominal Subtyping)
 - 明示的な宣言で型の基本型-派生型の関係が決まる!
- `typeof()` 関数、`<:` 演算子、`isa` 演算子等

```
julia> typeof(1) === Int
true # Int という型は唯一
```

```
julia> Int <: Signed <: Integer <: Real <: Number <: Any
true # ↑ `<:` は派生型演算子 (左辺が右辺の派生型なら `true`)
```

```
julia> 1 isa Number
true # `a isa T` は `typeof(a) <: T` と (ほぼ) 等価
```

Julia の型システム(2)

サブタイピングについて…

- (直接の)基本型はただ1つだけ
 - `supertype()` 関数
- 派生型はいくらでもありうる
 - `subtypes()` 関数
- Any型
 - 全ての型の基本型

```
julia> supertype(Int)
Signed
```

```
julia> supertype(String)
AbstractString
```

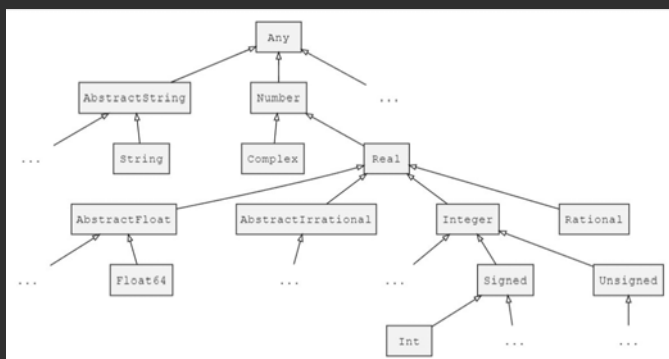
```
julia> subtypes(AbstractString) # 環境によって結果は変わる
6-element Vector{Any}:
 Core.Compiler.LazyString
 LazyString
 String
 SubString
 SubstitutionString
 Test.GenericString
```

```
julia> length(subtypes(Any)) # 環境によって結果は変わる
587
```

Julia の型システム(3)

型階層...

- 型どうしのサブタイピング関係を図にするときれいな階層構造が出現(型階層)
- (Anyを根とした)型階層の図を『型ツリー』とも呼ぶ



画像は拙著 [実践Julia入門](#) より引用

代表的な型の紹介(1)

整数型

- 符号付き整数
 - Int8, Int16, Int32, Int64, Int128
 - Int は Int64 のエイリアス
- 符号なし整数
 - UInt8, UInt16, UInt32, UInt64, UInt128
 - UInt は UInt64 のエイリアス
- 多倍長整数(BigInt)
- リテラル
 - 10進表記なら符号付き、16進表記(など)なら符号なし
 - 桁数や大きさが型が自動的に決まる

```
julia> typeof(4294967296)
Int64
```

```
julia> typeof(9223372036854775808)
Int128
```

```
julia> typeof(170141183460469231731687303715884105728)
BigInt
```

```
julia> typeof(0x01)
UInt8
```

```
julia> typeof(0x0123456789abcdef)
UInt64
```

代表的な型の紹介(2)

浮動小数点数型

- Float16, Float32, Float64
- BigFloat
- リテラルは様々な書式がある

```
julia> typeof(1.0) === typeof(0.) === typeof(-.0) === Float64
true # よくある表記は大抵 Float64
```

```
julia> typeof(1e308) === typeof(0x193p-2) === Float64
true # `0x`《16進表記》p《2の冪数》という書式もOK
```

```
julia> typeof(3.14f0)
Float32 # `xxxfyy` という書式にすると Float32 になる
```

```
julia> Float16(3.1416) # Float16 は明示的に指定
Float16(3.14)
```

```
julia> big(π) # BigFloat の例
3.1415926535897932384626433832795028841971693993751058209749445
92307816406286198
```

代表的な型の紹介(3)

その他の数値型

- 有理数
 - $\circ//\Delta$
- 複素数
 - $\circ + \Delta im$

```
julia> typeof(1//2)
Rational{Int64}
```

```
julia> -12//20
-3//5 # 結果は適宜約分される
```

```
julia> 3//0 # これもOK (`0//0`のみエラー)
1//0
```

```
julia> im # 虚数単位 (定数として定義済)
im
```

```
julia> typeof(1 + 2im)
Complex{Int64}
```

```
julia> typeof(0.0 + 1.0im)
ComplexF64 (alias for Complex{Float64})
```

代表的な型の紹介(4)

文字型と文字列型

- Julia では **文字型** と **文字列型** は別物
- シングルクォーテーションで括ったものは文字型 (Char)
- ダブルクォーテーションで括ったものは文字列型 (String)

```
julia> typeof('a')
Char
```

```
julia> typeof("a")
String
```

```
julia> typeof("""
    複数行にわたる文字列
    これも String 型
    """)
String
```

代表的な型の紹介(5)

配列型

- Julia では(同じ型の)値の列はリストではなく**(1次元)配列**
- Julia の配列は1次元だけでなく**多次元配列**もOK
- 要素の型と次元数は型パラメータ(後述)で表現

```
julia> typeof([1.0, 2.0, 3.0])
Vector{Float64} (alias for Array{Float64, 1})
# リストではなくベクトル (=1次元配列)
```

```
julia> typeof([
    1 2
    3 4
]) # 行列 (=2次元配列)
Matrix{Int64} (alias for Array{Int64, 2})
```

```
julia> typeof([1;2;;3;4;;;5;6;;;7;8])
Array{Int64, 3} # 2×2×2 の3次元配列
```


代表的な型の紹介(6)

辞書型・集合型

- **辞書型(Dict)**: キーと値のマッピングで管理するコレクション
- **集合型(Set)**: 重複を許さない値のコレクション
- 配列型、辞書型、集合型と後述のタプル・名前付きタプルを総称して **コレクション型** と言う

```
julia> Dict{"Alice"=>1, "Bob"=>2, "Carol"=>3}
Dict{String, Int64} with 3 entries:
  "Carol" => 3
  "Alice" => 1
  "Bob"   => 2 # 結果は順不同

julia> Set([3, 1, 4, 1, 5, 9, 2, 6, 5, 3])
Set{Int64} with 7 elements:
 5 # 重複は排除される
 4
 6
 2
 9
 3
 1 # こちらも順不同
```

代表的な型の紹介(7)

タプル型・名前付きタプル型

- **タプル型(Tuple)**: (同じ型とは限らない)値の列
- **名前付きタプル型(NamedTuple)**: タプルの各値に名前(キー)が紐付いているもの(辞書に類似)
- これらは、各値の型も情報として保持するのが特徴

```
julia> typeof((1, 'b', "三"))
Tuple{Int64, Char, String}

julia> typeof((a=1, b='b', c="三"))
NamedTuple{(:a, :b, :c), Tuple{Int64, Char, String}}
```

型アノテーション

- $x::T$ という書式で **型アノテーション** を付けられる
- 宣言時や代入時の左辺の場合は(変数の)型の指定(変換に失敗するとエラー)
- 代入時の右辺などの場合は **型検査** (互換性のない型だとエラー)
- **※型ヒントではない!** (重要)

```
julia> let
    x::Int = sin(pi) # == 0.0 なので変換されて 0 が代入される
    x
end
0

julia> let
    x = (sin(pi)::Int) # こちらはエラー (!(sin(pi) <: Int) なので)
    x
end
ERROR: TypeError: in typeassert, expected Int64, got a value of type Float64
Stacktrace:
 [1] top-level scope
      @ REPL[XX]:2
```

複合型と抽象型

複合型(1) (構造体)

- `struct` 《型名》 ~ `end` で構造体を定義可能
- 構造体として定義された型のことを複合型と呼ぶ
- ※基本的な型以外のほとんどの型は複合型 (有理数型・複素数型、辞書型・集合型・名前付きタプル型も実は複合型)

```
julia> struct SSample
        x
        y
    end

julia> ssample = SSample(1, 2)
SSample{Int64, Int64}(1, 2)

julia> typeof(ssample)
SSample{Int64, Int64}
```

複合型(2) (フィールドの型指定)

- 構造体のフィールドには型が指定できる(型アノテーション)
- ※型アノテーションが付いていないフィールドは実は `::Any` と同じ意味(詳細略)
- 補足: 目的がはっきりしているときはそのフィールドには型アノテーションを付けましょう(詳細後述)

```
julia> struct SISample
        x::Int
        y::Int
    end

julia> sisample = SISample(1, 2)
SISample{Int64, Int64}(1, 2)

julia> typeof(sisample.x)
Int64

julia> SISample(2.0, 3.14) # `3.14` の方でエラーが発生
ERROR: InexactError: Int64(3.14)
# : 《以下略》
```

複合型(3) (基本型の指定)

- 型定義時に基本型を指定 (その型の派生型という宣言)ができる
- ※基本型の指定がない型定義は、<: Any(Any型の派生型の定義)と同じ意味(詳細略)

```
julia> struct MyDecimal <: Real
    value::BigInt
    point::Int
end # 固定小数点数を意図した型定義

julia> MyDecimal(1, 0) isa Number
true

julia> MyDecimal(1, 0) isa Integer
false
```

抽象型(1)

- abstract type 《型名》 end で 抽象型 を定義できる
- 抽象型は他の抽象型や複合型等の基本型にできる (=型階層が作れる)

```
julia> abstract type AbstractFPoint end

julia> struct FPoint2D <: AbstractFPoint
    x::Float64
    y::Float64
end

julia> struct FPoint3D <: AbstractFPoint
    x::Float64
    y::Float64
    z::Float64
end

julia> FPoint2D(1.0, 2.0) isa AbstractFPoint
true

julia> FPoint3D(3,  $\pi$ , 99.9) isa AbstractFPoint
true
```

型パラメータ(1)

- 型定義時に {T} のように型パラメータを指定できる
- 型パラメータの使い途:
 - フィールドの型指定
 - 性質を表すパラメータ(フラグ)(詳細後述)

```
julia> struct TWrapperSample{T}
    value::T
end

julia> intwrapper = TWrapperSample{1}
TWrapperSample{Int64}(1)

julia> typeof(intwrapper)
TWrapperSample{Int64}

julia> typeof(TWrapperSample("文字列"))
TWrapperSample{String}
```

型制約(1)

- 型パラメータ指定時に {T <: Real} のようにその型に制約を設けることができる(型制約)
- 制約に合わない型が来るとエラーになる

```
julia> abstract type AbstractPoint{T <: Real} end

julia> struct Point2D{T} <: AbstractPoint{T}
    x::T
    y::T
end

julia> Point2D(1.0, 3.2) isa AbstractPoint{Float64}
true

julia> Point2D(1 + 0im, 0 + im)
ERROR: TypeError: in AbstractPoint, in T, expected
T<:Real, got Type{Complex{Int64}}
# : 《以下略》
```

我々は何を見せられているのか…

型パラメータ(2) + 型制約(2)

この形 →
は非常によく見るので、
ぜひ覚えておきましょう！

利点:

- フィールドの型を特定しないので柔軟な設計が可能
- 実行時には確定するので **型安定性** に繋がる
- 型制約と組み合わせるとさらに賢い設計に！

```
julia> abstract type AbstractPoint{T <: Real} end
```

```
julia> struct Point2D{T} <: AbstractPoint{T}
    x::T
    y::T
end
```

多重ディスパッチの基本

多重ディスパッチとは(1)

関数の多重定義(1)

- Julia では同名の関数を引数の違いで **多重定義** できる
- 個々の実体のことを **メソッド** と呼ぶ
- 引数の違いとは
 - **引数の個数の違い**
 - **引数の型の違い**
 - **その組み合わせ**

```
julia> add(x, y) = x + y
add (generic function with 1 method)
```

```
julia> add(x, y, z) = x + y + z
add (generic function with 2 methods)
```

```
julia> add(x, y, z...) = add(x + y, z...)
add (generic function with 3 methods)
```

```
julia> methods(add)
# 3 methods for generic function "add" from Main:
 [1] add(x, y)
 [2] add(x, y, z)
 [3] add(x, y, z...)
```

多重ディスパッチとは(2)

関数の多重定義(2)

- Julia では同名の関数を引数の違いで多重定義できる
- 個々の実体のことをメソッドと呼ぶ
- 引数の違いとは
 - 引数の個数の違い
 - 引数の型の違い
 - その組み合わせ

```
julia> double(x) = 2x
double (generic function with 1 method)

julia> double(s::AbstractString) = s ^ 2
double (generic function with 2 methods)

julia> double(x, y) = string(double(x), double(y))
double (generic function with 3 methods)

julia> double(x::Number, y::Number) = double(x) + double(y)
double (generic function with 4 methods)

julia> methods(double)
# 4 methods for generic function "double" from Main:
# : 《略》
```

多重ディスパッチとは(3)

ディスパッチ

- 多重定義された関数は、呼び出し時の実引数の組み合わせで適切なメソッドが選択されて実行(=ディスパッチ)される

```
julia> double("文字列")
# "文字列" isa AbstractString なので s ^ 2 の実装 (メソッド) が選択される
"文字列文字列"

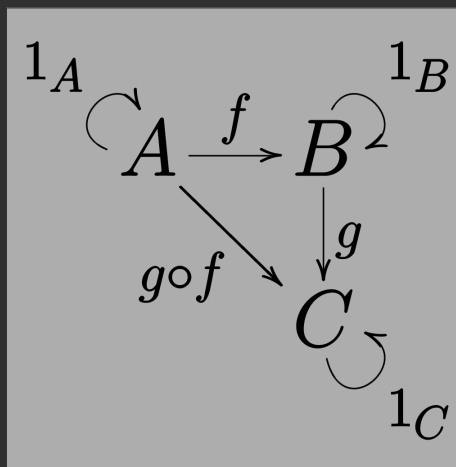
julia> double(1, 2)
# 1 isa Number かつ 2 isa Number なので double(x) + double(y)
6

julia> double(160, "円")
# どうしてこうなるのか考えてみよう!
"320円円"
```


そもそも関数とは？(1)

抽象的な例えとして…

- 数学(圏論)で言う『射(射影)』

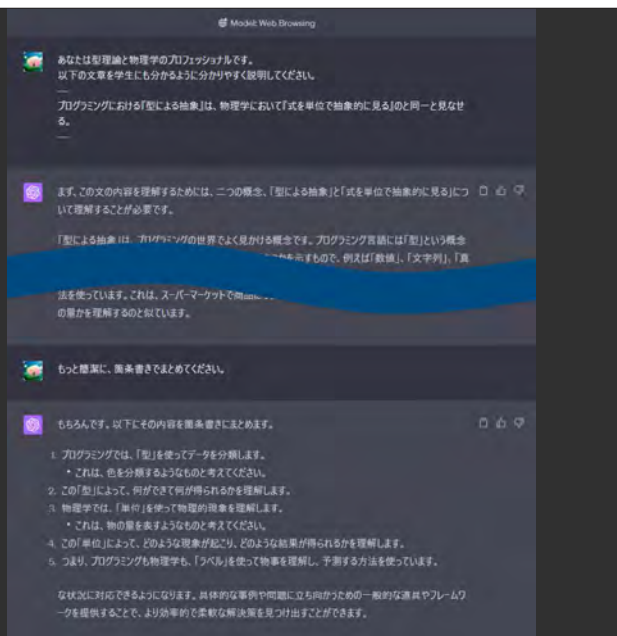


画像は [Wikipediaより引用](#) (再掲)
A, B, C が『対象』、f, g, $g \circ f$ は『射』(Julia における関数に対応)

そもそも関数とは？(2)

少し概念的な例えとして…

- 物理で言えば『式』
- 物理: 似たような『式』でも『単位』を意識することで表される現象や得られる結果が変わる
- Julia: 『関数』に渡される『型』によってその挙動の意味や得られる結果が変わる



多重ディスパッチとは(4)

重要な概念として…

- 関数の名前を固定することで、その関数に『役割』を与える。
 - 物理学において「だいたい同じ形の式で書けるものは似たような現象を表す」ことにだいたい相当
- その関数を型(の組み合わせや個数)の違いで多重定義(メソッドを定義)することで、特定の状況に置ける意味とその結果を与える(実装する)。
 - 同じ式でも『単位』の違いで状況・意味・結果が変わることに類似

Juliaにおける多重ディスパッチの利用例(1)

引数の個数による意味分け

- 同じ関数で、引数の個数によって意味を分ける
- 例: `log()` 関数
- 例: `atan()` 関数

```
julia> log(20) # 自然対数
2.995732273553991
```

```
julia> log(5, 20) # 底を5とする対数
1.8613531161467862
```

```
julia> atan(0.3) # tan(θ) == 0.3 となる θ
0.2914567944778671
```

```
julia> atan(1, 3) # tan(θ) == 1/3 となる θ
0.3217505543966422
```

Juliaにおける多重ディスパッチの利用例(2)

演算子オーバーロード

- 同じ演算子でも両辺(被演算子)の型で挙動(結果の値や型)に違いが出る
- わかりやすい例: * 演算子
 - 数値同士なら乗算
 - 文字列同士なら結合
 - 数値同士でも、型の違いによって適切な処理

```
julia> 1 * 2 # 整数どうしの `*` 演算は結果も整数
2
```

```
julia> 3.0 * π # 浮動小数点数と無理数の `*` 演算は浮動小数点数
9.42477796076938
```

```
julia> "Hello, " * "Julia!" # 文字列同士の `*` 演算は結合
"Hello, Julia!"
```

Juliaにおける多重ディスパッチの利用例(3)

その他の例...

- Holyトレイト
 - 特殊な型による定義分けを利用して、関係ない型での共通の挙動(振る舞い)の定義(実装)を共有化するテクニック
- アルゴリズムの分離・モジュール化
 - ソートアルゴリズムの定義など
- 擬似値ディスパッチ
- ポリモーフィズム

実習: Julia の多重ディスパッチと型定義

お題: `double()`関数(1)

Step1: 型定義

- `MyType` という型(複合型)を定義する
 - フィールドは `value` 1つだけ
 - フィールドの型は型パラメータ `T`
- ※これに関しては答えを示しておきます→

```
julia> struct MyType{T}
    value::T
end
```

```
julia> MyType(1)
MyType{Int64}(1)
```

お題:double()関数(2)

Step2: 関数の多重定義(1)

- 先ほど例示した double() 関数を、MyType 型が扱えるように多重定義する。
- まずは引数に MyType{T} 1つだけを受け取るメソッドの定義:
 - フィールド value に double() 関数を適用した結果を格納する Mytype を返す

```
julia> struct MyType{T}
    value::T
end

julia> MyType(1)
MyType{Int64}(1)

julia> double(mytype::MyType) = ... # ここを適切に実装
double (generic function with 5 methods)

julia> double(MyType(1))
MyType{Int64}(2)

julia> double(MyType("ABC"))
MyType{String}("ABCABC")
```

お題:double()関数(3)

Step3: 関数の多重定義(2)

- double() 関数を、MyType 型が扱えるように多重定義する。
- 引数を2つ(一方が MyType)を受け取るメソッドの定義(難易度高め):
 - 結果は MyType、ただし MyType{MyType{...}} のような入れ子にならないように注意!

```
julia> double(x::MyType, y::MyType) = ... # ここを適切に実装
double (generic function with 6 methods)

julia> double(x::MyType, y) = ... # ここを適切に実装
double (generic function with 7 methods)

julia> double(x, y::MyType) = ... # ここを適切に実装
double (generic function with 8 methods)

julia> double(2, MyType(n))
MyType{Float64}(10.283185307179586)

julia> double(MyType(35), "v")
MyType{String}("70vv")

julia> double(MyType("ABC"), MyType("XYZ"))
MyType{String}("ABCABCXYZXYZ")
```

お題: double()関数(4)

Extra Step(時間が余った人用)

- double() 関数を、他の型にも適用しよう
 - 例: 文字型⇒文字列として扱う
 - 例: コレクション型⇒各要素にdouble()を適用したコレクションにする
 - 他⇒自分で仕様を考えて実装してみよう
- 型パラメータの利用
 - 型パラメータの違いで処理を分ける、など

```
julia> double(x::Char) = ... # ここを適切に実装  
double (generic function with ? methods)
```

```
julia> double(x::Tuple) = ... # ここを適切に実装  
double (generic function with ? methods)
```

```
julia> double(x::Array) = ... # ここを適切に実装  
double (generic function with ? methods)
```

```
julia> double((1, 'b', [1.0, 2.0, 3.0]))  
(2, "bb", [2.0, 4.0, 6.0])
```

```
julia> double(x::MuType{T}) where {T <: Tuple} = ... # ここを適切に実装  
double (generic function with ? methods)
```

```
julia> double(MyType((1, 'b', "三")))  
#> MyType((1, 'b', "三", 1, 'b', "三")) # 一例
```

まとめ

講義の目的と目標(再掲)

- ✓ 型(の基本)の理解
 - ✓ 型とは何か？
 - ✓ Julia の型システム
 - ✓ 型の定義
 - ✓ 型パラメータ、型制約
- ✓ 多重ディスパッチ(の基本)の理解
 - ✓ 多重ディスパッチとは？
 - ✓ メソッドの選択の仕組み

お疲れさまでした

スレッド並列の基礎

数学と物理におけるJuliaの活用 チュートリアル講演(3)
2023/07/10 後藤俊介

本講義の予定

1. はじめに(5分)
2. スレッド並列とは何か(10分)
3. Juliaにおけるスレッド並列の基本(10分)
4. 実例(10分)
5. 実習:並列化によるパフォーマンス改善(10分)
6. 注意点とまとめ(5分)

はじめに

講義の目的と目標

- スレッド並列(の基本)の理解
 - 並列処理とは何か
 - スレッドの基本的な概念
 - スレッドの利点と制限
 - (スレッドのライフサイクル)
- Juliaによるスレッド並列の理解
 - 最低限の事前準備
 - 各種API
 - スレッドセーフ(とデッドロック)

実験コード

- 本講演資料に記載のサンプルコードを 以下のURLで公開しています。
<https://antimon2.github.io/julia-imi-workshop2023-tutorial/1st-example-nthreads.html>
- チュートリアル(1) で Pluto.jl を導入済の方は、上記URL→[Edit or Run] ボタン→「On your computer」の手順に従って手元の環境で実行できます。
- 実習課題も編集可能な状態で用意してありますのでご利用ください。



スレッド並列とは何か

並列処理とは何か(1)

プログラミング全般において…

- **並行計算(Concurrent Computing):**
複数の計算を同時に実行すること
- **並列計算(Parallel Computing):**
細分化された独立な処理 を並行計算すること
- (本講では)並列計算処理のことを指して
並列処理 と呼ぶことにする

並列処理とは何か(2)

なぜ必要かというお話…

- 大規模な計算タスクを効率的に処理するための手段
(開発・研究スピードを上げるため)
- リソースの効率的な利用
(CPUコアを遊ばせずフル活用するため)
- リアルタイムアプリケーションからの需要
(複数のタスクを同時に迅速に処理するため)

並列処理とは何か(3)

アーキテクチャ的な話…

- 一連の処理(タスク)を同時に走らせて処理をすること
- 目的は **スループット向上** (パフォーマンス向上)
- いくつか種類がある:
 - **スレッド並列**
 - **プロセス並列**

```
julia> Threads.nthreads()
6

julia> fib(n) = n ≤ 1 ? n : fib(n - 2) + fib(n - 1)
fib (generic function with 1 method)

julia> @time fib(40)
0.510351 seconds
102334155

julia> @time for _=1:4 fib(40) end
2.073198 seconds (8.14 k allocations: 600.650 KiB, 1.68% compilation time)

julia> @time Threads.@threads for _=1:4 fib(40) end
0.783171 seconds (34.33 k allocations: 2.571 MiB, 19.96% compilation time)
```

スレッド並列とは(1)

まずは用語整理…

- **タスク:** 一連の処理のまとめ(本講では)
- **スレッド:** CPUの最小処理単位
- **プロセス:** プログラムの実行単位

あなたは Julia 言語のプロフェッショナルです。
Julia における、タスク (Task) とスレッド (Thread) とプロセス (Process) の違いを説明してください。

Julia 言語におけるタスク (Task)、スレッド (Thread)、プロセス (Process) の違いについて説明します。

1. **タスク (Task):** Julia では、タスクは **コルーチン (coroutines)** とも呼ばれ、非同期タスクを実行するための抽象化です。タスクは軽量の操作で、一つのタスクがブロックされると、他のタスクが実行を続けることができます。これは、I/O 操作やユーザー入力の特徴など、特定の操作が完了するのを待つ必要がある場合に特に有用です。タスクは一つのスレッド内で動作し、他のタスクとメモリを共有します。
2. **スレッド (Thread):** スレッドはプロセス内の実行単位で、同じメモリ空間を共有します。Julia 1.3 以降、マルチスレッド機能が実験的に導入されました。これにより、複数のスレッドを使用してタスクを並行に実行することが可能になりました。各スレッドは独自の制御フローを持ち、それぞれが異なるタスクを実行できますが、同じプロセス内の他のスレッドとメモリを共有します。
3. **プロセス (Process):** プロセスはオペレーティングシステムがリソース (CPU 時間、メモリなど) を割り当てる基本的な単位です。各プロセスは独自のメモリ空間を持ち、他のプロセスから隔離されています。Julia では、**"Distributed"** パッケージを使用して複数のプロセスを起動し、それらの間でデータを送受信することができます。これにより、複数の CPU コアやマシン上で並行にコードを実行することが可能になります。

これらの概念は、Julia での並行性と並列性の実現に重要な役割を果たします。タスクは非同期的な操作を可能にし、スレッドは共有メモリ内での並行作業を可能にし、プロセスは複数の CPU コアやマシン間での並列作業を可能にします。

スレッド並列とは(2)

- 通常のプログラムでは、単一プロセス・単一スレッドで1つのタスクが動くだけ(シングルプロセス・シングルスレッド)
- 単一プロセス中で複数のスレッドを同時に管理し、複数のタスクを同時実行(≡並列処理)できる(シングルプロセス・マルチスレッド)
- 複数プロセスを同時に立ち上げて複数のタスクを同時実行(≡並列処理)することもできる(マルチプロセス)

スレッド並列とは(3)

- 通常のプログラムでは、単一プロセス・単一スレッドで1つのタスクが動くだけ(シングルプロセス・シングルスレッド)
- 単一プロセス中で複数のスレッドを同時に管理し、複数のタスクを同時実行(≡並列処理)できる(シングルプロセス・マルチスレッド)⇒**スレッド並列**
- 複数プロセスを同時に立ち上げて複数のタスクを同時実行(≡並列処理)することもできる(マルチプロセス)

スレッド並列の利点と制限

- ◎複数タスク同時実行によるスループット向上(パフォーマンス向上)(≒高速化)
- ○単一プロセスなので管理が比較的楽(v.s.マルチプロセス)
- △単一プロセスなのでメモリは共有(プロセス間通信による大きなオーバーヘッドはない)
- ▲(その代わりに)メモリ競合には注意(スレッドセーフ意識せよ、詳細後述)
- ▲デッドロック注意(概要後述)

補足:『真のスレッド並列』

時間があれば(なければ飛ばすので読んでおいてください)...

- 他言語にもスレッド(マルチスレッド)という機能はある
- 一部言語のスレッドは、GIL(Global Interpreter Lock)という機構により並列動作が制限されている
- Julia にはその制限がない(『真のスレッド並列』)
- 他言語で「並列処理するならマルチプロセス」として提案・提唱されている事案でも、Julia ならスレッド並列で間に合う(むしろそちらの方が手っ取り早い)ものも多い
- Julia では「オーバーヘッドによるデメリットよりも『スレッドアンセーフやデッドロックの心配をしなくて済む』というメリットの方が勝る」ような場合には プロセス並列 も選択出来る(選択肢が多い)と建設的に考えることもできる

補足:スレッドのライフサイクル

さらに概念的かつ内部仕様のな話(時間がなければ飛ばすので読んでおいてください)...

- 一般的なスレッドの状態(ステージ):「1. 新規(New)」「2. 実行可能(Runnable)」「3. 実行中(Running)」「4. ブロック(Blocked)」「5. 終了(Terminated)」
- 低レベルでスレッドを扱う必要のある環境では、これらの状態(ステージ)を適切に把握・管理する必要がある
- Julia ではあまり深く考える必要はない
- 必要になったときに Julia が内部でスレッドを立ち上げ(「1. New」)タスクを走らせて(「2. Runnable」「3. Running」)適宜協調動作(「4. Blocked」)しながらタスクが終了したら終了(「5. Terminate」)してくれる
- のでこの状態(ステージ)は、概念として参考までに

Juliaにおけるスレッド並列の 基本

最低限の事前準備

Julia 起動時に最大スレッド数を指定する

- `julia -t 4`
(絶対数指定)
- `julia -t auto`
(利用できるCPUコア最大確保)
- 環境変数による指定も可能
- Pluto の場合デフォルトで「利用できる最大÷2」が設定される

```
> julia -t 4 # 最大スレッド数は4

> julia -t auto # 確保できる分だけスレッド数を確保

> env JULIA_NUM_THREADS=XXX julia # Bash/Zshなどで環境変数指定と同時にJulia起動

# Julia 内では「」で利用可能なスレッド数が取得できる
julia> Threads.nthreads()
6
```

各種API(1)

@threads

- スレッド並列の基本
- `for ~ end` を簡単にマルチスレッド化できる
- 本講ではモジュールプレフィックス付きで `Threads.@threads` という表記を利用(以下同様)

```
julia> Threads.nthreads()
6

julia> fib(n) = n ≤ 1 ? n : fib(n - 2) + fib(n - 1)
fib (generic function with 1 method)

julia> @time fib(40)
0.510351 seconds
102334155

julia> @time for _=1:4
    fib(40)
end
2.073198 seconds (8.14 k allocations: 600.650 KiB, 1.68% compilation time)

julia> @time Threads.@threads for _=1:4
    fib(40)
end
0.783171 seconds (34.33 k allocations: 2.571 MiB, 19.96% compilation time)
```


各種API(2)

@spawn

- スレッドの起動
(その都度本当に新しいスレッドが起動するわけではない、詳細割愛)
- 続く式を1つのタスクとしてまとめ、(空いている)スレッドで処理する
- @sync マクロで同期を取る必要あり

```
julia> Threads.nthreads()
6

julia> @time Threads.@threads for _=1:4
    fib(40)
end
0.783171 seconds (34.33 k allocations: 2.571 MiB, 19.96% compilation time)

julia> @time @sync for _=1:4
    Threads.@spawn fib(40)
end
0.715108 seconds (2.01 k allocations: 135.894 KiB, 8.50% compilation time)
```

各種API(3)

@threads v.s. @spawn (1)

- 大量の細かい(粒度がほぼ揃っている)タスクを並列処理したい場合は @threadsの方が優秀

```
julia> let
    hist = zeros{Int, Threads.nthreads()}
    @time Threads.@threads for _=1:(10000*Threads.nthreads())
        fib(15) # 毎回ほぼ同じ負荷の処理が実行される例
        hist[Threads.threadid()] += 1
    end
    hist
end
0.134166 seconds (22.88 k allocations: 1.588 MiB, 169.92% compilation time)
#> [10000, 10000, 10000, 10000, 10000, 10000]

julia> let
    hist = zeros{Int, Threads.nthreads()}
    @time @sync for _=1:(10000*Threads.nthreads())
        Threads.@spawn begin
            fib(15) # 毎回ほぼ同じ負荷の処理が実行される例
            hist[Threads.threadid()] += 1
        end
    end
    hist
end
0.301360 seconds (569.85 k allocations: 35.267 MiB, 102.04% compilation time)
#> [0, 12151, 12635, 11490, 11973, 11751]
```

各種API(4)

@threads v.s. @spawn (2)

- タスクの数がそこまで大量でなく、粒度がバラバラの場合は、@spawnの方が高パフォーマンスとなることもある(負荷分散)

```
julia> let
    hist = zeros{Int, Threads.nthreads()}
    @time Threads.@threads for _=1:(250*Threads.nthreads())
        fib(rand(25:35)) # 毎回わずかに異なる負荷の処理が実行
        hist[Threads.threadid()] += 1
    end
    hist
end
4.370164 seconds (34.68 k allocations: 2.347 MiB, 6.02% compilation time)
#> [250, 250, 250, 250, 250, 250]
```

```
julia> let
    hist = zeros{Int, Threads.nthreads()}
    @time @sync for _=1:(250*Threads.nthreads())
        Threads.@spawn begin
            fib(rand(25:35)) # 毎回わずかに異なる負荷の処理が実行
            hist[Threads.threadid()] += 1
        end
    end
    hist
end
4.310160 seconds (19.18 k allocations: 1.329 MiB, 1.33% compilation time)
#> [245, 244, 224, 283, 271, 233]
```

各種API(5)

その他

- Channel(spawn=true) do ~ end
 - 特定の処理だけをラップして別スレッドで処理できる
 - 適宜値の受け渡しができる
(元々Channelはタスク間で値を受け渡ししながら協調動作するためのもの)
- Threads.foreach()
 - Base.foreach() のマルチスレッド版
 - 処理(関数)とイテレータ(Channelでラップしたもの)を受け取りマルチスレッドで処理
 - 負荷分散に有用
- ※これらの詳細は割愛

実例

実例(1)

@threads の例(1)

- 配列の異なるインデックスに同様の演算を実施して値を格納するパターン(このパターンが多い)
- @threads は入れ子にできる(Julia v1.8 以降)

```
julia> function my_matmul(A::AbstractMatrix, B::AbstractMatrix)
    T = promote_type(eltype(A), eltype(B))
    C = Matrix{T}(undef, (size(A, 1), size(B, 2)))
    Threads.@threads for x = axes(B, 2)
        Threads.@threads for y = axes(A, 1)
            C[y, x] = @view(A[y, :])' * @view(B[:, x])
        end
    end
    C
end

my_matmul (generic function with 1 method)

julia> let A=[1 2; 3 4; 5 6; 7 8], B=[1 2 3; 4 5 6]
    C = my_matmul(A, B)
    @assert C == A * B
    @show C;
end

C = [9 12 15; 19 26 33; 29 40 51; 39 54 69]
```

実例(2)

@threads の例(2)

- 前スライドで示したサンプルのベンチマーク
- (※なお普通に $A * B$ を計算した方がメモリ使用量も少ないし断然速い(あくまで参考))

```
julia> function my_matmul_st(A::AbstractMatrix, B::AbstractMatrix)
    T = promote_type(eltype(A), eltype(B))
    C = Matrix{T}(undef, (size(A, 1), size(B, 2)))
    for x = axes(B, 2)
        for y = axes(A, 1)
            C[y, x] = @view(A[y, :])' * @view(B[:, x])
        end
    end
    C
end # シングルスレッド版
my_matmul_st (generic function with 1 method)

julia> using BenchmarkTools

julia> @btime my_matmul_st(A, B) setup=(A=rand(100, 100); B=rand(100, 100));
374.714 μs (2 allocations: 78.17 KiB)

julia> @btime my_matmul(A, B) setup=(A=rand(100, 100); B=rand(100, 100));
329.578 μs (3239 allocations: 419.11 KiB)

julia> # 参考
@btime (A * B) setup=(A=rand(100, 100); B=rand(100, 100));
48.012 μs (2 allocations: 78.17 KiB)
```

実例(3)

@spawn の例

- 標準の map 関数をマルチスレッド化してみた
- fetch() は、同期を取って結果の値を取得する関数

```
julia> function threaded_map(fn, array::AbstractArray)
    tasks = [Threads.@spawn(fn(v)) for v in array]
    [fetch(task) for task in tasks]
end
threaded_map (generic function with 1 method)

julia> threaded_map(fib, 15:40) == map(fib, 15:40)
true

julia> using BenchmarkTools

julia> @btime map(fib, 15:40);
1.419 s (1 allocation: 272 bytes)

julia> @btime threaded_map(fib, 15:40);
652.908 ms (167 allocations: 13.77 KiB)
```

注意(1)

スレッドセーフについて(1)

- なんでもマルチスレッド化して期待通りに動作するわけではない
- 例: 同じ変数(オブジェクト)の内容を更新する場合など
- ⇒スレッドアンセーフ

```
julia> mutable struct UnsafeCounter
    count::Int
    UnsafeCounter() = new(0)
end

julia> begin
    counter1 = UnsafeCounter()
    for n=1:1000
        counter1.count += 1
    end
    counter1.count
end
1000

julia> begin
    counter2 = UnsafeCounter()
    Threads.@threads for n=1:1000
        counter2.count += 1
    end
    counter2.count
end
650 # 想定より少ない値になってしまう (ことがある)
```

注意(2)

スレッドセーフについて(2)

- 更新するフィールドに @atomic を付ける
- そのフィールドを更新するときも @atomic を付ける
- ⇒スレッドセーフになる

```
julia> mutable struct AtomicCounter
    Threads.@atomic count::Int
    AtomicCounter() = new(0)
end

julia> begin
    counter3 = AtomicCounter()
    Threads.@threads for n=1:1000
        Threads.@atomic counter3.count += 1
    end
    counter3.count
end
1000 # 期待通りの結果になる

julia> begin
    counter4 = AtomicCounter()
    for n=1:1000 # シングルスレッドでも `Threads.@atomic` 必須
        Threads.@atomic counter4.count += 1
    end
    counter4.count
end
1000
```

注意(3)

その他

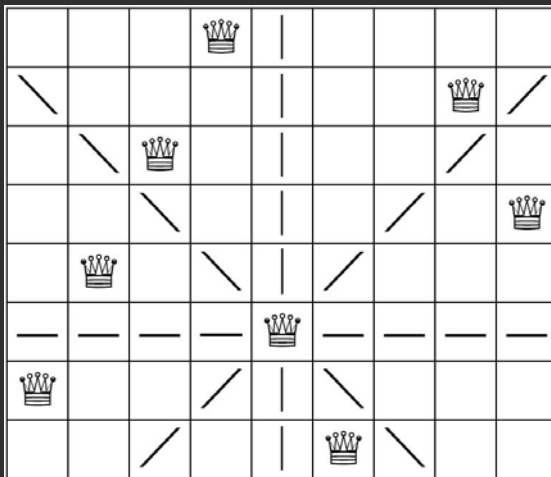
- もっと広い範囲をスレッドセーフにしたい
⇒ロック機構(@lock 《lockオブジェクト》 begin ~ end)による 排他制御を導入する必要あり
- デッドロック(競みの状態になってどのスレッドもブロック状態(Blocked)になってしまう)にも注意！
- ※これらの詳細は割愛

実習:並列化によるパフォーマンス改善

お題:N-Queen問題(1)

「N-Queen問題」とは？

- 8-Queen 問題とは、
8×8のマス目(=チェス盤)に8個のクイーンを、互いに攻撃し合わないよう
に配置するパターンが何
通りあるか、というパズル
問題
- これを $N \times N$ のマス目に
 N 個配置する、と一般化し
たのが N-Queen 問題



8-Queen 問題の解(一例)

お題:N-Queen問題(2)

N-Queen問題を解く(1)

- Julia のコード例を示す
- その1(前半):
メイン部および再帰部

```
function nQueen(n::Int)
    counter = AtomicCounter() # 先ほど作ったカウンタを利用
    for y=1:n
        Threads.@atomic counter.count += nQueen_sub(n, 1, [y])
    end
    counter.count
end

function nQueen_sub(n::Int, k::Int, params::Vector{Int})
    k == n && return 1
    board = create_board(n, params)
    counter = AtomicCounter()
    for y=1:n
        if issafe(board, y, k + 1)
            Threads.@atomic counter.count += nQueen_sub(n, k+1, [params; y])
        end
    end
    counter.count
end
```

お題:N-Queen問題(3)

N-Queen問題を解く(2)

- Julia のコード例を示す
- その2(後半):
盤面を表す BitMatrix
を生成する関数と、その盤
面に指定位置にコマを置
いて安全かを判定する関
数

```
function create_board(n, params)
    board = falses(n, n)
    for (x, y) in pairs(params)
        board[y, x] = true
    end
    board
end

function issafe(board::BitMatrix, y, x)
    h, w = size(board)
    any(board[y, :]) && return false
    any(board[:, x]) && return false
    any(board[y-i, x-i] for i=1:min(y, x)-1) && return false
    any(board[y+i, x+i] for i=1:min(h-y, w-x)) && return false
    any(board[y-i, x+i] for i=1:min(y-1, w-x)) && return false
    any(board[y+i, x-i] for i=1:min(h-y, x-1)) && return false
    true
end
```

お題:N-Queen問題(4)

N-Queen問題を解く(3)

- Julia のコード例を示す
- その3:
実行例(N=8~12まで)
- ※算出した解は所謂 バラエ
ティ解(回転・反転で同一視
できる物も個々に列挙)で
あり、基本解(回転・反転で
同一視できる物を1つと数
える方法)ではないことにも
注意

```
julia> using BenchmarkTools

julia> @btime nQueen(8)
 3.699 ms (75032 allocations: 2.95 MiB)
92

julia> @btime nQueen(9)
16.535 ms (326209 allocations: 13.11 MiB)
352

julia> @btime nQueen(10)
82.212 ms (1483298 allocations: 60.40 MiB)
724

julia> @btime nQueen(11)
436.700 ms (7371379 allocations: 303.26 MiB)
2680

julia> @btime nQueen(12)
 2.611 s (39709620 allocations: 1.61 GiB)
14200
```


お題:N-Queen問題(5)

N-Queen解法の並列化(1)

- 前半のコード
(`nQueen()` 関数、
`nQueen_sub()` 関数)
をマルチスレッド化(ス
レッド並列化)してみよ
う！
- ヒント:今日覚えたAPI
(`@threads`、`@spawn`
など)を使えばOK

```
# マルチスレッド化してみよう！
function nQueenMT(n::Int)
    counter = AtomicCounter() # 先ほど作ったカウンタを利用
    for y=1:n
        Threads.@atomic counter.count += nQueen_subMT(n, 1, [y])
    end
    counter.count
end

# マルチスレッド化してみよう！
# 適宜編集して最適な挙動にしてみよう！
function nQueen_subMT(n::Int, k::Int, params::Vector{Int})
    k == n && return 1
    board = create_board(n, params)
    counter = AtomicCounter()
    for y=1:n
        if issafe(board, y, k + 1)
            Threads.@atomic counter.count += nQueen_sub(n, k+1, [params; y])
            # Threads.@atomic counter.count += nQueen_subMT(n, k+1, [params; y])
        end
    end
    counter.count
end
```

お題:N-Queen問題(6)

N-Queen解法の並列化(2)

- 実行(N=8~12まで)し
てみて、パフォーマンス向
上しているか確認しよ
う！
(だいたい1/2~1/3くら
いの処理時間になるはず)
- ※うまく並列化できていな
いと逆にパフォーマンス低
下(2~10倍の処理時間)に
なることもあります、ご注
意！

```
julia> using BenchmarkTools

julia> @btime nQueenMT(8)
  ???? ms (????? allocations: ??? MiB)
92

julia> @btime nQueenMT(9)
  ???? ms (??????? allocations: ???.?? MiB)
352

julia> @btime nQueenMT(10)
  ???.??? ms (????????? allocations: ???.?? MiB)
724

julia> @btime nQueenMT(11)
  ????.??? ms (????????? allocations: ????.?? MiB)
2680

julia> @btime nQueenMT(12)
  ???? s (????????? allocations: ??? GiB)
14200
```

補足:アムダールの法則

- スレッド数 N で並列化しても、処理時間は $1/N$ とはならない
- 全体の 90% が並列化できるとした場合、パフォーマンスは最大でも $1/((1-0.9)+0.9/N)$ 倍 ($N \rightarrow \infty$ で 10倍)にしかない
- 「いかに並列化不可能な部分を少なくするか」が課題
- 参考: [Wikipedia\[ja\] アムダールの法則](#) 等

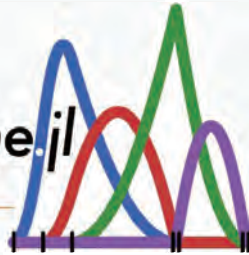
まとめ

講義の目的と目標

- ✓ スレッド並列(の基本)の理解
 - ✓ 並列処理とは何か
 - ✓ スレッドの基本的な概念
 - ✓ スレッドの利点と制限
 - ✓ (スレッドのライフサイクル)
- ✓ Juliaによるスレッド並列の理解
 - ✓ 最低限の事前準備
 - ✓ 各種API
 - ✓ スレッドセーフ(とデッドロック)

お疲れさまでした

BasicBSpline.jl で始める B-spline



堀川由人 (@Hyrodium)
2023 年 7 月 11 日

スライド公開先



<https://www.docswell.com/s/hyrodium/5Q89MJ-B-spline>

目次

1. 自己紹介
2. B-spline 最速入門
3. B-spline もう少し入門
4. B-spline の応用例
5. まとめ

2

目次

1. 自己紹介
2. B-spline 最速入門
3. B-spline もう少し入門
4. B-spline の応用例
5. まとめ

3

経歴・所属

名前

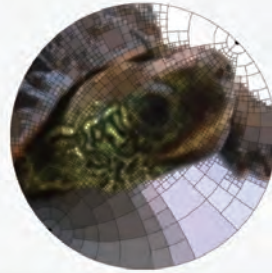
- 堀川由人
- @Hyrodium

経歴

- 大阪府立工業高等専門学校
- 大阪大学 (学部・修士)
- DMG 森精機

現所属

- Rist
- 大阪大学大学院基礎工学研究科
(招聘研究員)



4

Julia と私

Zenn.dev

- Julia の行列・ベクトルを完全に理解すっぞ!!
- FastGaussQuadrature.jl で数値積分しましょう
- Quaternions.jl をメンテナンスしてる話
- Julia 言語における中置演算子の扱い



GitHub









- JuliaGeometry/Rotations.jl
- JuliaApproximation/FastGaussQuadrature.jl
- JuliaArrays/StaticArrays.jl
- JuliaGeometry/Quaternions.jl
- JuliaMath/IntervalSets.jl
- hyrodium/ImageClipboard.jl
- hyrodium/**BasicBSpline.jl**



5

B-spline と私

これまでに執筆した記事など

-  B-spline 入門（線形代数がすこし分かる人向け）
B-spline のモチベーションを伝えるための動画
-  NURBS 多様体による形状表現
証明を詳しく書いた PDF 資料
-  BasicBSpline.jl のドキュメント
パッケージの使用方法和簡単な数学的解説
-  BasicBSpline.jl を作ったので宣伝です！
BasicBSpline.jl を作ったアナウンス on Zenn.dev
-  新しくなった BasicBSpline.jl(v0.8.3) の紹介
バージョンが上がった件について解説
-  なめらかな曲線を Julia で SVG 出力する
BasicBSpline.jl の応用例の紹介記事
-  BasicBSpline.jl - Basic mathematical operations for B-spline
Discourse でのパッケージ告知
-  Plotting smooth graphs with Julia
「なめらかな曲線を Julia で SVG 出力する」の英訳版

6

目次

1. 自己紹介
2. B-spline 最速入門
 - B-spline とは？
 - 関数近似
 - B-spline 基底関数
3. B-spline もう少し入門
4. B-spline の応用例
5. まとめ

7

B-spline とは？

区分多項式を使って関数近似するための道具

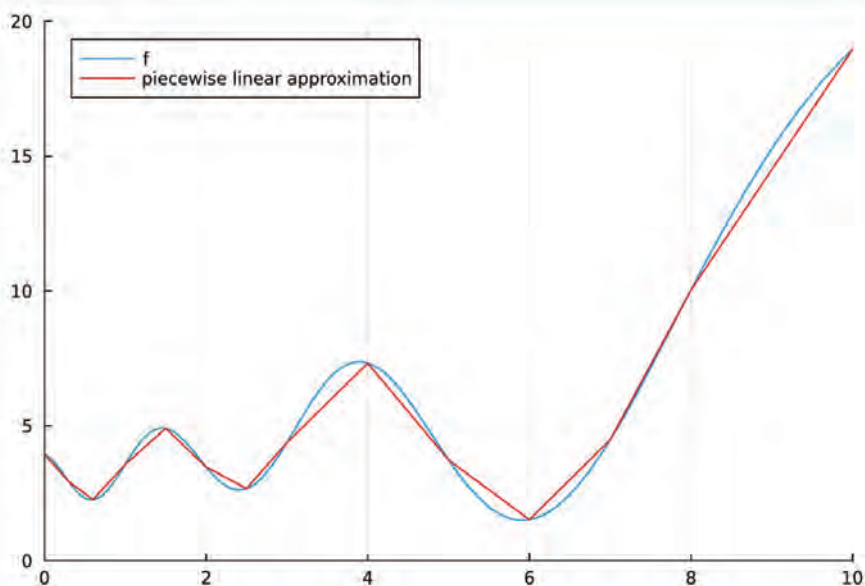
- 多項式
閉区間上の任意の連続関数は多項式で近似可能
(Weierstrass の多項式近似定理)
- 区分関数
定義域を適当な区間に分割し、各区間で関数を近似

B-spline で実現できること：

- 高すぎない多項式次数
- 多すぎない区間分割数
- 区分点で滑らかに繋げる

8

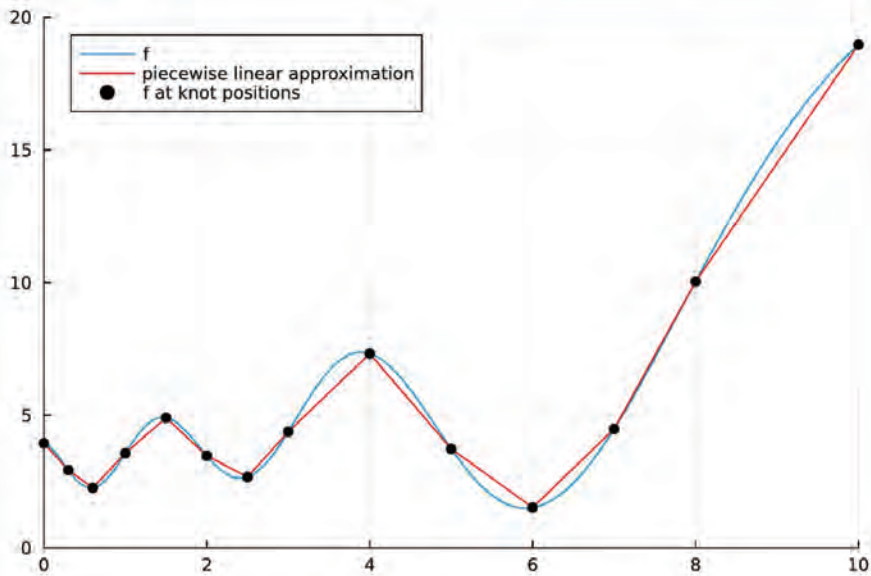
関数近似 ① (折れ線)



適当な点を折れ線で繋げば近似になる

9

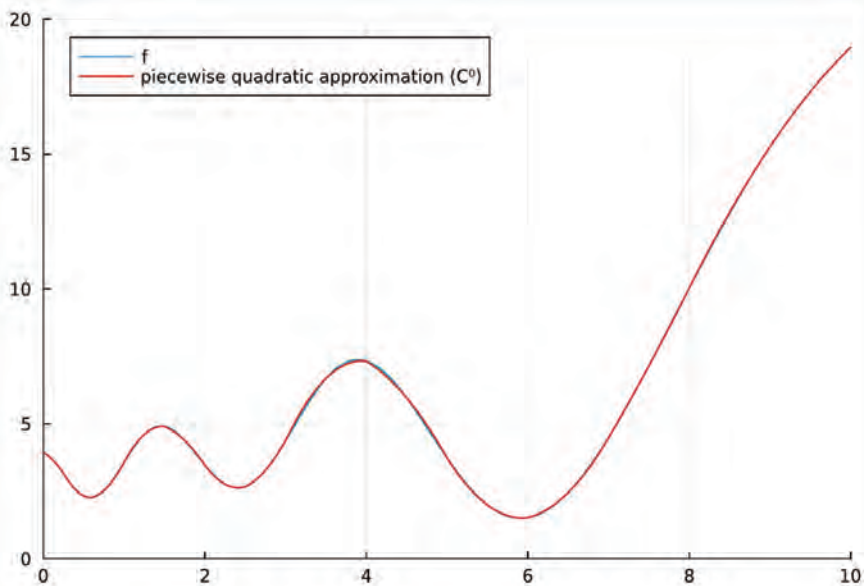
関数近似 ① (折れ線)



適当な点を折れ線で繋げば近似になる
多項式の繋ぎ目 (区分点) を「ノット (knot)」と呼ぶ

10

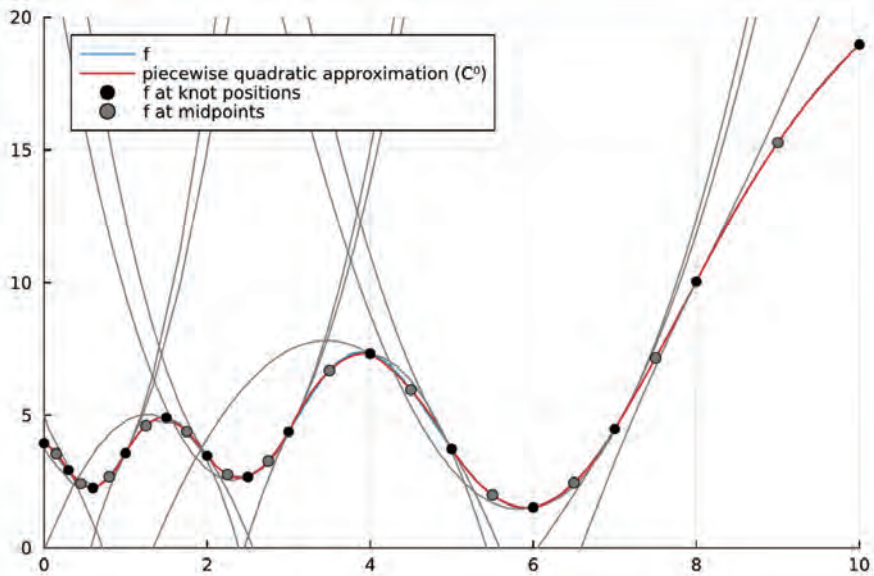
関数近似 ② (放物線)



各区間で3点取って放物線 (2次多項式) を描けば近似になる

11

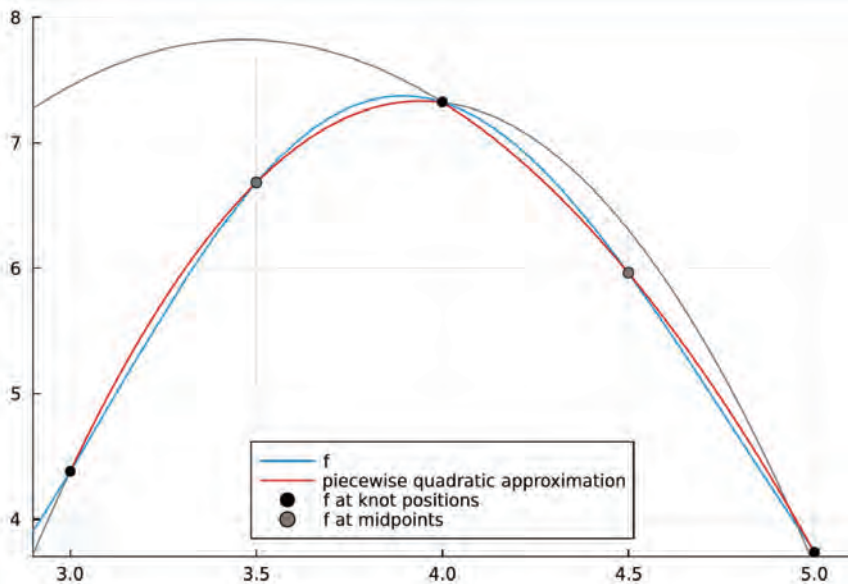
関数近似 ② (放物線)



各区間で3点取って放物線(2次多項式)を描けば近似になる

12

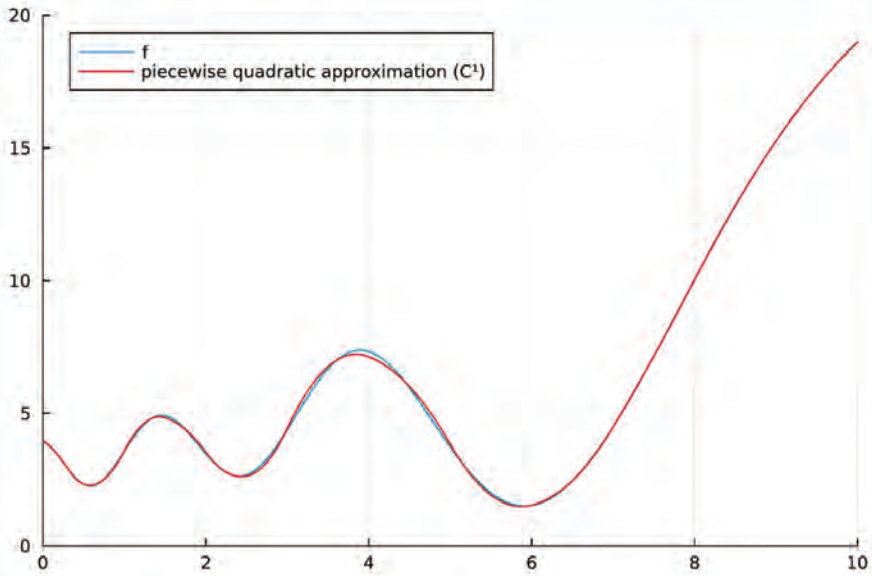
関数近似 ② (放物線)



各区間で3点取って放物線(2次多項式)を描けば近似になる
区分点で C^0 級になる

13

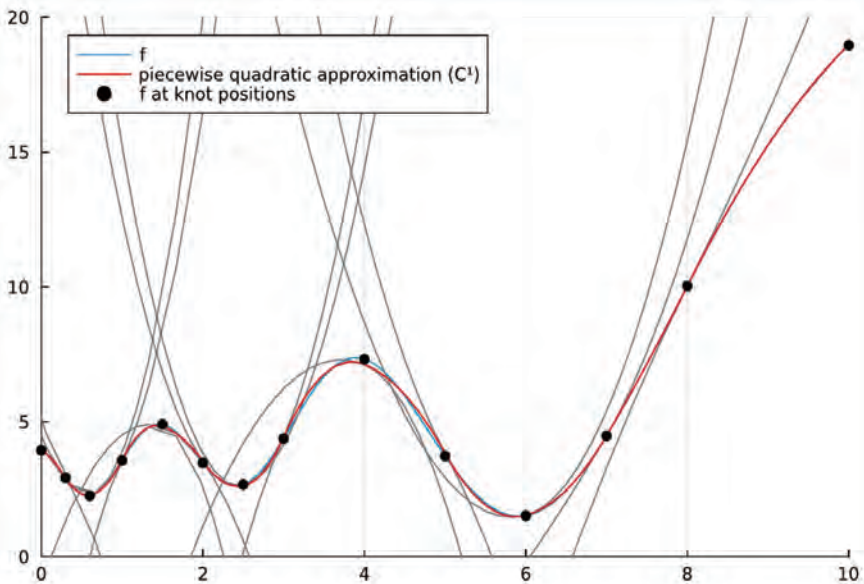
関数近似 ③ (放物線、ただし C^1 級)



放物線で近似し、区分点で C^1 級も要求

14

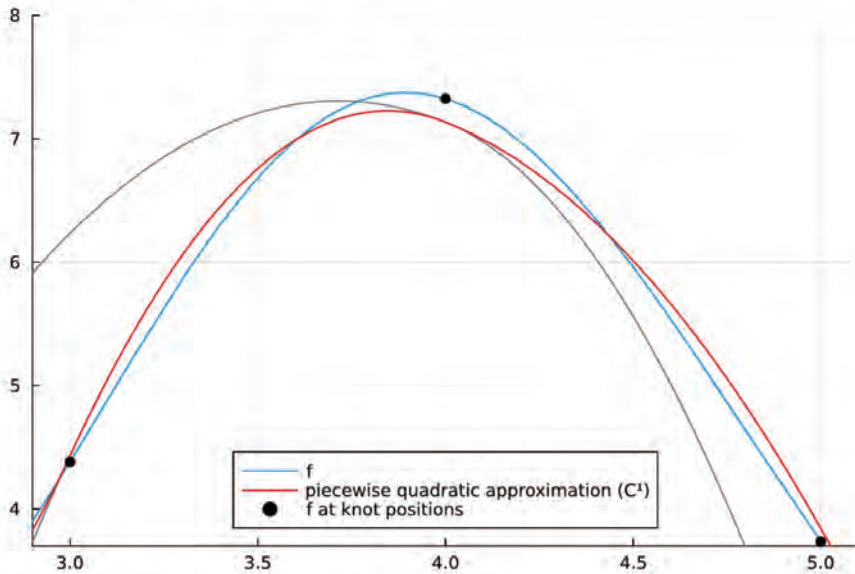
関数近似 ③ (放物線、ただし C^1 級)



放物線で近似し、区分点で C^1 級も要求

15

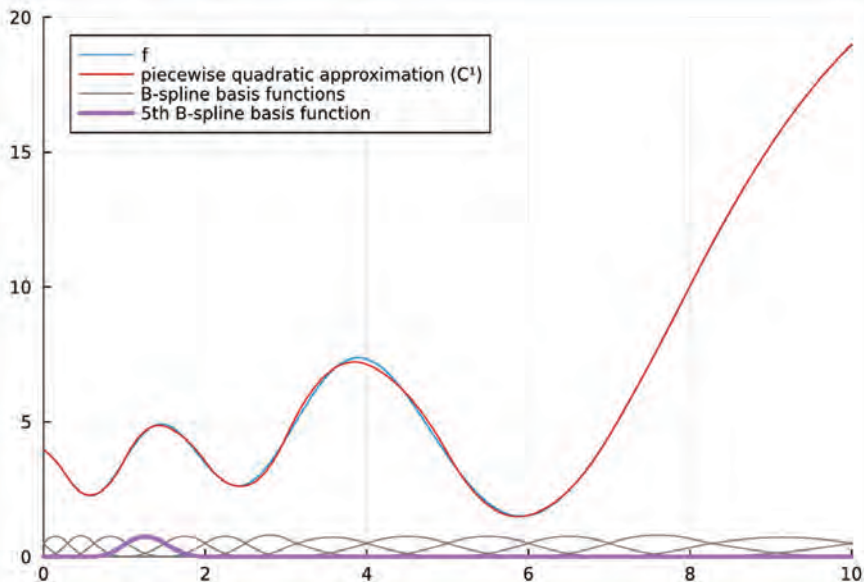
関数近似 ③ (放物線、ただし C^1 級)



放物線で近似し、区分点で C^1 級も要求
→ どうすれば実現できる…?

16

B-spline 基底関数 ① (近似の方針)



「滑らかさを保証した区分多項式」をたくさん用意して、線型結合すれば OK!
これが B-spline 基底関数!

17

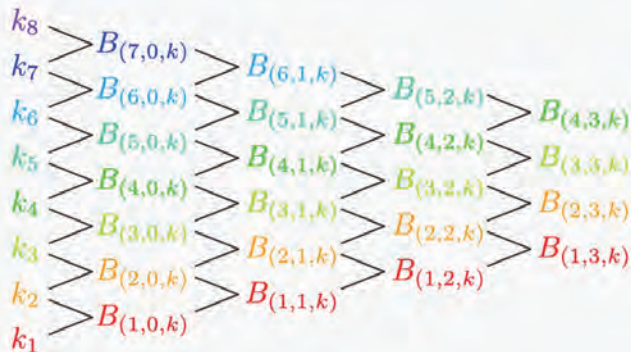
B-spline 基底関数 ② (定義)

定義 (B-spline 基底関数)

ノット列 $k = (k_1, \dots, k_l)$ に対する p 次 B-spline 基底関数 $B_{(i,p,k)}$ は以下で定義される

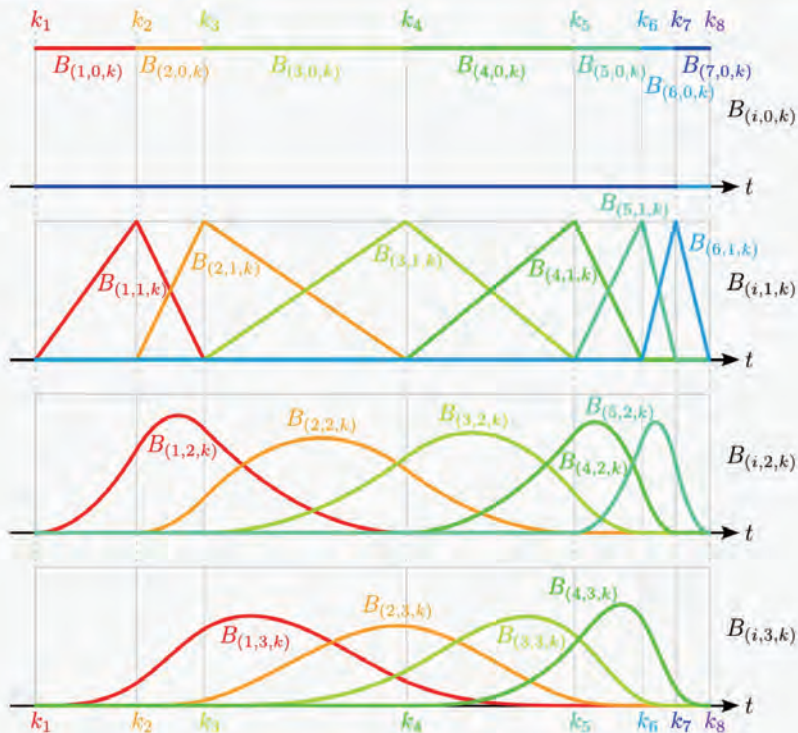
$$B_{(i,p,k)}(t) = \frac{t - k_i}{k_{i+p} - k_i} B_{(i,p-1,k)}(t) + \frac{k_{i+p+1} - t}{k_{i+p+1} - k_{i+1}} B_{(i+1,p-1,k)}(t)$$

$$B_{(i,0,k)}(t) = \begin{cases} 1 & (k_i \leq t < k_{i+1}) \\ 0 & (\text{otherwise}) \end{cases}$$



18

B-spline 基底関数 ② (定義)



19

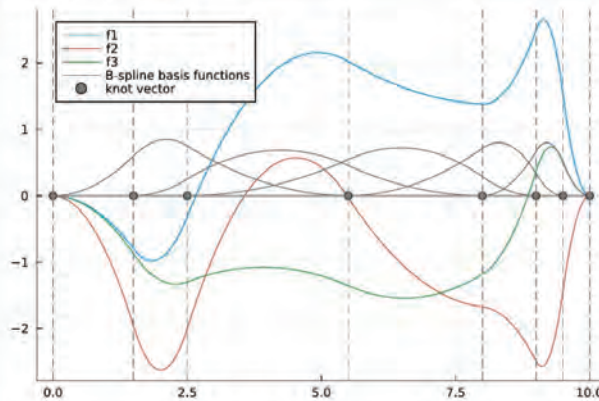
B-spline 基底関数 ③ (B-spline 基底関数の張る空間)

定理 (B-spline 基底関数の張る空間)

狭義単調増加なノット列 $k = (k_1, \dots, k_l)$ で構成される B-spline 基底関数について以下が成立

$$\text{span}_i \{B_{(i,p,k)}\} = \left\{ f \in C^{p-1}(\mathbb{R}) \mid \begin{array}{l} \forall i, f|_{(k_i, k_{i+1})} \text{ は } p \text{ 次多項式} \\ \text{supp}(f) \subseteq [k_1, k_l] \end{array} \right\}$$

この線形空間を $\mathcal{P}[p, k]$ で表す



20

B-spline 基底関数 ④ (疑問点いろいろ)

- 区間多項式空間 $\mathcal{P}[p, k]$ の次元は何で決まる？
- 2つの線形空間 $\mathcal{P}[p^1, k^1], \mathcal{P}[p^2, k^2]$ の包含関係は何で決まる？
- 特定のノット k_i での滑らかさの制約を変更できる？
- B-spline 基底関数 $B_{(i,p,k)}$ はどんな良い性質がある？
- B-spline 基底関数 $B_{(i,p,k)}$ の導関数は？
-

21

B-spline 基底関数 ④ (疑問点いろいろ)

- 区分数項式空間 $\mathcal{P}[p, k]$ の次元は何で決まる？
- 2つの線形空間 $\mathcal{P}[p^1, k^1], \mathcal{P}[p^2, k^2]$ の包含関係は何で決まる？
- 特定のノット k_i での滑らかさの制約を変更できる？
- B-spline 基底関数 $B_{(i,p,k)}$ はどんな良い性質がある？
- B-spline 基底関数 $B_{(i,p,k)}$ の導関数は？
- **BasicBSpline.jl** で何ができるのか？

22

B-spline 基底関数 ④ (疑問点いろいろ)

- 区分数項式空間 $\mathcal{P}[p, k]$ の次元は何で決まる？
- 2つの線形空間 $\mathcal{P}[p^1, k^1], \mathcal{P}[p^2, k^2]$ の包含関係は何で決まる？
- 特定のノット k_i での滑らかさの制約を変更できる？
- B-spline 基底関数 $B_{(i,p,k)}$ はどんな良い性質がある？
- B-spline 基底関数 $B_{(i,p,k)}$ の導関数は？
- **BasicBSpline.jl** で何ができるのか？

```
using BasicBSpline, Plots
P = BSplineSpace{2}(KnotVector([0,1,2,3,4,5,6]))
n = dim(P)
P ⊆ BSplineSpace{2}(KnotVector(0:8))
plot(P)
plot!(BSplineSpace{2}(KnotVector([0,1,2,3,3,4,5,6])))
plot!(t -> sum(bsplinebasis(P,i,t) for i in 1:n))
plot!(BSplineDerivativeSpace{1}(P))
```

↑ 完全理解しなくてOK。雰囲気だけ

23

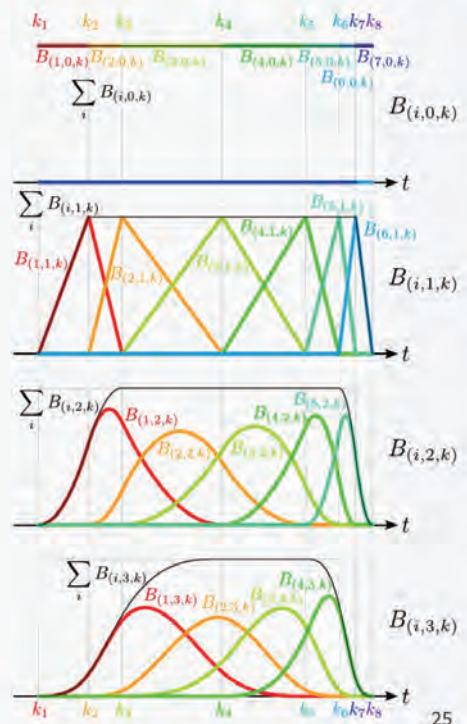
目次

1. 自己紹介
2. B-spline 最速入門
3. B-spline もう少し入門
 - B-spline 基底関数の性質
 - B-spline 多様体
 - リファインメント
4. B-spline の応用例
5. まとめ

B-spline 基底関数の性質 ①

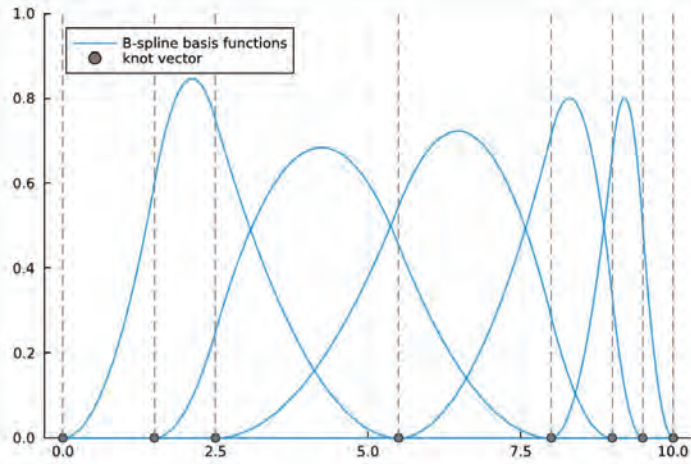
- 台
 - $\text{supp}(B_{(i,p,k)}) = [k_i, k_{i+p+1}]$
- 1 の分割
 - $\sum_i B_{(i,p,k)}(t) = 1 \quad (t \in [k_{1+p}, k_{l-p}])$

```
k = KnotVector(1:8)
P = BSplineSpace{2}(k)
n = dim(P)
plot(P)
scatter!(
    collect(k),
    zeros(length(k))
)
bsplinesupport(P, 2)
plot!(t->sum(
    bsplinebasis(P, i, t)
    for i in 1:n
))
```



B-spline 基底関数の性質 ②

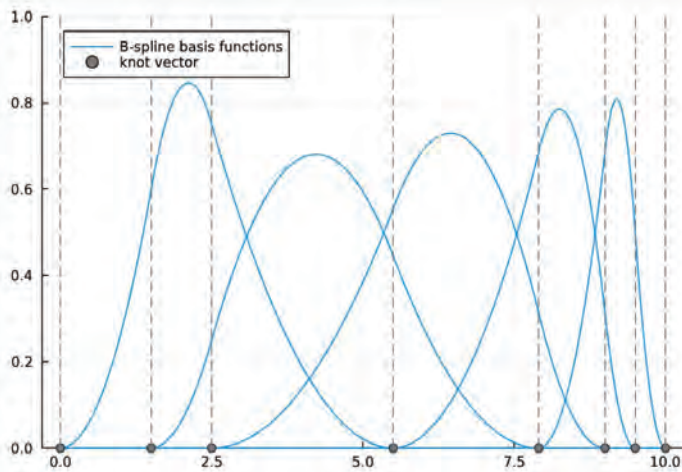
ノット列が重なるとどうなるか？



26

B-spline 基底関数の性質 ②

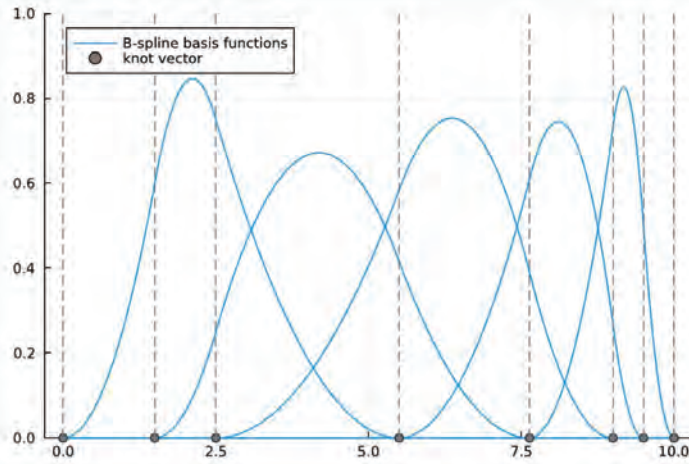
ノット列が重なるとどうなるか？



27

B-spline 基底関数の性質 ②

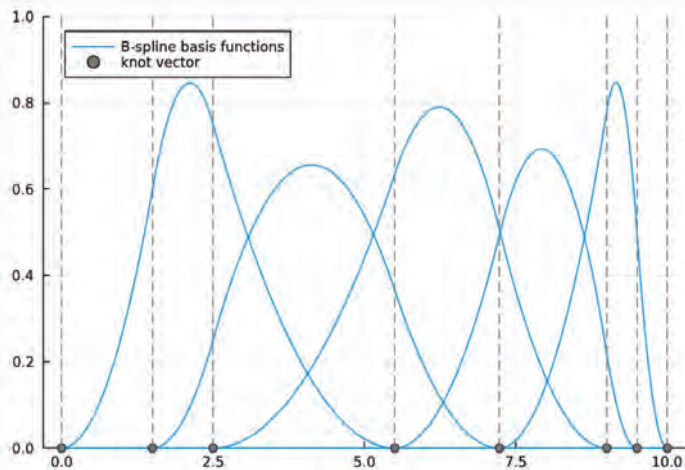
ノット列が重なるとどうなるか？



28

B-spline 基底関数の性質 ②

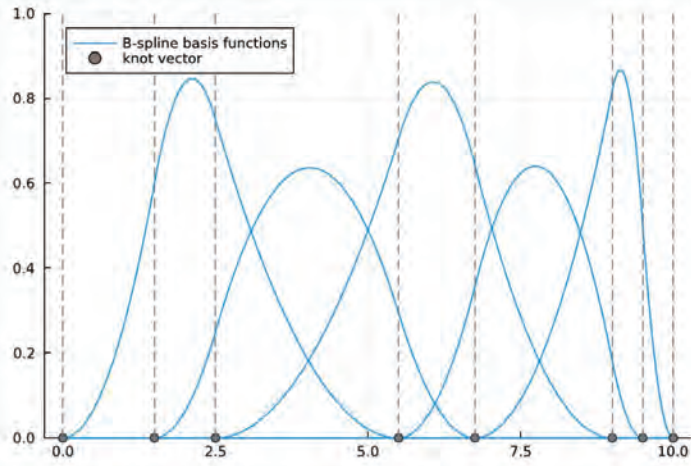
ノット列が重なるとどうなるか？



29

B-spline 基底関数の性質 ②

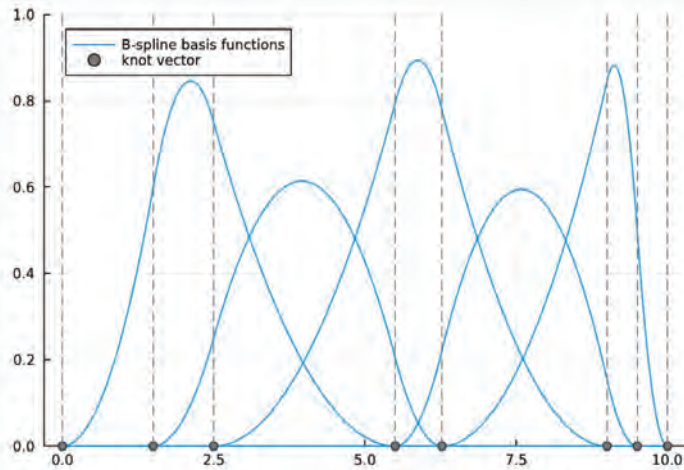
ノット列が重なるとどうなるか？



30

B-spline 基底関数の性質 ②

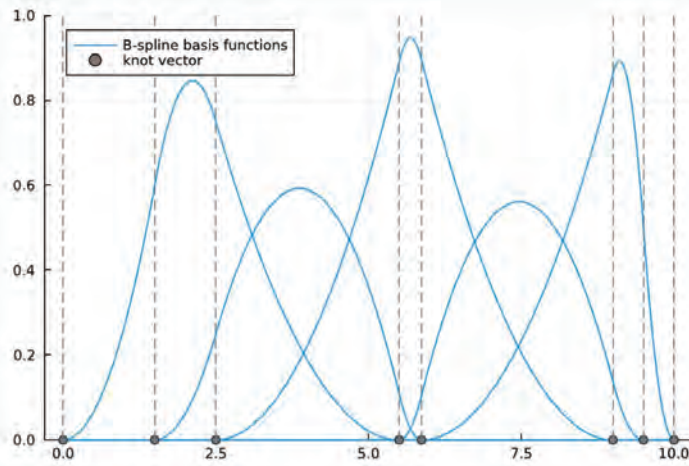
ノット列が重なるとどうなるか？



31

B-spline 基底関数の性質 ②

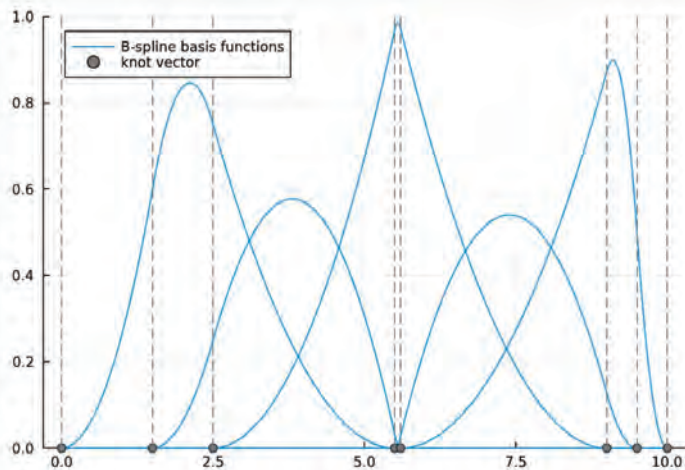
ノット列が重なるとどうなるか？



32

B-spline 基底関数の性質 ②

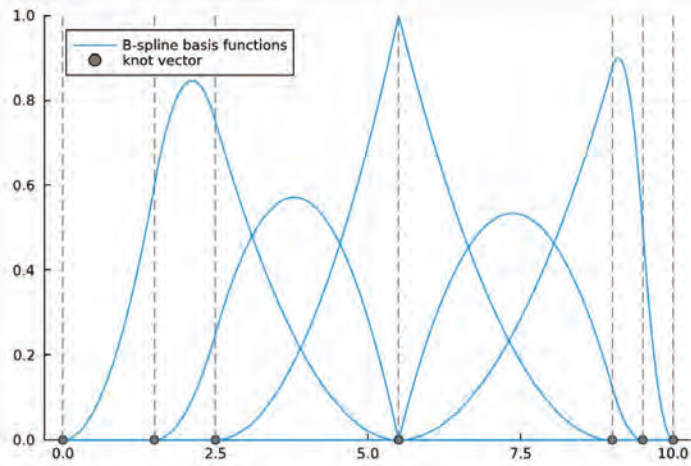
ノット列が重なるとどうなるか？



33

B-spline 基底関数の性質 ②

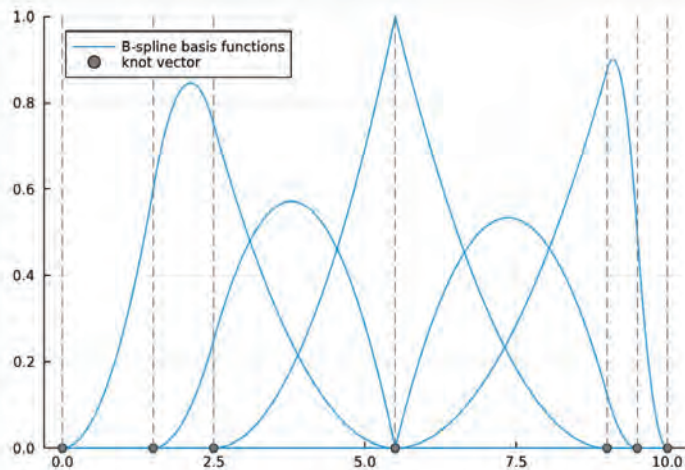
ノット列が重なるとどうなるか？



34

B-spline 基底関数の性質 ②

ノット列が重なるとどうなるか？



- 各点収束している
- 収束先で滑らかさが落ちている

35

B-spline 基底関数の定義 (ふたたび)

ノット列の重複があるとき：

- 各点収束している
- 収束先で滑らかさが落ちている

しかし、B-spline 基底関数の定義はノット列の重複を許さないはず

$$B_{(i,p,k)}(t) = \frac{t - k_i}{k_{i+p} - k_i} B_{(i,p-1,k)}(t) + \frac{k_{i+p+1} - t}{k_{i+p+1} - k_{i+1}} B_{(i+1,p-1,k)}(t)$$

$$B_{(i,0,k)}(t) = \begin{cases} 1 & (k_i \leq t < k_{i+1}) \\ 0 & (\text{otherwise}) \end{cases}$$

36

B-spline 基底関数の定義 (ふたたび)

ノット列の重複があるとき：

- 各点収束している
- 収束先で滑らかさが落ちている

しかし、B-spline 基底関数の定義はノット列の重複を許さないはず

$$B_{(i,p,k)}(t) = \frac{t - k_i}{k_{i+p} - k_i} B_{(i,p-1,k)}(t) + \frac{k_{i+p+1} - t}{k_{i+p+1} - k_{i+1}} B_{(i+1,p-1,k)}(t)$$

$$B_{(i,0,k)}(t) = \begin{cases} 1 & (k_i \leq t < k_{i+1}) \\ 0 & (\text{otherwise}) \end{cases}$$

ゼロ除算をゼロとすれば、収束先の関数が得られる！

区分多項式空間 $\mathcal{P}[p, k]$ の定義も拡張される

$$\mathcal{P}[p, k] = \text{span}_i \{ B_{(i,p,k)} \} = \left\{ f : \mathbb{R} \rightarrow \mathbb{R} \left| \begin{array}{l} \forall i, f|_{(k_i, k_{i+1})} \text{ は } p \text{ 次多項式} \\ \text{supp}(f) \subseteq [k_1, k_l] \\ f \text{ は } t \text{ の近傍で } C^{p-n_k(t)} \text{ 級} \end{array} \right. \right\}$$

$$n_k(t) = \# \{ i \mid t = k_i \}$$

37

一旦復習

これまでのあらすじ：

- (1) 与えられた関数 $f: I \rightarrow \mathbb{R}$ を区分局多項式で近似したい (I : 有界閉区間)
- (2) B-spline 基底関数 $B_{(i,p,k)}$ の線型結合で近似するのが良さそう

$$f(t) \approx \sum_i a_i B_{(i,p,k)}(t)$$

- (3) ノット列 k に重複があると $B_{(i,p,k)}$ の滑らかさが落ちる
- (4) $B_{(i,p,k)}$ の張る関数空間が $\mathcal{P}[p, k]$

次に気になること：

- ・ 関数だけでなく、曲線 $f: I \rightarrow V$ を近似できるか？
係数 $a_i \in \mathbb{R}$ を線形空間の元 $a_i \in V$ に置き換え

$$f(t) \approx \sum_i a_i B_{(i,p,k)}(t)$$

- ・ 1変数関数だけでなく、2変数関数 $f: I \times J \rightarrow \mathbb{R}$ を近似できるか？
B-spline 基底関数のテンソル積

$$f(u^1, u^2) \approx \sum_{i,j} a_{ij} B_{(i,p,k)}(u^1) B_{(j,p,k)}(u^2)$$

38

一旦復習

これまでのあらすじ：

- (1) 与えられた関数 $f: I \rightarrow \mathbb{R}$ を区分局多項式で近似したい (I : 有界閉区間)
- (2) B-spline 基底関数 $B_{(i,p,k)}$ の線型結合で近似するのが良さそう

$$f(t) \approx \sum_i a_i B_{(i,p,k)}(t)$$

- (3) ノット列 k に重複があると $B_{(i,p,k)}$ の滑らかさが落ちる
- (4) $B_{(i,p,k)}$ の張る関数空間が $\mathcal{P}[p, k]$

次に気になること：

- ・ 関数だけでなく、曲線 $f: I \rightarrow V$ を近似できるか？
係数 $a_i \in \mathbb{R}$ を線形空間の元 $a_i \in V$ に置き換え

$$f(t) \approx \sum_i a_i B_{(i,p,k)}(t)$$

- ・ 1変数関数だけでなく、2変数関数 $f: I \times J \rightarrow \mathbb{R}$ を近似できるか？
B-spline 基底関数のテンソル積

$$f(u^1, u^2) \approx \sum_{i,j} a_{ij} B_{(i,p,k)}(u^1) B_{(j,p,k)}(u^2)$$

39

一旦復習

これまでのあらすじ：

- (1) 与えられた関数 $f: I \rightarrow \mathbb{R}$ を区分局多項式で近似したい (I : 有界閉区間)
- (2) B-spline 基底関数 $B_{(i,p,k)}$ の線型結合で近似するのが良さそう

$$f(t) \approx \sum_i a_i B_{(i,p,k)}(t)$$

- (3) ノット列 k に重複があると $B_{(i,p,k)}$ の滑らかさが落ちる
- (4) $B_{(i,p,k)}$ の張る関数空間が $\mathcal{P}[p, k]$

次に気になること：

- 関数だけでなく、曲線 $f: I \rightarrow V$ を近似できるか？
係数 $a_i \in \mathbb{R}$ を線形空間の元 $a_i \in V$ に置き換え

$$f(t) \approx \sum_i a_i B_{(i,p,k)}(t)$$

- 1変数関数だけでなく、2変数関数 $f: I \times J \rightarrow \mathbb{R}$ を近似できるか？
B-spline 基底関数のテンソル積

$$f(u^1, u^2) \approx \sum_{i,j} a_{ij} B_{(i,p^1,k^1)}(u^1) B_{(j,p^2,k^2)}(u^2)$$

40

B-spline 曲線 ①

定義 (B-spline 曲線)

V を \mathbb{R} -線形空間、 a_i を V 上の点とする。このとき

$$p(t) = \sum_i a_i B_{(i,p,k)}(t) \quad (t \in [k_{1+p}, k_{l-p}])$$

で表される曲線を B-spline 曲線と呼ぶ。 a_i を制御点と呼ぶ。

```
using StaticArrays
# ノット列の定義
k = KnotVector([0,1,2,3,3,4,5,6,7])
# 区分局多項式空間の定義
P = BSplineSpace{2}(k)
n = dim(P)
# 制御点の定義
a = [SVector(cos(i),sin(i)) for i in 0:n-1]
# B-spline 曲線の定義
C = BSplineManifold(a, P)
plot(C)
```

41

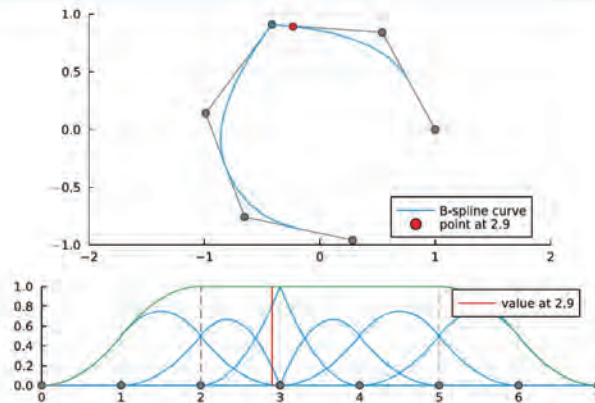
B-spline 曲線 ②

定義 (B-spline 曲線)

V を \mathbb{R} -線形空間、 \mathbf{a}_i を V 上の点とする。このとき

$$\mathbf{p}(t) = \sum_i \mathbf{a}_i B_{(i,p,k)}(t) \quad (t \in [k_{1+p}, k_{l-p}])$$

で表される曲線を B-spline 曲線と呼ぶ。 \mathbf{a}_i を制御点と呼ぶ。



端点を通るとは限らない!

42

B-spline 曲線 ③

制御点の平行移動で B-spline 曲線も平行移動する

```
a2 = [p+SVector(1,2) for p in a]
C2 = BSplineManifold(a2, P)
plot!(C2)
```

より一般に次の命題が成立

命題 (Affine 可換性)

$f: \mathbf{x} \mapsto A\mathbf{x} + \mathbf{b}$ を Affine 変換とし、制御点 $\{\mathbf{a}_i\}$ から構成される B-spline 曲線を $\mathbf{p}(\{\mathbf{a}_i\}; t)$ で表すとする。このとき次が成立。

$$\mathbf{p}(\{f(\mathbf{a}_i)\}; t) = f(\mathbf{p}(\{\mathbf{a}_i\}; t))$$

証明. 1 の分割の性質を使う

$$\begin{aligned} \mathbf{p}(\{f(\mathbf{a}_i)\}; t) &= \sum_i (A\mathbf{a}_i + \mathbf{b})B_{(i,p,k)}(t) = A \sum_i \mathbf{a}_i B_{(i,p,k)}(t) + \mathbf{b} \sum_i B_{(i,p,k)}(t) \\ &= A \sum_i \mathbf{a}_i B_{(i,p,k)}(t) + \mathbf{b} = f(\mathbf{p}(\{\mathbf{a}_i\}; t)) \quad \square \end{aligned}$$

43

B-spline 曲面

定義 (B-spline 曲面)

V を \mathbb{R} -線形空間、 \mathbf{a}_{ij} を V 上の点とする。このとき

$$p(u^1, u^2) = \sum_{i,j} \mathbf{a}_{ij} B_{(i,p^1,k^1)}(u^1) B_{(j,p^2,k^2)}(u^2)$$
$$((u^1, u^2) \in [k_{1+p^1}^1, k_{l^1-p^1}^1] \times [k_{1+p^2}^2, k_{l^2-p^2}^2])$$

で表される曲面を B-spline 曲面と呼ぶ。 \mathbf{a}_{ij} を制御点と呼ぶ。

```
# 区分多項式空間の定義
P = BSplineSpace{3}(KnotVector(0:7))
# 制御点の定義
a = [SVector(i-1.5, j-1.5, 1<=i<=2&&1<=j<=2)
      for i in 0:3, j in 0:3]
# B-spline 曲線の定義
M = BSplineManifold(a, P, P)
plotly()
plot(M)
gr()
```

44

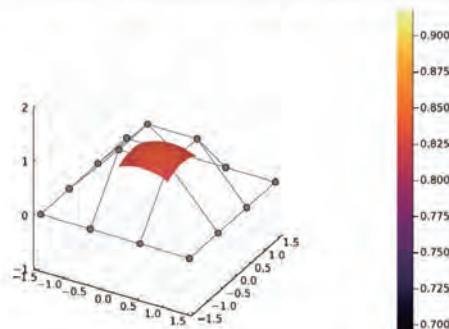
B-spline 曲面

定義 (B-spline 曲面)

V を \mathbb{R} -線形空間、 \mathbf{a}_{ij} を V 上の点とする。このとき

$$p(u^1, u^2) = \sum_{i,j} \mathbf{a}_{ij} B_{(i,p^1,k^1)}(u^1) B_{(j,p^2,k^2)}(u^2)$$
$$((u^1, u^2) \in [k_{1+p^1}^1, k_{l^1-p^1}^1] \times [k_{1+p^2}^2, k_{l^2-p^2}^2])$$

で表される曲面を B-spline 曲面と呼ぶ。 \mathbf{a}_{ij} を制御点と呼ぶ。

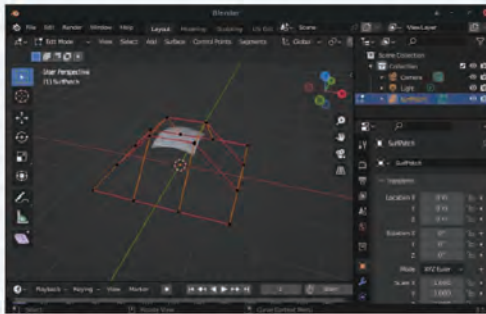


3次元以上も同様に定義可能

45

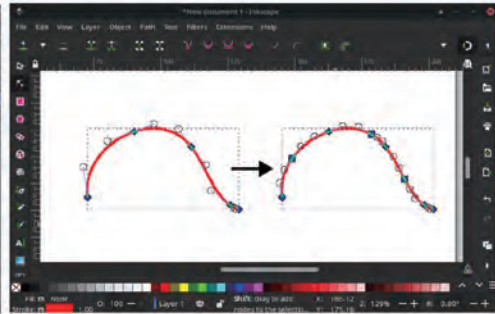
GUI アプリケーションでの例

Blender



B-spline 曲面を編集可能

Inkscape

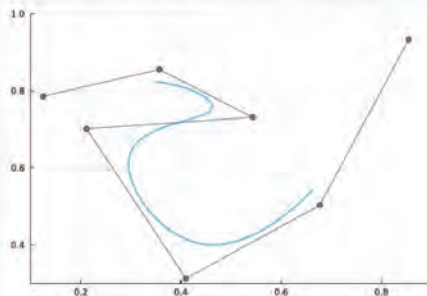


Bézier 曲線を編集可能

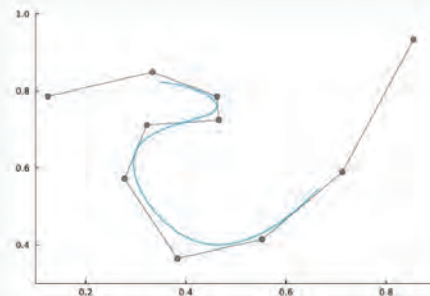
46

リファインメント

制御点の数を増やす操作のこと



元の B-spline 曲線 ($\mathcal{P}[p, k]$)



リファインメント後 ($\mathcal{P}[p', k']$)

- (1) 新しい次数 p' とノット列 k' を決める
- (2) 次数とノット列に合わせて制御点 a'_j を配置

$$p(t) = \sum_{i=1}^n a_i B_{(i,p,k)}(t) = \sum_{j=1}^{n'} a'_j B_{(j,p',k')}(t)$$

$$B_{(i,p,k)} = \sum_{j=1}^{n'} A_{ij} B_{(j,p',k')} \quad a'_j = \sum_{i=1}^n a_i A_{ij}$$

47

区分多項式空間の包含関係

$B_{(i,p,k)} = \sum_j A_{ij} B_{(j,p',k')}$ を満たす行列 A が存在するには $\mathcal{P}[p, k] \subseteq \mathcal{P}[p', k']$ でなければならない

定理 (区分多項式空間の包含関係)

$$\mathcal{P}[p, k] \subseteq \mathcal{P}[p, k'] \Leftrightarrow k \subseteq k'$$

$p < p'$ の場合はもう少し複雑な条件になる

```
k = KnotVector(0:10)
k' = k + KnotVector([3.4, 5.0, 6.3])
P = BSplineSpace{3}(k)
P' = BSplineSpace{3}(k')
k ⊆ k'
P ⊆ P'
plot(P)
plot!(P')
A = changebasis(P, P')
```

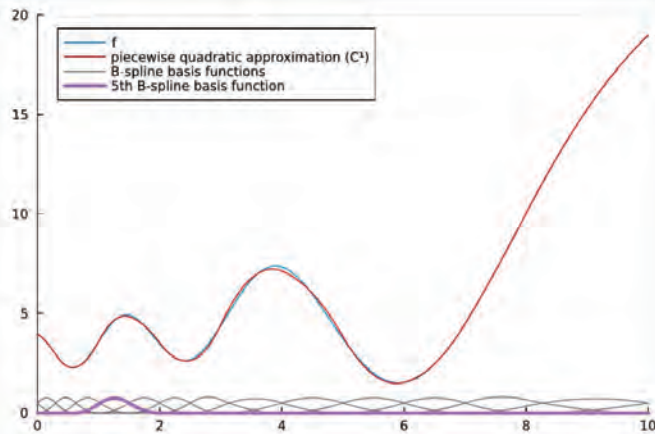
48

目次

1. 自己紹介
2. B-spline 最速入門
3. B-spline もう少し入門
4. B-spline の応用例
 - 関数近似
 - 滑らかなグラフを出力する
 - 内挿補間
 - 数値計算の例：編み紙
5. まとめ

49

関数を近似するには？(伏線回収)



最小2乗法で次式を最小化して係数 $\{a_i\}$ を決めれば良い

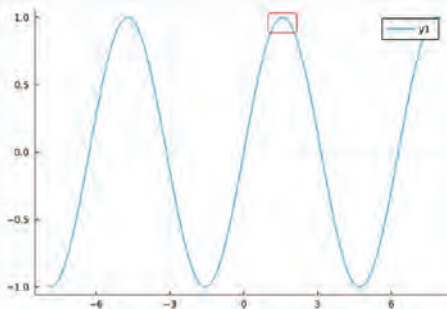
$$\int_I \left(f(t) - \sum_i a_i B_{(i,p,k)}(t) \right)^2 dt$$

```
using BasicBSplineFitting
f(x) = (1+x^2/10)*sin(exp(-(x-12)/4))+x/2+3
a = fittingcontrolpoints_I(f, P)
```

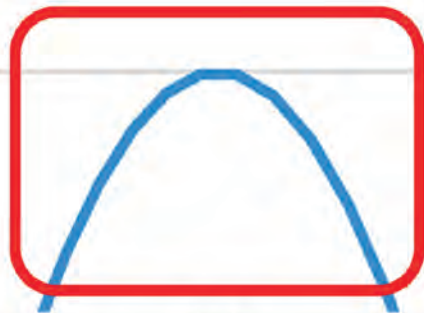
50

滑らかなグラフを出力するには？①

Plots.jl の出力グラフは折れ線



plot(sin,-8,8) の出力



拡大

滑らかなグラフが欲しい！

51

滑らかなグラフを出力するには？②

```
using BasicBSpline
using BasicBSplineFitting
using BasicBSplineExporter
using StaticArrays
f(t) = SVector(t,sin(t))
a, b = -8, 8
p = 3
k = KnotVector(a:b)+p*KnotVector([a,b])
P = BSplineSpace{p}(k)
a = fittingcontrolpoints_I(f,(P,))
C = BSplineManifold(a,(P,))
save_svg("sin.svg", C, xlims=(-10,10), ylims=(-2,2))
```

BasicBSplineExporter.jl パッケージで画像保存が可能！



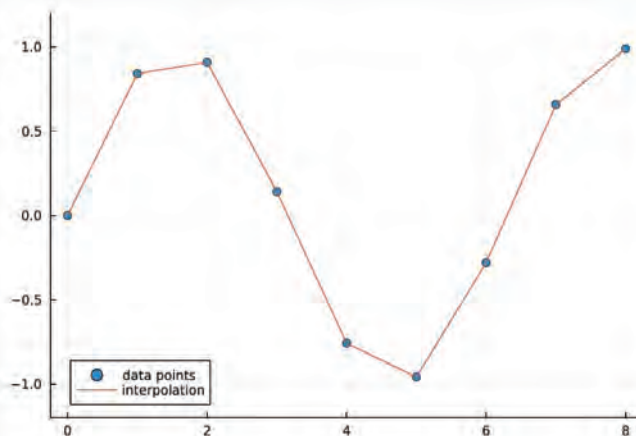
52

点列を内挿補間するには？①

以下の条件の補間を考える

- p 次区分多項式で内挿補間
- 区分点とデータ点が一致
- 区分点の近傍で C^{p-1} 級

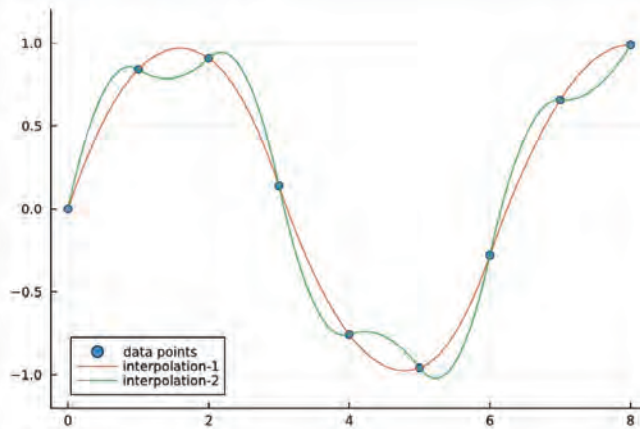
1 次区分多項式では自明



53

点列を内挿補間するには？②

2次区分多項式では一意的不是

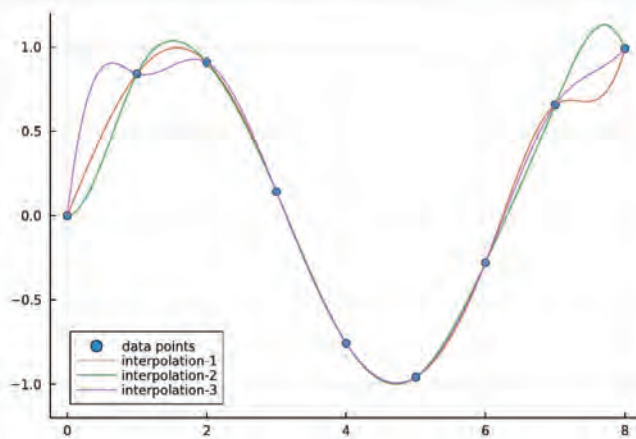


データ点を通る区分多項式は無数に存在 (1 自由度)

54

点列を内挿補間するには？③

3次区分多項式でも一意的不是

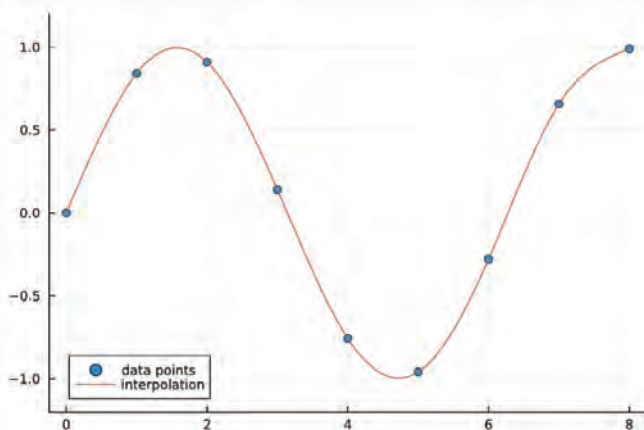


データ点を通る区分多項式は無数に存在 (2 自由度)

55

点列を内挿補間するには？④

3次区分多項式でも一意的不是



データ点を通る区分多項式は無数に存在 (2自由度)

両端での2階微分を0とすれば一意!

56

点列を内挿補間するには？⑤

BasicBSpline.jl では内挿補間の API は無いが、実装は容易

```
function interpolate(xs, fs::AbstractVector{T}) where T
    k = KnotVector(xs) + KnotVector([xs[1],xs[end]]) * 3
    P = BSplineSpace{3}(k)
    ddP = BSplineDerivativeSpace{2}(P)
    dda = [bsplinebasis(ddP,j,xs[1]) for j in 1:dim(P)]
    ddb = [bsplinebasis(ddP,j,xs[end]) for j in 1:dim(P)]
    M = [bsplinebasis(P,j,x) for x in xs, j in 1:dim(P)]
    M = vcat(dda', M, ddb')
    y = vcat(zero(T), fs, zero(T))
    return BSplineManifold(M\y, P)
end
ts = 0:8
F = interpolate(ts, sin.(ts))
scatter(ts, sin.(ts), label="data points")
plot!(t->F(t), 0, 8, label="interpolation")
```

57

編み紙 ①

紙を編んで滑らかな曲面を作る話



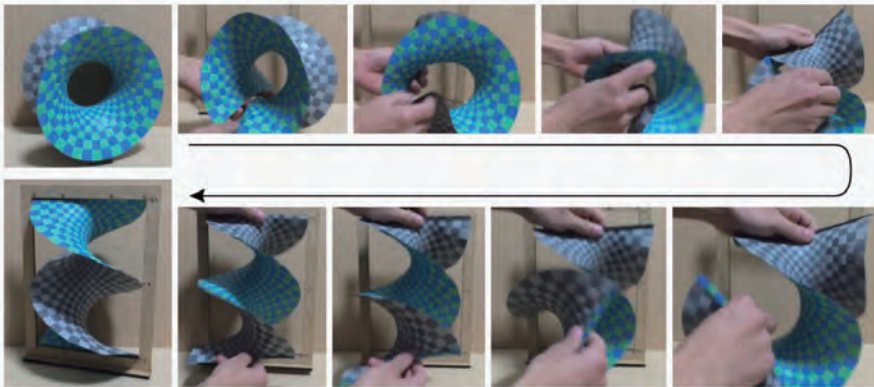
埋め込み形状 (上図中央) を B-spline 多様体として数値計算

- 曲面片の弾性変形エネルギーを最小化 (偏微分方程式)
- 制御点を未知変数として Galerkin 法が使える
- B-spline によって滑らかな (C^2 級の) 解が得られる
- 3 次 B-spline 曲線は SVG 画像として出力可能

58

編み紙 ②

本日、曲面模型を持ってきています！



詳細は JuliaCon2023 にて！

59

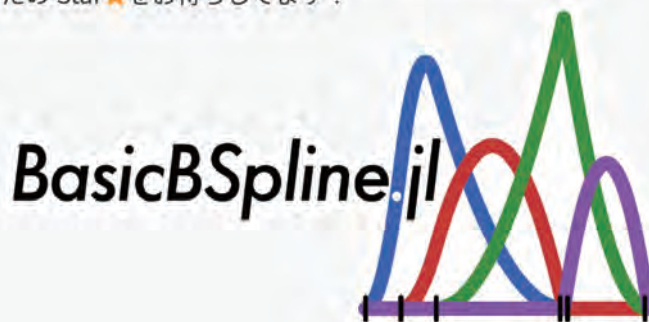
目次

1. 自己紹介
2. B-spline 最速入門
3. B-spline もう少し入門
4. B-spline の応用例
5. まとめ

60

まとめ

- B-spline は滑らかな区分多項式を扱うための道具
「区分関数」と「多項式」両方の性質を持つ ♣
- B-spline は数学的に奥深い
区分多項式空間 $\mathcal{P}[p, k]$ の性質、など
- B-spline は応用先が広い
幾何形状表現、内挿補間、Galerkin 法など
- B-spline を Julia で扱うには BasicBSpline.jl が便利！
あなたの Star★をお待ちしてます！



61

Julia 言語の良いところ

計算が速い、以外にも色々ある

- Unicode コーディング

```
k1 ⊆ k2, k1 + k2, P1 ⊆ P2
```

- 抽象型・具象型

```
KnotVector <: AbstractKnotVector,  
UniformKnotVector <: AbstractKnotVector
```

- パラメトリック型

```
BSplineSpace{p}
```

- メタプログラミング

```
P = BSplineSpace{3}(KnotVector(1:8))  
@less bsplinebasisall(P, 1, 4.2)
```

62

他パッケージとの比較

- [sostock/BSplines.jl](#) ★22
 - B-spline の基本的な操作を実装
 - 多項式次数 p が型パラメータに含まれないので遅い
- [jipolanco/BSplineKit.jl](#) ★40
 - 応用を重視した B-spline パッケージ
 - 悪い歴史的経緯を踏襲している (e.g. `BSplineOrder`)
- [HoBeZwe/NURBS.jl](#) ★2
 - 幾何形状表現への応用を意図したパッケージ
 - 実装途中の箇所が多い
- [kbarbary/Dierckx.jl](#) ★141
 - Fortran ライブラリの DIERCKX のラッパー
 - Pure Julia ではない (e.g. 有理数型が使えない)
- [hydrodium/BasicBSpline.jl](#) ★76
 - 数学志向で高速な B-spline の実装

(公平な比較になっていないことに注意)

63

- Elaine Cohen, Richard F. Riesenfeld, Gershon Elber (2001) “Geometric Modeling with Splines: An Introduction”
- Larry Schumaker (1989) “Spline Functions: Basic Theory”
- 市田浩三, 吉本富士市 (1979) 『スプライン関数とその応用』

応用数学の概念を用いた物理モデルをいかに Julia でプログラミングを行うか

降旗 大介 (大阪大学サイバーメディアセンター)
daisuke.furihata.cmc@osaka-u.ac.jp

数学と物理における Julia の活用@九州大学 IMI, 2023.07.11

1/19

Tips: 数値計算プログラムを組む上で Julia だと嬉しいこと

- 粗い近似解をある程度良い近似解へ近づける (繰り返し) 手法・アルゴリズムはたくさんあるし、原理もしっかりした、明快なものが多い。しかしその後、
ある程度良い近似解から、相当に良い近似解 \cong ほぼ正しい解を得る
 \longleftrightarrow 非線形方程式の根を求める
方法は結構実装が面倒。これに対し、Julia には “NLsolve” package があり、とても助かる…
- 科学技術計算プログラムでは連立一次方程式の解を求めるシーンが多いのに、通常のコンピュータ言語にはそういったライブラリは標準装備ではない。ライブラリを探して、関数をマニュアルで調べて、使い方を確認して… 面倒だ！
 \implies Julia だと “LinearAlgebra” package が標準装備なので、とても楽。
例: 行列 A とベクトル x, b に対して $Ax = b$ を解いて x を求めたい場合, $x = A \setminus b$ と書くだけで良い！
- 既に多くの科学技術系のライブラリが存在する！ そのうえ, dump コマンドをうまく使うと、ライブラリの計算結果を自由に抽出できたりする。
 \implies ライブラリのソースを読まなくても、内部結果などを知ることができる。

2/19

さて… 研究目的・背景

- Cahn–Hilliard 方程式系系統の問題 (相分離現象などのモデル方程式) の数値解析をしたい
 - ▶ movie(method of line による数値解)
 - ▶ movie(DVDM による数値解)
- これらの問題には質量保存, エネルギー減少性などの大域的な性質がある. そもそも Cahn–Hilliard にかぎらずこうした問題は多い.
- これらを保存する数値解析法 構造保存数値解法 structure-preserving method の研究はそれなりに内外で発展している (例: SciCADE という国際研究集会ではこの内容のセッションがいつもある).
- structure-preserving method は数値解が優れていることが多い.
- ODE だとハミルトン系 $+\alpha$ で考えることが多いが, PDE だと変分構造を介して考えることが多い (PDE のそれは離散変分導関数法などと称していて日本発).
- これらの手法は基本的に微積を始めとした「連続極限で定義される operator」を離散的に定義してその一貫性を担保する方法. ある意味数学的に厳密. **ただし計算量は大きめで, そこが難点だが, 近年では高速化手法も充実してきた.**
- 上の枠組みはそれなりに充実してきているので, そろそろ根本的に違う切り口も考えたい

3/19

Cahn–Hilliard 方程式 概要 (1)

相分離現象のモデル方程式の一つで, 点 (x, t) において状態が A 相, B 相のいずれに近いかを示す物理量 $u(x, t)$ に対し以下のような PDE で記述されるもの (多くの場合, A 相状態を $u = 0$, B 相状態を $u = 1$ などとする).

$$\frac{\partial u}{\partial t} = \Delta \left[-\epsilon \Delta u + \frac{d}{du} P(u) \right] \quad \text{for } t \in [0, \infty), x \in \Omega \quad (1)$$

ただし, Δ は x に関するラプラシアン, $0 < \epsilon \ll 1$, $P(u)$ は A 相状態, B 相状態を底にもつ二重井戸型関数, 境界条件は説明略.

このとき, (適切な境界条件のもとで) 次の 2 つの大域的性質がある.

質量保存性

$$\frac{d}{dt} \int u \, dx = 0 \quad (2)$$

エネルギー減少性

$$\frac{d}{dt} \int G(u, \nabla u) \, dx \leq 0 \quad (3)$$

ただし, G はエネルギー関数と呼ばれる下記の量.

$$G(u, \nabla u) = \frac{\epsilon}{2} |\nabla u|^2 + P(u) \quad (4)$$

4/19

Cahn–Hilliard 方程式 概要 (2)

とくにエネルギー減少性は重要で、解の存在性を証明する数学的なキーでもある。その方程式との関連性は明瞭で以下の通り。

エネルギー関数 G に対して、その変分導関数が

$$\frac{\delta G(u, \nabla u)}{\delta u} = -\epsilon \Delta u + \frac{d}{du} P(u) \quad (5)$$

と導出できるので、これを用いて Cahn–Hilliard 方程式は下記の形に書き換えられる (**この形式がポイント**)。

$$\frac{\partial u}{\partial t} = \Delta \frac{\delta G}{\delta u} \quad (6)$$

よってエネルギー減少性は

$$\begin{aligned} \frac{d}{dt} \int G(u, \nabla u) dx &= \int \frac{\delta G}{\delta u} \frac{\partial u}{\partial t} dx + \text{境界項}^* \\ &= \int \frac{\delta G}{\delta u} \Delta \frac{\delta G}{\delta u} dx = - \int \left\| \nabla \frac{\delta G}{\delta u} \right\|^2 dx + \text{境界項}^* \leq 0 \quad (7) \end{aligned}$$

* : 想定されている境界条件によって 0 となる。

注: 離散変分導関数法はこのページを丸々離散化する

5/19

Cahn–Hilliard 方程式の通常の数値解法

PDE に対する普通の解法、特殊な解法等が使える。

- method of line(空間方向を先に離散化して連立 ODE で近似し、ODE 数値解法で解く)。

[▶ この解法の解説 pdf](#)

時間方向の離散化幅 Δt を小さくすれば一応使えるが、質量保存性、エネルギー減少性はときおり破れ、そのせいで物理的にみてもおかしい数値解になりがち (とくに長時間発展させた場合)。

- 構造保存数値解法としての**離散変分導関数法**(差分法、有限要素法等いろいろあり。線形スキームもある)。

長時間発展も考えると現在の本命はこちらか。基本的にきちんと動かし、数値解の様子が物理的におかしいということもあまり無い。数値解の一意存在性や安定性が証明できるケースもある。ただし、多くの場合数値スキームが時間方向に陰的(未知の数値解に対して連立非線形方程式を解いて数値解を求めるという形のこと)で、**計算量は大きめ**。

線形スキームを設計する方法論(多段化)もあるが、問題の非線形性が多項式の形状でないと使えないこと、4次を越えると数値的な不安定性が強くなることなどの制限がある。

「妥当な数値解を高速に得る」という方法論については、充実しているとはまだ言い難い。

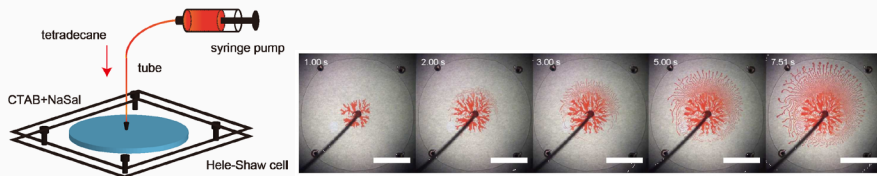
6/19

Cahn–Hilliard 方程式の order reduction: 現状

妥当な数値解を得るための方法の一つに、order reduction (問題をより低次元の問題に置き換えて解く) がある。

- Alikakos らが $\epsilon \rightarrow 0$ のときこの粗視化過程の相の境界面の挙動が Hele–Shaw 問題 (領域境界面の挙動問題) の解に収束することを証明[†]。
- order reduction としては上の結果は数学的にはほぼ満足 (問題の次元が文字通り 1 次元下がる)。
- ただし、Hele–Shaw 問題の数値解析は厄介。むしろ「Hele–Shaw 問題の数値解法として Cahn–Hilliard 方程式を解く方法」が提案される状態。

[†] N. D. Alikakos, P. W. Bates and X. Chen, Convergence of the Cahn–Hilliard equation to the Hele–Shaw model, Arch. Rat. Mech. Anal. 128 (1994), 165–205.



Hele–Shawflow の実験模式図と実験画像 (K.Yoshii and Y.Sumino, arXiv 1904.10673, 2019. より引用)

7/19

Cahn–Hilliard 方程式の粗い order reduction を作るには (method 1)

背景・アイデアのスタート地点:

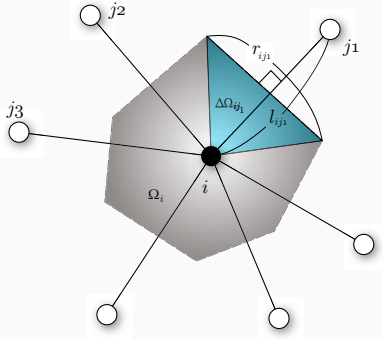
- 微分幾何をベースに、粒子法的な particle dynamics model 方程式を作る。つまり、各地点での $u(x, t)$ が時間変化するという問題ではなく、(たとえば) $u \cong 1$ な粒子が移動する、という問題に書き換える。
- このとき解決すべき問題は大きくわけて以下の2つ。
 - 1 「解 u の値が大きい \iff 粒子がそこで密集している」という関係性をどうやって定量化するか。
 - 2 関数 $u(x, t)$ に対するラプラシアン Δ などの微分作用素を、粒子の位置関係によって発生する力などによって解釈し直すか。
- 実は上の2つの問題とも、「粒子の位置を母点、生成点とした空間分割」をもとにした**離散微分**を用いることで近似として解決可能 \implies Voronoi 分割 (次スライド) などが代表的。

8/19

ポロノイ分割とは: 一筋の良い空間分割

- 任意の点はもっとも近いポロノイ領域に含まれるとする。つまり、母点 x_i によるポロノイ領域 Ω_i を以下のように定義する。

$$\Omega_i \stackrel{\text{def}}{=} \{x \mid \|x - x_i\| \leq \|x - x_j\| \text{ for any } j \neq i\}$$



- ・母点, 生成元 = 空間をこの点をもとにポロノイ分割する
- ・ポロノイ点 = ポロノイ領域の頂点
- ・ Ω_i と Ω_j が隣接しているとき

$$\begin{cases} r_{ij} & \stackrel{\text{def}}{=} |\Omega_i \cap \Omega_j|, \\ l_{ij} & \stackrel{\text{def}}{=} \|x_i - x_j\|. \end{cases}$$

▶ 母点から Voronoi 分割が進む様子の movie

▶ Voronoi 分割等の離散化に伴う微積分の離散化

9/19

方程式を粒子法相当に真面目に離散化した方程式

モデル方程式の全容は以下ようになる。

$$\begin{aligned} \frac{dx_i}{dt} &= -\left(\frac{1}{4m}\right) \sum_{(M_{ad})_{ij} \text{ is true}} \frac{\delta G}{\delta u} \Big|_i n_{ij} r_{ij}, \\ \frac{\delta G}{\delta u} \Big|_i &= p(2u_i - 1) + r(u_i - 1)^3 + \frac{2q}{|\Omega_i|} \sum_{(M_{ad})_{ij} \text{ is true}} \left(\frac{u_j - u_i}{l_{ij}}\right) r_{ij}, \\ u_i &= \frac{m}{|\Omega_i|}. \end{aligned}$$

ただし, $x_i(t)$ は i -番 particle の位置, m は particle の大きさ (面積), p, q, r は Cahn–Hilliard 方程式の定数. $|\Omega_i|$ は母点 x_i により生成されたポロノイ領域 Ω_i の大きさで, 行列 M_{ad} はポロノイ領域の隣接関係を列挙した隣接関係行列. n_{ij}, l_{ij}, r_{ij} は隣接している場合のみ非ゼロで, それぞれ particle i と particle j 間境界の (i からみた) 外向き単位法線ベクトル, particle 間の距離, 同境界の長さである。

このモデル方程式は数学的には根拠がある合理的なものだが, 粒子の位置が近づきすぎたり, 領域の外へ行ってしまったり (密度 $\rightarrow 0$ を実現しようとして) すると Voronoi 分割が不安定になるため, 実際の計算にはそうとうの繊細さが必要となる。

▶ このモデルの Julia プログラム

▶ このモデルによる数値計算例 (動画) 10/19

数値計算結果より見えてくること (method 1)

- 👍 数学的にはきちんと近似しているので、(計算が可能なら) 計算結果はそれなりに信頼できそう.
- 👍 密度の高いところに粒子がたくさん集まり空間分割の解像度が高くなるので、メッシュ調整の一種になっている.
- 👍 離散近似の際に空間次元に依存する箇所がないので、3次元、4次元でも問題ない.
- ⚠️ 点の距離が近くなると Voronoi 分割の数値計算は不安定になりがち.
- ⚠️ 空間の「外へ」出ていこうとする粒子が存在しうるので、この対処が必要.

というわけで、もう少し「簡潔なモデル」を考えたい.

11/19

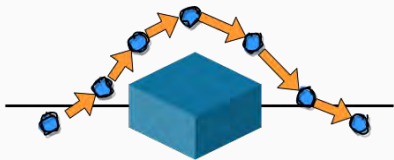
Cahn–Hilliard 方程式の粗い order reduction を作るには (method 2)

背景・アイデアのスタート地点:

- Cahn–Hilliard 方程式を数学的な手法で order reduction するのはいったん止めて…
- もともとの相分離現象における「粗視化過程 (coarsening process)」のみを再現する素朴なモデルを考える
- その際、少なくとも質量保存性を再現する (そうしないと物理的にかなり奇妙に見える)

粗視化過程を観察すると:

- 全体に「流体」現象のような挙動
- 少領域同士に主に距離に依存した引力的な相互作用がありそう
- 流体的挙動と考えると、近くの領域を「直接越えて」力が働くモデルには違和感



流体では、物質同士の相互作用により力が障害物を迂回して伝わる ⇒ 迂回しての隣接関係が成り立つようにしたい…

というわけで、

12/19

particledynamics model of Cahn–Hilliard 方程式 (1/2)

- $u \cong 1$ で半径が一定の球 (particle) の位置移動のみを記述するモデルを考える.
- particle は増減しないし, 大きさも変わらないし, 重ならない. これによって質量保存性を確保.
- particle 同士は, 互いの距離に応じて引き合う (とりあえず逆二乗則)
- ただし, particle が重ならないように非常に近い場合は体積排除効果的な力が働く.
- ただし, 近くの particle を何重にも越えての力はあまり働かない (緩和隣接関係)

これは例えば下記のようにしてモデル化する.

- 空間上の particle をそれぞれ母点として空間を **Voronoi 分割**し, Voronoi 領域の隣接関係から作られる隣接行列を M_{ad} とする.
 - モデルパラメータ p (正整数) に対し, 緩和隣接行列を $M_p \stackrel{\text{def}}{=} (M_{ad})^p$ とし, M_p によって隣接していると判断される particle 間でのみ力が発生するとする.
- 化学物質の易動度の概念のように, particle に働く力 \propto particle の速度
 - 流体抵抗のように, particle の速度には上限あり

…という, いわば **re-modeling** を行う

13/19

particledynamics model of Cahn–Hilliard 方程式 (2/2)

モデル方程式の全容は, 具体的には以下の通り.

$$\begin{aligned} \frac{dx_i(t)}{dt} &= C_1 \left(\frac{\tanh(C_2 \|f_i(t)\|)}{C_2 \|f_i(t)\|} \right) f_i(t), \\ f_i(t) &= \sum_{(M_p)_{ij} \text{ is true}} f_{ij}(t), \\ f_{ij}(t) &= \begin{cases} C_3 \frac{x_j - x_i}{\|x_j - x_i\|^3} & : \|x_j - x_i\| \geq r_{\text{cri}}, \\ -2C_3 \frac{x_j - x_i}{\|x_j - x_i\|^3} & : \|x_j - x_i\| < r_{\text{cri}}. \end{cases} \end{aligned}$$

ただし, $x_i(t)$ は i -番 particle の位置, C_1, C_2, C_3 は正定数.

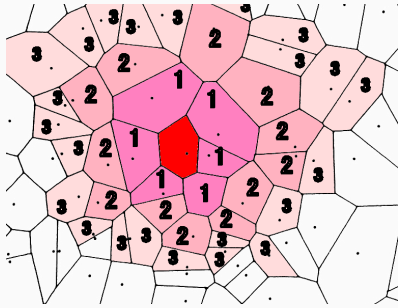
行列 M_p は $M_p \stackrel{\text{def}}{=} (M_{ad})^p$ (p は正整数, 3 程度), その計算過程に必要な真偽値計算 (ブール値の計算) は $a \cdot b = a$ and $b, a + b = a$ or b となる.

行列 M_{ad} はボロノイ領域の隣接関係を列挙した隣接関係行列で, 母点 x_i により生成されたボロノイ領域 V_i と母点 x_j により生成されたボロノイ領域 V_j とが隣接しているときは $(M_{ad})_{ij}$ が真, そうでなければ偽である.

14/19

ポロノイ領域の隣接関係

図の中心の赤いポロノイ領域 V_i をに対する隣接関係行列 $(M_{ad})^p$ (p は正整数) の意味は…



- 1 が書かれた領域 … $(M_{ad})^1$ の (i, j) 成分が「真」となるポロノイ領域 V_j .
- 2 が書かれた領域 … $(M_{ad})^2$ の (i, j) 成分が「真」となるポロノイ領域 V_j .
- 3 が書かれた領域 … $(M_{ad})^3$ の (i, j) 成分が「真」となるポロノイ領域 V_j .

… 注目！ この隣接関係は「迂回路」を持ちうる
→ 流体の「回り込み」挙動をモデリングできる.

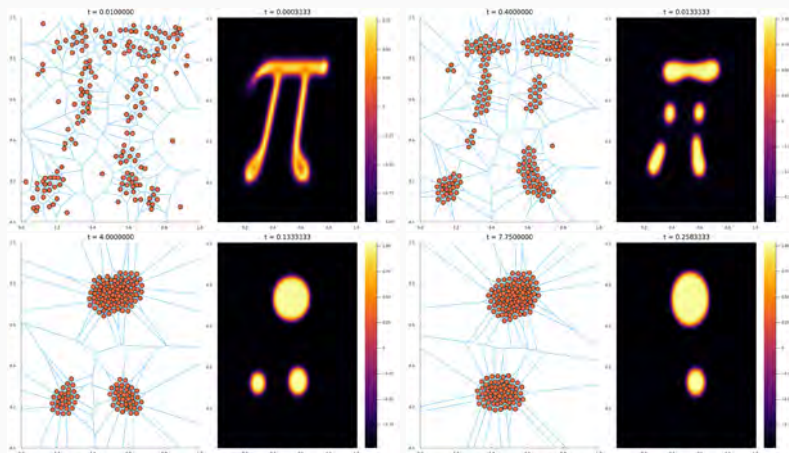
空間のポロノイ分割の例.

▶ この解法の julia code

▶ >> movie (初期配置が random)

15/19

実験結果: 粗視化過程をそれなりに re-modeling できたのか?



左: particle モデルによる数値解, 右: Cahn-Hilliard 方程式による数値解

▶ >> movie (初期配置が pi 形状)

16/19

数値計算結果より見えてくること (method 2) 1/2

- 👍 **非常に遅い挙動が発現している**. この挙動は領域の移動・変形過程で見られる特徴の一つである.
- 👍 **領域の移動速度がそのサイズに依存する** (大きいほど遅い). こうした効果をモデルに直接取り入れているわけではないので興味深い.
- 👍 **質量保存則が成り立つ**. これはモデルの設計からして当然.
- ⚠️ **決定方法が不明な未知パラメータが含まれる**モデルである. 具体的には, C_1, C_2, C_3, p などが未知パラメータ. これらをどう決めると Cahn–Hilliard 方程式の解挙動を近似できるのか, は future work.
 - 領域の生成・消滅メカニズムが陽的にはこのモデルには組み込まれていない. ただし, 擬似的にこうしたメカニズムが再現はされている様子.
なお, この欠如は今回のモデルとしては対象現象から切り捨てたから当然であるが, 組み込めるならばそれに越したことはない. (→ future work)

17/19

数値計算結果より見えてくること (method 2) 2/2

- 👍 Cahn–Hilliard 方程式の解の挙動をシミュレートするためのシンプルな粗視化モデルとして **particle motion model** を設計した.
- 👍 数値計算例をみるとこのモデルで領域の移動・変形に**遅い挙動が発現**しており, Cahn–Hilliard 方程式の粗視化モデルとして期待できる.
- ⚠️ モデルに含まれる未知パラメータの決定方法は現状不明.
 - Cahn–Hilliard 方程式の大域エネルギーに相当するこのモデルでの特徴量を定義・計算し, その挙動を調べる必要がある. なぜなら Cahn–Hilliard 方程式の大域エネルギーは時間発展に伴い減少する散逸量であることが判明しているため.
 - 3次元領域でのこのモデルでの数値計算の実験等を行いたい.
なお, 下記のように, 空間次元を上げても, 計算コストは指数的に増大したりしない.
 - 2次元問題では空間のポロノイ分割の計算コストは多くのアルゴリズムで $O(n \log n)$.
ただし $n = \# \text{ points}$.
 - Delaunay 三角分割を行ってポロノイ分割を計算する Bowyer–Watson アルゴリズムは, 問題の次元によらず (おお!), 普通は計算コストが $O(n \log n)$.
ただし最悪のケース (点配置が退化している場合) では $O(n^2)$ になりうる.

18/19

まとめ

- 👍 複雑な問題に対しても, Julia + jupyter 環境だと, 少しずつ試行錯誤することで意外に気楽にプログラムの完成にこぎつけられる.
- 👍 想像するより多くのライブラリが有り, かなり「頼る」ことができる.
- 👍 他の言語にはもう戻れない!

Juliaと量子多体計算

スパースモデリングからQuantics Tensor Trainまで

品岡寛

埼玉大学

- 学術変革領域研究B 「量子古典融合アルゴリズムが拓く計算物質科学」
- JST さきがけ 「量子情報処理」
- 基盤研究B 「次元圧縮技術が拓く二粒子応答の第一原理計算」

自己紹介 物性の定量的予言がしたい → 計算物理 + 第一原理計算

Fortran, Ruby, C/C++, Python, Julia と使ってきて、今はJuliaがメイン

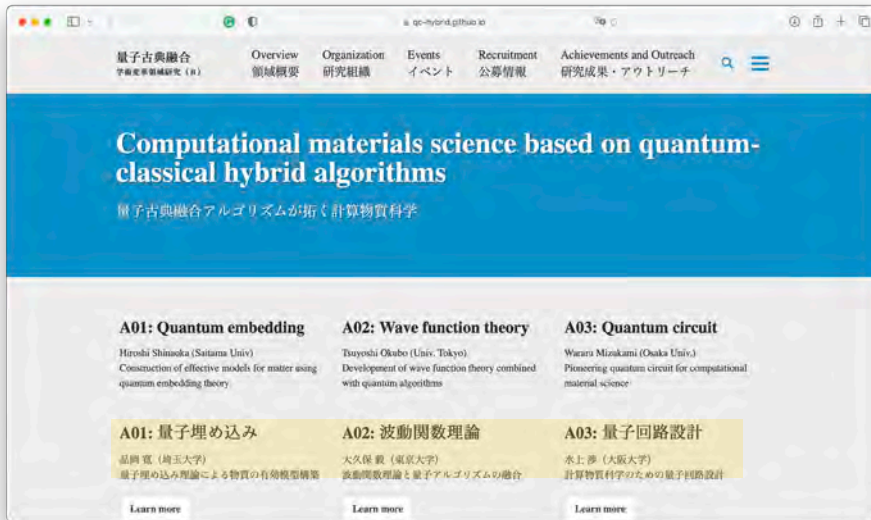
- Python的な気楽な書き方から、Fortran的な速度優先の書き方まで可能
- 学生が勉強しやすい

- ALPS/CT-HYB C++による量子モンテカルロ法実装 (C++)
<https://github.com/ALPSCore/CT-HYB>
- SparseIR.jl/sparse-ir 虚時間グリーン関数のスパースモデリング (Python/Julia)
<https://github.com/SpM-lab/SparseIR.jl>
<https://github.com/SpM-lab/sparse-ir>
- DCore 動的平均場近似計算 (Python)
<https://github.com/issp-center-dev/DCore>
- SpM 数値解析接続 (C++)
<https://github.com/SpM-lab/SpM>

量子古典融合アルゴリズムが拓く計算物質科学 (2023年4月～)

<https://qc-hybrid.github.io/>

領域代表: 品岡寛



概要

計算物理は、桁違いに異なる長さスケールの共存を扱う

例 量子物性物理 (低温物理現象 vs 10^1 eV ~ 10^5 K のバンド幅)
宇宙ひも (プランクスケールの細さ vs 宇宙スケールの長さ)
乱流

幅広い長さスケールを扱える表現の開発 → 試行錯誤 & 並列計算

Quantics tensor train



連続変数関数を離散化した「テンソルネットワーク」表現

MSSTA.jl (仮名)

2009年に応用数学で提案され、ここ数年で流体力学、量子力学等への適用が始まる

グリーン関数のスパースモデリング



悪条件の積分方程式の擬似四倍精度演算
虚時間の最適基底を構成

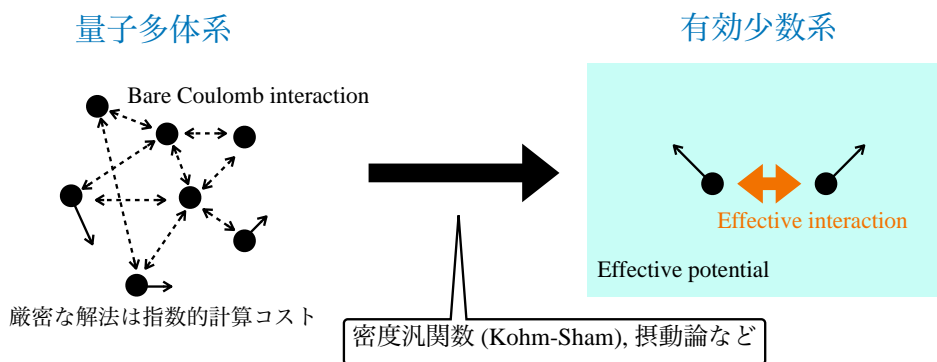
SparseIR.jl

品岡等によって2017年に提案され、虚時間形式の場の量子論・第一原理計算に応用が進む

発表の構成

- 背景 時空データの圧縮
- Quantics tensor train 連続時空の離散化とテンソルネットワーク
- スパースモデリング 疑似多倍精度による積分方程式の解法
- 今後のJulia応用について

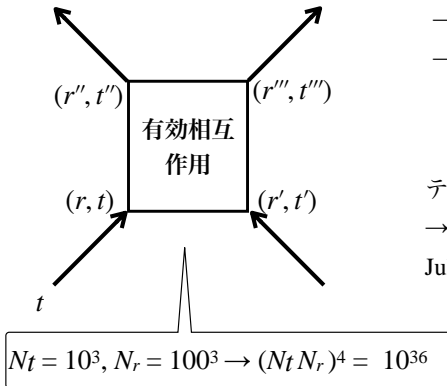
情報圧縮技術がなぜ必要？ (1/2ページ)



- グランドチャレンジ
- より精度の良い近似を捜す (高次の摂動論、機械学習？など)
 - 少数系ですら計算が難しい → 次ページ

情報圧縮技術がなぜ必要？ (2/2ページ)

2つの粒子が散乱される過程



桁違いに異なるスケールの混在

→単純メッシュではデータ量が発散

→圧縮&演算する技術が必要

流体力学計算等でも同じような問題

テンソルネットワーク、スパースモデリング...

→試行錯誤が迅速で、並列計算までカバーする

Juliaが最適

Quantics tensor train

- [HS et al.](#), Physical Review X **13**, 021015 (2023)
- M. K. Ritter, Y. N. Fernández, M. Wallerberger, J. von Delft, [HS](#), X. Waintal, arXiv:2303.11819

<https://github.com/shinaoka/MSSTA.jl>

Quantics tensor train (QTT)とは？

I. V. Oseledets, Doklady Math. **80**, 653 (2009)

B. N. Khoromskij, Constr. Approx. **34**, 257 (2011)

桁違いに異なる長さスケールが混在する関数を圧縮可能

Image compression

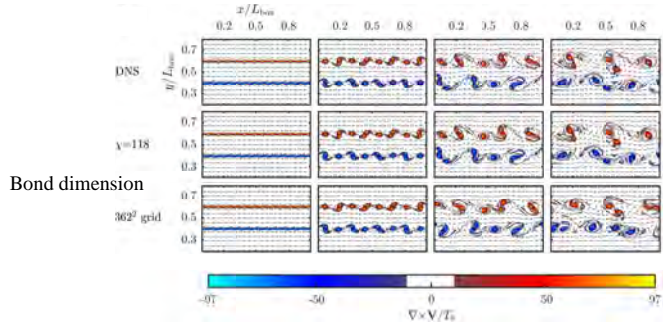


José I. Latorre, arXiv:quant-ph/0510031v1

Vlasov-Poisson equations for collisionless plasmas

E. Ye and N. F. G. Loureiro, arXiv:2205.11990

2022年8月にアイデア思いつく、2022年10月に投稿 (Juliaだから可能だった)



N. Gourianov et al., Nat. Comput. Sci. **2**, 30 (2022)

場の量子論の時空依存性を圧縮・演算できる！ HS et al., PRX **13**, 021015 (2023)

テンソルネットワーク (1/3)

スカラー

a



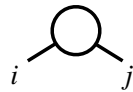
ベクトル

a_i



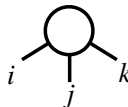
行列

a_{ij}



3階テンソル

a_{ijk}



突き出している足がindexに対応。突き出している向き、図形の形・色に意味はない。

テンソルネットワーク

$$i \text{---} \boxed{A} \text{---} \boxed{x} \text{---} j = \sum_j A_{ij} x_j$$

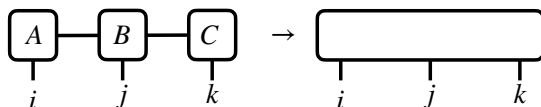
共通している足に対する和を取る。

$$\boxed{A} \text{---} \alpha \text{---} \boxed{B} \text{---} \beta \text{---} \boxed{C} = \sum_{\alpha, \beta} A_i^\alpha B_j^{\alpha, \beta} C_k^\beta$$

上・下付きにも意味はない。

テンソルネットワーク (2/3)

Contraction

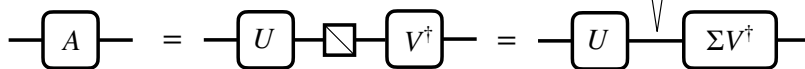


実際に virtual bond に対する和を計算することを contraction をとるといふ。行列積として実行可能

テンソルネットワーク (3/3)

特異値分解

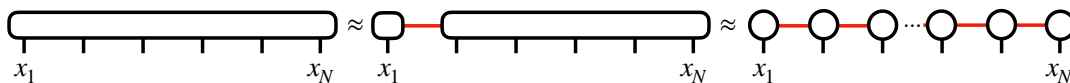
$$A = U \Sigma V^\dagger$$



$$(\Sigma)_{ij} = \delta_{ij} s_i$$

特異値は非負。急速に減衰する場合は、行列の低ランク近似が可能

行列積表現 (matrix product state) / tensor train



分解は特異値分解などで実行可能。最悪の場合、virtual bond の大きさは、指数的に大きい。

→ 多くの物理系、データにおいて、小さな次元ですむことがある。

例: 1次元量子系 離れた場所の間での「量子エンタングルエントロピー」が小さい

ITensors.jl

Flatiron instituteで開発が進められている

M. Fishman, S. R. White, E. Miles Stoudenmire, SciPost Phys. Codebases 4 (2022)

- MPSからより汎用的なテンソルネットワークまで
- 階層構造
 - バックエンド: NDTensors.jl (型安定性を保った高速なライブラリ)
 - MPSコード: 動的で柔軟なコード
- GPU対応 (multi GPU?)

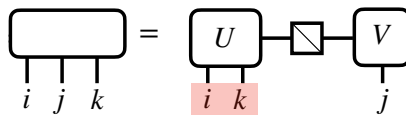
```
i = Index(3, "i")
j = Index(4, "j")
k = Index(5, "k")

T = randomITensor(i, j, k)

U, S, V = svd(T, (i, k))

@show norm(U*S*V-T)
```

$$A = U\Sigma V^\dagger$$



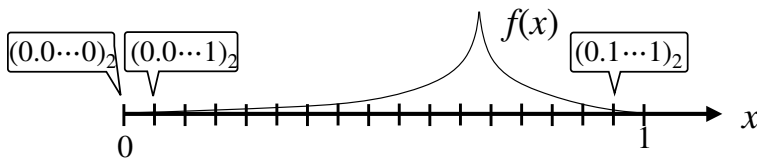
<https://itensor.github.io/ITensors.jl/stable/examples/ITensor.html>

QTTのまとめ (1/3ページ)

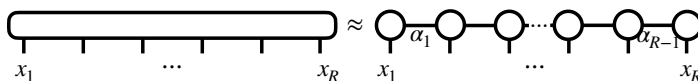
I. V. Oseledets, Doklady Math. **80**, 653 (2009)
 B. N. Khoromskij, Constr. Approx. **34**, 257 (2011)

Scalar-valued function $f(x)$ with $0 \leq x < 1$

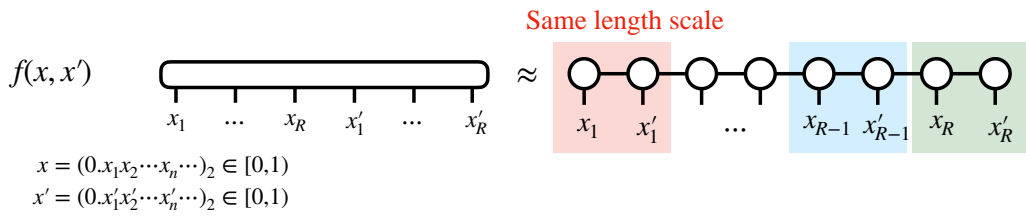
Binary coding: $x = (0.x_1x_2\dots x_R)_2 = x_1/2 + x_2/2^2 + \dots + x_R/2^R$



$$f(x_1, \dots, x_R) \approx \sum_{\alpha_1=1}^{D_1} \dots \sum_{\alpha_{R-1}=1}^{D_{R-1}} F_{x_1, \alpha_1}^{(1)} F_{x_1, \alpha_1, \alpha_2}^{(2)} \dots F_{x_R, \alpha_{R-1}}^{(R)}$$



QTTのまとめ (2/3ページ)



同程度の長さスケールの変数が、強くエンタングルしている場合

QTTのまとめ (3/3ページ)

指数関数

$$f(x) = e^{-x} = e^{-x_1/2} e^{-x_2/2^2} \dots e^{-x_n/2^n} \dots \quad D = 1$$

$$x = (0.x_1x_2 \dots x_n \dots)_2 \in [0,1)$$

The Sum of N exponential functions can be represented as a QTT of rank at most N .
 ∴ Bond dimensions are added when MPSs are added.

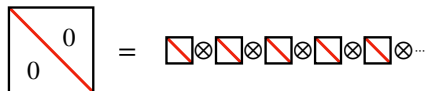
多項式

$$D \leq 1 + p$$

M. Ali and A. Nouy, "A. Approximation Theory of Tree Tensor Networks: Tensorized Univariate Functions", Constr Approx (2023)

単位行列

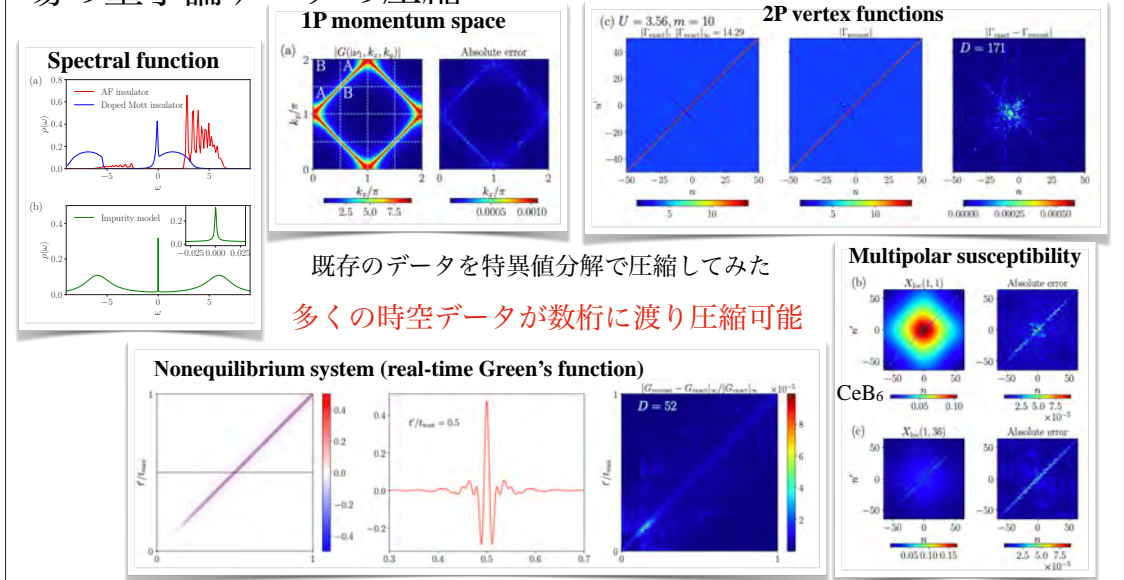
$$f(x, y) = \delta_{x,y} = \delta_{x_1,y_1} \delta_{x_2,y_2} \dots \quad D = 1$$



Representation of Continuous Functions: <https://tensornetwork.org/functions/> Written by Miles Stoudenmire

場の量子論データの圧縮

HS et al., Physical Review X 13, 021015 (2023)



圧縮したまま(場の量子論)計算ができるか？

R を増やすことで、指数的な解像度向上 (計算量は R に線形)

フーリエ変換

$$F(r) = \int dk \hat{F}(k) e^{ikr}$$

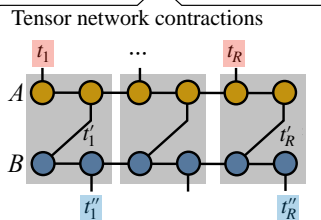
量子フーリエ変換回路=古典コンピュータで効率的に実行可能

K. J. Woolfe et al., Quantum Inf. Comput. 17, 1 (2017), J. Chen et al., arXiv:2210.08468v1

行列積

$$C(t, t'') = \int dt' A(t, t') B(t', t'')$$

$$C(t_1, t_1'', \dots, t_R, t_R'') = \sum_{t_1', \dots, t_R'} A(t_1, t_1', \dots, t_R, t_R') B(t_1', t_1'', \dots, t_R', t_R'')$$



ほぼ任意のダイアグラム計算が実行可能

Juliaライブラリの実装 MSSTA.jl (multiscale space-time ansatz)

まだ、実験的なライブラリです (○ユニットテスト、△ドキュメンテーション)

<https://github.com/shinaoka/MSSTA.jl>

- フーリエ変換などの演算を簡単に実装可能なライブラリ
- バックエンドとして、ITensors.jlを利用

Juliaをなぜ選んだのか？

- forループの使用が避けられない
- 戦略的にJuliaで場の量子論計算のエコシステムを作る
- (ITensors.jlの開発者と交流があった)

VS Codeを使ってデモしてみます！

QTTの展望

- 分散並列計算、GPU対応
- より複雑なテンソルネットワークへの拡張
- 皆さんの周りに、桁違いの長さスケールが共存する難しい問題ありませんか？
QTTで非線型常微分方程式も解けます Cahn-Hilliard方程式?
- Quantics Tensor Cross Interpolation (TCI): 特異値分解の代替
M. K. Ritter, Y. N. Fernández, M. Wallerberger, J. von Delft, HS, X. Waintal, arXiv:2303.11819

宣伝: ICIAM2023



Quantics表現の宣伝予定!

[00789] Algorithmic advances in computational quantum mechanics

- **Session Date & Time :**
 - 00789 (1/3) : 3C (Aug.23, 13:20-15:00)
 - 00789 (2/3) : 3D (Aug.23, 15:30-17:10)
 - 00789 (3/3) : 3E (Aug.23, 17:40-19:20)
- **Type :** Proposal of Minisymposium
- **Abstract :** Chemistry, physics, and materials science have benefited tremendously from advances in algorithmic tools for the simulation of quantum systems. In recent years, ideas developed in collaboration with the applied mathematics community have played an increasingly prominent role. This minisymposium will focus on recent algorithmic advances in computational quantum mechanics driven by numerical linear algebra, numerical methods for partial differential equations and integral equations, fast algorithms for the manipulation of structured operators, convex optimization, tensor networks, randomized algorithms, and machine learning methods.
- **Organizer(s) :** Jason Kaye, Michael Lindsey
- **Classification :** 81-08, 65Z05, Computational Quantum Physics
- **Speakers Info :**
 - Jason Kaye (Flatiron Institute, Simons Foundation)
 - Chao Yang (Lawrence Berkeley National Laboratory)
 - Hiroshi Shinaoka (Saitama University)
 - Fabian Faulstich (University of California, Berkeley)
 - Yuehaw Khoo (The University of Chicago)
 - Eloise Letournel (INRIA Paris)
 - Kangbo Li (Cornell University)
 - Miles Stoudenmire (Flatiron Institute, Simons Foundation)

MultiFloats.jl vs DoubleFloats.jl

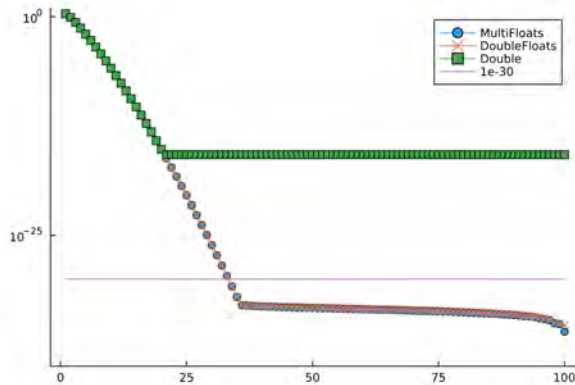
ヒルベルト行列

$$H_{ij} = 1/(i+j-1)$$

$$\begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix}$$

共に疑似四倍精度浮動小数点演算を実装

- SVDは、MultiFloats.jlの方が数割速い
- MultiFloats.jlは、四倍精度より高次の実装を含む



MultiFloats.jl 39 ms

```
n = 100
Hmf = [1/Float64x2(i+j-1) for i=1:n, j=1:n] # MultiFloats
u_mf, s_mf, vt_mf = svd(Hmf)
```

DoubleFloats.jl 62 ms

```
Hdd = [1/Double64(i+j-1) for i=1:n, j=1:n] # DoubleFloats
u_dd, s_dd, vt_dd = svd(Hdd)
```

cf. Float64 0.5 ms 二桁の速度差

Julia v1.91, MultiFloats v1.0.3, DoubleFloats v1.2.4
on Apple M1 Pro (1 thread)

虚時間形式の相関関数とは？

相関関数

$$\langle A(r, t)B(r', t') \rangle$$

違う時空点間での物理量の相関を測る。
久保公式によって、物理系の外場応答と対応。

長時間極限 $|t-t'|$ で速く減衰しない。→数値的に圧縮しづらい

虚時間形式

(松原形式とも呼ぶ) 数値的なダイアグラム計算でよく用いられる。

$$t = i\tau \quad \text{時間を仮想的に純虚数にする}$$

長時間極限 $|\tau-\tau'|$ で指数的に減衰する。→数値的に圧縮し易い(が、情報も一部失われている)

“スパースモデリング” = 虚時間での最適基底の構成

応用の拡がり

量子多体計算から第一原理計算まで

Analytic continuation

Sparse modeling (SpM)

- J. Otsuki, M. Ohzeki, **HS**, K. Yoshimi, *PRE* **95**, 061302(R) (2017)
- [Application to lattice QCD] E. Itou and Y. Nagai, *JHEP* **07**, 007 (2020)

Nevalinna/Carathéodory formalism

- J. Fei, C.-N. Yeh, E. Gull, *PRL* **126**, 056402 (2021)
- J. Fei, C.-N. Yeh, D. Zgid, and E. Gull, *PRB* **104**, 165111 (2021)

All electron GW calculations

- K. Haule, S. Mandal, *arXiv:2008.07727*

FLEX

- N. Witt, E. G. C. P. van Loon, T. Nomoto, R. Arita, T. O. Wehling, *PRB* **103**, 205148 (2021)
- N. Witt, J. M. Pizarro, T. Nomoto, R. Arita, T. O. Wehling, *arXiv:2108.01121v1*

Estimation of magnetic interactions

- T. Nomoto, T. Koretsune and R. Arita, *PRB* **102**, 014444 (2020)
- T. Nomoto, T. Koretsune and R. Arita, *PRL* **125**, 117204 (2020)
- Y. Nomura, T. Nomoto, M. Hirayama and R. Arita, *PRR* **2**, 043144 (2020)

Migdal-Eliashberg theory

- T. Wang, T. Nomoto, Y. Nomura, **HS**, J. Otsuki, T. Koretsune, and R. Arita, *PRB* **102**, 134503 (2020)

Self-energy embedding calculations

- S. Isakov, C.-N. Yeh, E. Gull, and D. Zgid, *PRB* **102**, 085105 (2020)
- C. N. Yeh, S. Isakov, D. Zgid, and E. Gull, *PRB* **103**, 195149 (2021)

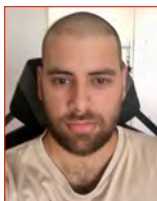
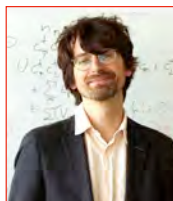
Extension to the two-particle level

- **HS**, J. Otsuki, M. Ohzeki, K. Yoshimi, K. Haule, M. Wallerberger, E. Gull, *PRB* **97**, 205111 (2018)
- **HS**, D. Geffroy, M. Wallerberger, J. Otsuki, K. Yoshimi, E. Gull, J. Kuneš, *SciPost Phys.* **8**, 012 (2020)
- M. Wallerberger*, **HS***, A. Kauch, *PRR* **3**, 033168 (2021)

独自実装や、Pythonライブラリの使用も含まれます

SparseIR.jl

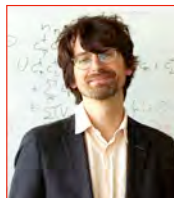
M. Wallerberger, S. Badr, S. Hoshino, S. Huber, F. Kakizawa, T. Koretsune, Y. Nagai, K. Nogaki, T. Nomoto, H. Mori, J. Otsuki, S. Ozaki, T. Plaikner, R. Sakurai, C. Vogel, N. Witt, K. Yoshimi, H. Shinaoka, *SoftwareX* **21**, 101266 (2023)



<https://github.com/SpM-lab/SparseIR.jl>

- 基底関数の区分多項式 (Legendre) による展開: 特異分解→特異値分解
- 疑似四倍精度演算による特異値分解 MultiFloats.jl + GenericLinearAlgebra.jl
- 基底の構成が終われば、後は倍精度演算で十分

Python実装 (sparse-ir) との比較



Pure Python 実装: <https://github.com/SpM-lab/sparse-ir>

numpyの疑似四倍精度浮動小数点拡張 <https://github.com/tuwien-cms/xprec>

C言語 + Python

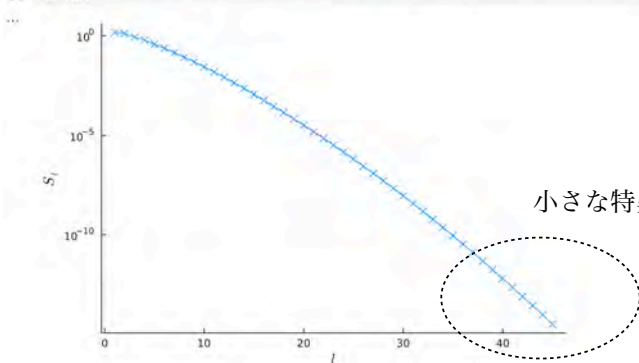
M. Wallerberger, S. Badr, S. Hoshino, S. Huber, F. Kakizawa, T. Koretsune, Y. Nagai, K. Nogaki, T. Nomoto, H. Mori, J. Otsuki, S. Ozaki, T. Plaikner, R. Sakurai, C. Vogel, N. Witt, K. Yoshimi, H. Shinaoka, SoftwareX **21**, 101266 (2023)

パッケージのリリース時に、バイナリーの自動ビルド+登録

- condaとPyPIを両方メンテナンスする必要
- ビルドに使うGitHub Actionsのランナーとcondaの互換性 (glibcのバージョンなど)
- 普通の物理学者、学生にはメンテナンスが難しい...

SparseIR.jlのデモンストレーション

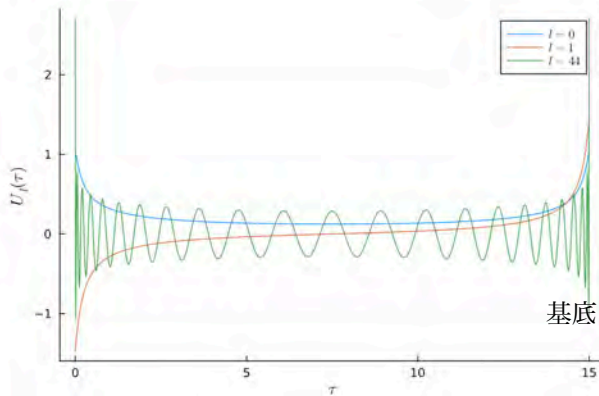
```
1 using SparseIR
2 using Plots
3 using LaTeXStrings
4
5 # 基底を作成
6 beta = 15.0
7 wmax = 10.0
8 basis = FiniteTempBasis(Fermionic(), beta, wmax, 1e-15)
9
10 plot(basis.s, title="", xlabel="l", ylabel="S_l", label="", axes=:log, marker=:x)
```



小さな特異値まで精度良く計算

SparseIR.jlのデモンストレーション

```
1 p = plot(title="", xlabel="\tau", ylabel="U_l(\tau)", label="")
2 ts = collect(LinRange(0, beta, 1000))
3 for l in [0, 1, length(basis.u)-1]
4     plot!(p, ts, basis.u[l+1](ts), label="l=%$(l)")
5 end
6 p
```



チュートリアル集

- 典型的な計算のサンプルコード
- Jupyter Book
- 手元でのビルド環境は、VS Code + Dockerで用意

<https://spm-lab.github.io/sparse-ir-tutorial/>

Basic theory

- intermediate representation (IR)
- Sparse sampling
- Discrete Leifmann representation

Reference

- API reference
- Additional material

Sample codes

- Transformation from IR
- Sparse sampling
- Discrete Leifmann Representation
- Second-order perturbation
- The GF2 & GW method
- DMFT with IR (QMC)
- FLUX approximation
- TPSC approximation
- Eliashberg theory for Holstein-Hubbard model
- Orbital magnetic susceptibility
- Estimation of exchange interaction

Welcome to sparse-ir tutorial

This online document provides comprehensive tutorials of the Python library `sparse-ir` library as well as its port to Julia and Fortran.

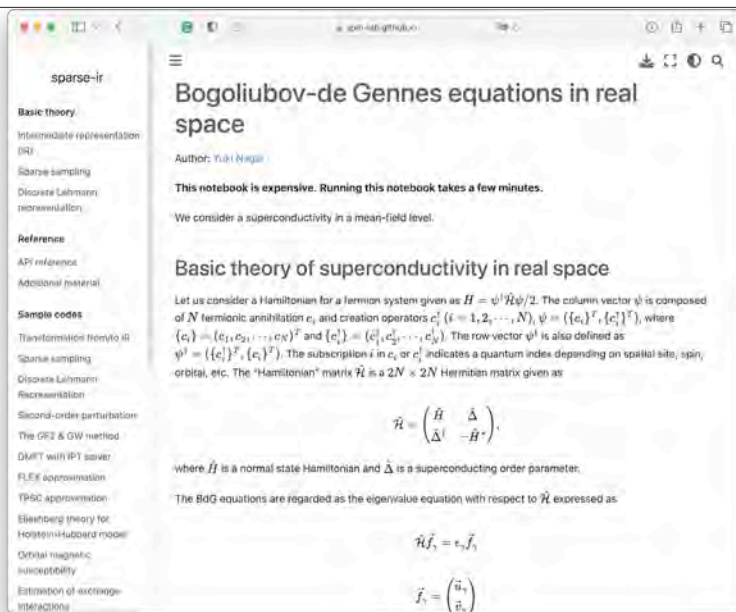
You can run jupyter notebooks online on [myBinder!](#)

Please free to make a contact with the authors of each tutorial page if you have any questions!

Contributors

- Markus Wallerberger (TU Wien)
- Samuel Badr (TU Wien)
- Shintaro Hoshino (Saitama Univ.)
- Fumiya Kakizawa (Saitama Univ.)
- Takashi Koretsune (Tohoku Univ.)
- Yuki Nagai (JAEA, Riken)
- Kosuke Nogaki (Kyoto Univ.)
- Takuya Nomoto (Univ. Tokyo)
- Hitoshi Mori (Riken)
- Junya Otsuki (Okayama Univ.)
- Soehun Ozaki (Univ. Tokyo)
- Rihito Sakurai (Saitama Univ.)
- Constanze Vogel (TU Wien)
- Niklas Witt (Univ. Hamburg)
- Kazuyoshi Yoshimi (ISSP)
- Hiroshi Shinacka (Saitama Univ.)

解説 + コード例
by 永井さん



最後に: Juliaによる計算物理の展望

Juliaを使うことで、試行錯誤のスピードを上げることが出来た！

今後挑戦したい

- テンソルネットワークの分散並列計算 Distributed vs MPI
- Multi GPUs

研究員募集中!

- 学術変革領域研究B「量子古典融合アルゴリズムが拓く計算物質科学」(領域代表品岡: 2023~2025年)
- JST創発「2粒子レベルの量子埋め込み理論に基づく新規第一原理計算手法の開発と実証」(2024~2026年: フェーズ1、2027~2030年: フェーズ2)
- 3年任期 **Juliaを使った時空圧縮技術の開発と応用、物性物理外からの挑戦も歓迎!**
- 特任助教の名称の付与可
- 助教級の待遇 (1年目: 480~530万円、2年目: 530~580万円)
- 別途、家賃補助、通勤手当あり

数論における Julia の援用

IMI 共同利用「数学と物理における Julia の活用」



横山 俊一 (東京都立大学)

s-yokoyama@tmu.ac.jp

2023年7月11日 ※ 本日38歳になりました

内容の一部は JSPS 科研費・基盤(C) 20K03537

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

(代表：横山 俊一) の援助を受けています



代数系（数論）における Julia の立ち位置

- 幾何系や解析系に比べると 活用状況・認知度ともに低い 状況
- 既存の CAS 開発においては、1つの分野（代数的対象物）に特化したものが多く、長期間使い込んだユーザが異なるプラットフォームに移行するのは困難
例：Magma (C, 次頁)、Sage (Python, 次々頁)、Pari/GP (C) etc.
- そもそも Julia 言語で書かれたパッケージはまだ少数
- それでも開発速度は徐々に上がってきている：
例：AbstractAlgebra.jl, Gap.jl, GroebnerBasis.jl etc.
- 正しく実装すれば、多項式代数においても高速

Magma

- 豪 シドニー大学 を中心として開発運営が進められている
計算代数システム。前身は Cayley (1982) で、
正式ローンは 1993 年
有償だがプロプライエタリライセンス (非営利)
- 主として 数論および計算機代数に特化 した関数群
圏論 に基づいた設計思想
- 2013年から Intel AVX 演算に対応 (Linux 64bit OS のみ)
- Bosma-Cannon-Playoust, *The Magma algebra system I: the user
language* (CANT/London, 1997)



Sage/CoCalc/SageMathCell

- 米 ワシントン大学 を中心として開発運営が
進められて「いた」汎用数式処理システム
- 数論幾何学の研究者 William Stein が開発責任者
1990年代には Magma の HECKE パッケージ (モジュラー形式の計算) の
開発に従事、2005年の正式ローンチの際に Python へ移植
- web ベースの UI 開発の変遷：
Sage Notebook -> salv.us -> SageMathCloud -> CoCalc
SageMathCell は CoCalc の簡易版
- 2019年に大学を退職し、SageMath Inc. CEO/ファウンダーとして運営



Nemo

- 独 [カイザースラウテルン工科大学](#) を中心として開発運営が進められている、Julia で開発された数論計算パッケージおよび C/C++ wrapper 集



- Singular の開発本拠地でもある (→ Singular.jl pkg)
- 中心メンバ : Claus Fieker, William Hart, Tommy Hofmann, Frederik Johansson, Marek Kaluba, Carlo Sircana
- FHHJ, [Nemo/Hecke: Computer Algebra and Number Theory Packages for the Julia Programming Language](#), ISSAC'17.

Nemo

- 2018年、Nemo は2つのパッケージに分離 :
[Nemo.jl](#) Flint, Arb, Antic 限定 wrapper
[AbstractAlgebra.jl](#) 上記外の generic なパッケージ



- Julia 単独でも
 - Z, Q, R, C (Flint/Arb)
 - 行列計算, FFT, 並列計算 etc.などが扱えるが、これに加えて整数論に特化した機能をもつ :
 - [代数体](#) (Antic) , [有限体](#) (Flint)
 - [1変数/多変数多項式環](#) (generic)
 - [p進体・関数体](#) (Flint, finite precision)

Nemo

- Flint を用いることなく generic に生成されたもの
(例：係数環 R) は `AbstractAlgebra.jl` に格納



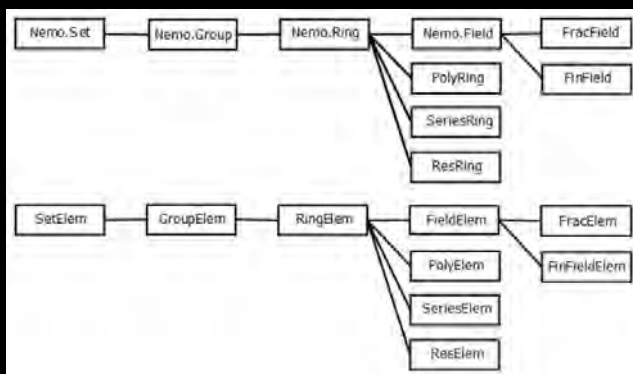
- Flint や Arb 単独で高速処理できるものは、すべて Julia 言語で実行するだけ
- より高速な実装を目指すならば、まず `AbstractAlgebra.jl` のリストを確認

- 例： $S = \mathbb{Q}_p$ (p 進数体, precision m)

```
function O(R::FlintPadicField, m::fmpz)
    if isone(m)
        N = 0
    else
        p = prime(R)
        if m == p
            N = 1
        else
            N = flog(m, p)
            p^(N) != m && error("Not a power of p in p-adic O()")
        end
    end
    d = padic(N)
    d.parent = R
    return d
end
```

$m=p^r$ 以外の $prec$ が代入されるとエラーを返す

Abstract-type hierarchy



- 例えば xxx.Field は xxx.Ring に属する
 - Magma では異なる category として処理
 - Sage では構造が 逆 : xxx.ring は xxx.Field の部分集合

Hecke

- Nemo (と AbstractAlgebra パッケージ) に準じて開発されている **整数論** に特化した計算パッケージ
- 1990 年代に William Stein (Sage/CoCalc 開発者) が Magma で開発していた、モジュラー形式に関するパッケージ HECKE とは別物
- <https://github.com/thofma/Hecke.jl> (ソースコード)
- <https://thofma.github.io/Hecke.jl/latest/> (ドキュメント)

```

(@v1.5) pkg> add Primes
  Updating registry at `~/julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
  Resolving package versions...
Updating `~/julia/environments/v1.5/Project.toml`
 [27ebfcd6] + Primes v0.5.0
Updating `~/julia/environments/v1.5/Manifest.toml`
 [27ebfcd6] + Primes v0.5.0

julia> using Primes
[ Info: Precompiling Primes [27ebfcd6-29c5-5fa9-bf4b-fb8fc14df3ae]

julia> factor(20200811)
17^2 * 69899

julia> @time factor(37837492843298492383902389201839237982374892374)
 0.000457 seconds (49 allocations: 3.492 KiB)
2 * 13 * 1455288186280711245534707276993816845475957399

```

```

julia> @time factor(155091189582461323967422273374060483191006034694003396144697
71087037330452965523211078730059982890210854424804938137751121970242869392787964
3018958702811151575938519984102178816)
^ERROR: InterruptException:
Stacktrace:
 [1] BigInt at ./gmp.jl:56 [inlined]
 [2] sub(::BigInt, ::BigInt) at ./gmp.jl:158
 [3] - at ./gmp.jl:476 [inlined]
 [4] pollardfactors!(::BigInt, ::Primes.Factorization{BigInt}) at /Users/s-yokoyama_lab/.julia/packages/Primes/eRBAQ/src/Primes.jl:415
 [5] pollardfactors!(::BigInt, ::Primes.Factorization{BigInt}) at /Users/s-yokoyama_lab/.julia/packages/Primes/eRBAQ/src/Primes.jl:431 (repeats 3 times)
 [6] factor!(::BigInt, ::Primes.Factorization{BigInt}) at /Users/s-yokoyama_lab/.julia/packages/Primes/eRBAQ/src/Primes.jl:271
 [7] factor at /Users/s-yokoyama_lab/.julia/packages/Primes/eRBAQ/src/Primes.jl:304 [inlined]
 [8] top-level scope at ./timing.jl:174 [inlined]
 [9] top-level scope at ./REPL[9]:0

```


Magma の Pollard's Rho + ECM

```
> Factorization(15509118958246132396742227337406048319100603469400339614469771\
087037330452965523211078730059982890210854424804938137751121970242869392787964\
3018958702811151575938519984102178816);
Integer main factorization (primality of factors will be proved)
Effort: 3
Seed: 260009057 0
  Number: 1550911895824613239674222733740604831910060346940033961446977108703\
733045296552321107873005998289021085442480493813775112197024286939278796430\
18958702811151575938519984102178816

Pollard Rho
  Trials: 8191
  Number: 2316616648442896824810580101804251439783171605295524330506899425233\
74022958870415809752076217738879615251
  (105 digits)
  Factor: 21727894856911 (14 digits)
  Cofactor: 10661947067117961789114870291074345630713615031862630703290949593\
900960465823539183339042941 (92 digits)
  Time: 0.009
```

```
Pollard Rho
  Trials: 8191
  Number: 1066194706711796178911487029107434563071361503186263070329094959390\
0960465823539183339042941
  (92 digits)
  No factor found
  Time: 0.000
```

1 composite number remaining

ECM

```
x: 106619470671179617891148702910743456307136150318626307032909495939009604\
65823539183339042941
  (92 digits)
  Initial B1: 5000, limit: 858248
  Initial Pollard p - 1, B1: 45000
  Step 1; B1: 5000 [858248], digits: 92, elapsed time: 0.020
  Factor: 146234082633259 (15 digits)
  Cofactor: 72910137466770304935106535275228904632898328810503066577055464788\
170201591799 (77 digits)
  Total ECM time: 0.109
```


ECM

```
x: 10846044215834722665042065816010288453465430807005605353874509
(62 digits)
Initial B1: 6341, limit: 17552
Step 1; B1: 6341 [17552], digits: 62, elapsed time: 0.000
Step 10; B1: 7072 [17552], digits: 62, elapsed time: 0.160
Step 20; B1: 7932 [17552], digits: 62, elapsed time: 0.360
Step 30; B1: 8842 [17552], digits: 62, elapsed time: 0.580
Step 40; B1: 9802 [17552], digits: 62, elapsed time: 0.820
Step 50; B1: 10809 [17552], digits: 62, elapsed time: 1.100
Step 60; B1: 11864 [17552], digits: 62, elapsed time: 1.399
Step 70; B1: 12969 [17552], digits: 62, elapsed time: 1.710
Step 80; B1: 14124 [17552], digits: 62, elapsed time: 2.060
Step 90; B1: 15329 [17552], digits: 62, elapsed time: 2.430
Factor: 75045259055838983 (17 digits)
Cofactor: 144526707646708212356380247022426585248301323 (45 digits)
Total ECM time: 2.740
```

Total time: 3.139

```
[ <2, 206>, <67, 2>, <271, 1>, <5351, 1>, <3726911, 1>, <5830001, 1>,
<146234082633259, 1>, <6722279202985811, 1>, <75045259055838983, 1>,
<144526707646708212356380247022426585248301323, 1> ]
```

Nemo の Pollard's Rho + ECM

```
julia> using Nemo
```

```
Welcome to Nemo version 0.17.6
```

```
Nemo comes with absolutely no warranty whatsoever
```

```
julia> factor(ZZ(20200811))
```

```
1 * 69899 * 17^2
```

```
julia> @time factor(ZZ(155091189582461323967422273374060483191006034694003396144
69771087037330452965523211078730059982890210854424804938137751121970242869392787
9643018958702811151575938519984102178816))
```

```
0.941467 seconds (97.52 k allocations: 9.656 MiB, 1.21% gc time)
```

```
1 * 3726911 * 5830001 * 67^2 * 2^206 * 6722279202985811 * 5351 * 146234082633259
* 75045259055838983 * 144526707646708212356380247022426585248301323 * 271
```

注：Magma ではさらに MPQS が使える

```

julia> Qx,x=PolynomialRing(FlintQQ,"x");
julia> K1,a=NumberField(x^2-2,"a")
(Number field over Rational Field with defining polynomial x^2-2, a)
julia> K2,b=NumberField(x^2-3,"b")
(Number field over Rational Field with defining polynomial x^2-3, b)
julia> isisomorphic(K1,K2)
(false, Injection of Number field over Rational Field with defining polynomial x
^2-2 into Number field over Rational Field with defining polynomial x^2-3
defined by a -> 0
)
julia> K,g=number_field([x^2-2,x^2-3])
(Non-simple number field with defining polynomials fmpq_mpoly[x1^2 - 2, x2^2 - 3
], NfAbsNSElem[_$1, _$2])
julia> g[2]^2
3
julia> minpoly(g[1]+g[2])
x^4-10*x^2+1

```

```

julia> Qx,x=PolynomialRing(FlintQQ,"x");
julia> f=x^2-10;
julia> k,a=NumberField(f,"a");
julia> @time Ok=maximal_order(k)
13.223396 seconds (27.29 M allocations: 1.298 GiB, 4.86% gc time)
Maximal order of Number field over Rational Field with defining polynomial x^2-1
0
with basis nf_elem[1, a]
julia> g=x^3-14;
julia> l,b=NumberField(g,"b");
julia> @time Ol=maximal_order(l)
0.000176 seconds (585 allocations: 29.828 KiB)
Maximal order of Number field over Rational Field with defining polynomial x^3-1
4
with basis nf_elem[1, b, b^2]

```

```

julia> c,mc=class_group(k)
(GrpAb: Z/2, ClassGroup map of
Set of ideals of OK
)

julia> A=ray_class_field(mc)
Class field defined mod (<1, 1>, InfPlc[]) of structure c

julia> K=number_field(A)
non-simple Relative number field over
Number field over Rational Field with defining polynomial x^2-10
with defining polynomials AbstractAlgebra.Generic.MPoly{nf_elem}[_$1^2-2]

```

```

julia> @time ZK=maximal_order(K)
0.000007 seconds (1 allocation: 48 bytes)
Relative maximal order of non-simple Relative number field over
Number field over Rational Field with defining polynomial x^2-10
with defining polynomials AbstractAlgebra.Generic.MPoly{nf_elem}[_$1^2-2]
with pseudo-basis
(1, 1//1 * <1, 1>)
(_$1+a, 1//4 * <2, a>)

julia> discriminant(ZK)
<1, 1>
Norm: 1
Minimum: 1
two normal wrt: 2

```

Magma's NF-CG-RCF-NF

```
> Qx<x>:=PolynomialRing(Rationals());
> k<a>:=NumberField(x^2-10);
> c,mc:=ClassGroup(k); c; mc;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
  2*c.1 = 0
Mapping from: GrpAb: c to Set of ideals of Maximal Equation Order with defining
polynomial x^2 - 10 over its ground order
> A:=RayClassField(mc); A;
FldAb, defined by (<[1, 0]>, [])
of structure: Z/2
> K:=NumberField(A); K;
Number Field with defining polynomial $.1^2 - 2 over k
> time ZK:=MaximalOrder(K);
Time: 0.020
> ZK;
Maximal Order of Equation Order with defining polynomial x^2 + [-2, 0] over its
ground order
> Discriminant(ZK);
Principal Ideal
Generator:
  [1, 0]
```

ベンチマーク例 (from NemoCas page)

Fateman benchmark

- $f = 1 + x + y + z + t$
- $p = f^{30}$
- time $q = p \cdot (p + 1)$

This benchmark tests generic polynomial arithmetic. Note that we don't have a dedicated multivariate polynomial module in Nemo yet, so this and the next benchmark are done with nested univariate polynomial rings. Where possible we do the same thing in the other systems we time (Pari/GP is an exception).

SageMath 6.8	Pari/GP 2.7.4	Magma V2.21-4	Nemo-0.3
132s	156s	233s	44s

ベンチマーク例 (from NemoCas page)

Magma:

```
> R<x>:=PolynomialRing(Integers());
> S<y>:=PolynomialRing(R);
> T<z>:=PolynomialRing(S);
> U<t>:=PolynomialRing(T);
> time f:=(x+y+z+t+1)^30;
> time g:=f*(f+1);
```

Nemo:

```
> R,x=PolynomialRing(FlintZZ,"x");
> S,y=PolynomialRing(R,"y");
> T,z=PolynomialRing(S,"z");
> U,t=PolynomialRing(T,"t");
> @time f=(x+y+z+t+1)^30;
> @time g=f*(f+1);
```

実はほぼ同一コード

ベンチマーク例 (from NemoCas page)

Pearce benchmark

- $f = (x + y + 2z^2 + 3t^3 + 5u^5 + 1)^{16}$
- $g = (u + t + 2z^2 + 3y^3 + 5x^5 + 1)^{16}$
- time $q = f \cdot g$

This benchmark is usually done with sparse multivariate polynomial arithmetic. We use it here with our recursive dense polynomial arithmetic (there is no sparse polynomial representation in Nemo yet).

Note: this is not fair to other packages that implement sparse multivariate polynomial arithmetic, as they could complete this benchmark faster if we used their sparse arithmetic. But that would not be a very meaningful benchmark if the aim is to measure the relative performance of the recursive dense polynomial arithmetic in Nemo. On the other hand, Pari/GP uses a sparse-recursive representation (similar to recursive-dense, but with dedicated zero objects in each ring).

SageMath 6.8	Pari/GP 2.7.4	Magma V2.21-4	Nemo-0.3
2900s	798s	647s	167s

ベンチマーク例 (from NemoCas page)

Resultant benchmark

- $R = GF(17^{11})$
- $S = R[y]$
- $T = S/(y^3 + 3x^*y + 1)$
- $U = T[z]$
- $f = T(3y^2 + y + x)^*z^2 + T((x + 2)^*y^2 + x + 1)^*z + T(4x^*y + 3)$
- $g = T(7y^2 - y + 2x + 7)^*z^2 + T(3y^2 + 4x + 1)^*z + T((2x + 1)^*y + 1)$
- $s = f^{12}$
- $t = (s + g)^{12}$
- time $r = \text{resultant}(s, t)$

This benchmark is designed to test generics, division and computation of the resultant.

SageMath 6.8	Pari/GP 2.7.4	Magma V2.21-4	Nemo-0.3
179907s	N/A	82s	0.2s

ベンチマーク例 (from NemoCas page)

Polynomials over number fields

- $R, x = \text{CyclotomicField}(20)$
- $S = R[y]$
- $f = (3x^7 + x^4 - 3x + 1)^*y^3 + (2x^6 - x^5 + 4x^4 - x^3 + x^2 - 1)^*y + (-3x^7 + 2x^6 - x^5 + 3x^3 - 2x^2 + x)$
- time $r = f^{300}$

This benchmark is designed to test dense polynomials over number fields.

SageMath 6.8	Pari/GP 2.7.4	Magma V2.21-4	Nemo-0.3
6.92s	0.21s	0.70s	0.13s

NemoCas Project

- <http://nemocas.org/> (Nemo/Hecke 開発ページ)
- Magma, Sage, Pari/GP と比較したいいくつかのベンチマーク結果が閲覧可能
- 最新版は開発者用 Git リポジトリから
 - Nemo: <https://github.com/Nemocas/Nemo.jl>
 - AbstractAlgebra: <https://github.com/Nemocas/AbstractAlgebra.jl>
 - Hecke: <https://github.com/thofma/Hecke.jl>
 - Singular: <https://github.com/oscar-system/Singular.jl/>
- 試しに使ってみたいが導入に失敗した方へ：
Nemo は [Jupyter Notebook](#) 経由で [Sage](#) 上で利用可能

OSCAR

OSCAR
SYMBOLIC TOOLS

The OSCAR project

Home
About
News
Installation
Tutorials
Examples
Documentation
Community
Talks
Meetings
Credits
Contributors

OSCAR Computer Algebra System

The OSCAR project develops a comprehensive **Open Source Computer Algebra Research** system for computations in algebra, geometry, and number theory, written in **Julia**. In particular, the emphasis is on supporting complex computations which require a high level of integration of tools from different mathematical areas.

The project builds on and extends the four cornerstone systems

- **GAP** - computational discrete algebra (via **GAP.jl** Julia package)
- **Singular** - commutative and non-commutative algebra, algebraic geometry (via **Singular.jl** Julia package)
- **Polymake** - polyhedral geometry (via **Polymake.jl** Julia package)
- **Antic (Hecke, Nemo)** - number theory

as well as further libraries and packages, which are tied together using [the Julia language](#) into the **Oscar.jl** Julia package.

The development of OSCAR is supported by the Deutsche Forschungsgemeinschaft DFG within the Collaborative Research Center TRR 195.

OSCAR

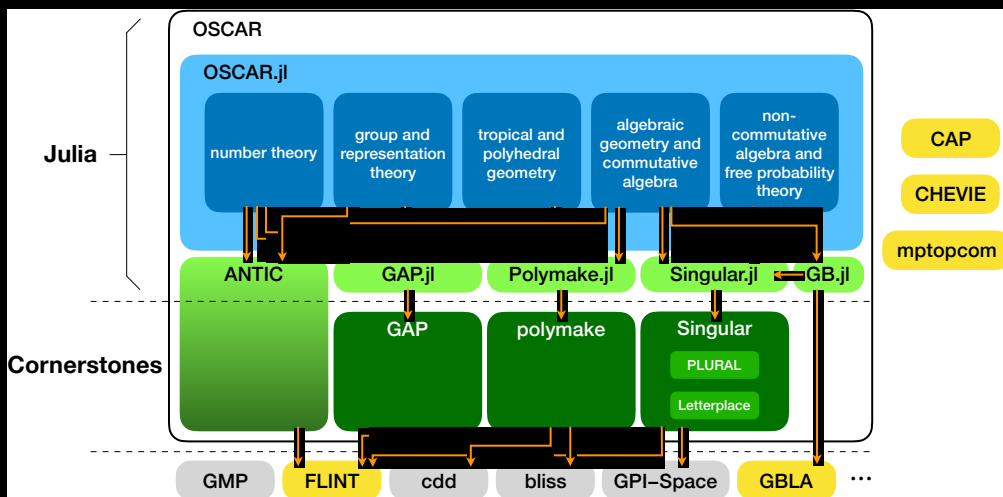
- 主として NemoCas プロジェクトから派生した、
Julia を用いた新計算代数システム開発プロジェクト
- GAP (Gap.jl) , Singular (Singular.jl) , Polymake (Polymake.jl) ,
Antic (Nemo.jl / Hecke.jl) の4ライブラリを主軸に構成
- プロジェクトリーダー：
 - Wolfram Decker (TU Kaiserslautern : NemoCas メンバ)
 - Claus Fieker (TU Kaiserslautern : NemoCas メンバ)
 - Max Horn (TU Kaiserslautern)
 - Michael Joswig (TU Berlin)
- DFG, TUK に研究所 TRR-195 を2017年に開設



OSCAR

```
Installed Oscar _____ v0.12.1
Downloaded artifact: libsingular_julia
Downloaded artifact: Antic
Downloaded artifact: Ncurses
Downloaded artifact: MongoC
Downloaded artifact: PPL
Downloaded artifact: GLPK
Downloaded artifact: Zstd
Downloaded artifact: Perl
Downloaded artifact: MUMPS_seq
Downloaded artifact: SPRAL
Downloaded artifact: FLINT
Downloaded artifact: OpenSSL
Downloaded artifact: bliss
Downloaded artifact: Hwloc
Downloaded artifact: AtkinModularPolynomialsDB
Downloaded artifact: ZLatDB
Downloaded artifact: ClassicalModularPolynomialsDB
Downloaded artifact: QuadLatDB
Downloaded artifact: HermLatDB
Downloaded artifact: SmallGroupDB
Downloaded artifact: Bzip2
Downloading artifact: polymake
Downloading [=====] ] 85.3 %
```


OSCAR



OSCAR



- 計算代数/数論における連携処理をシームレスに行う

Oscar.jl を提供

```

Precompiling project...
 74 dependencies successfully precompiled in 118 seconds. 8 already precompiled
.
julia> Using Oscar

OSCAR

...combining (and extending) ANTIC, GAP, Polymake and Singular
Version 0.132-1 ...
... which comes with absolutely no warranty whatsoever
Type: 'Oscar' for more information
(c) 2019-2023 by The OSCAR Development Team

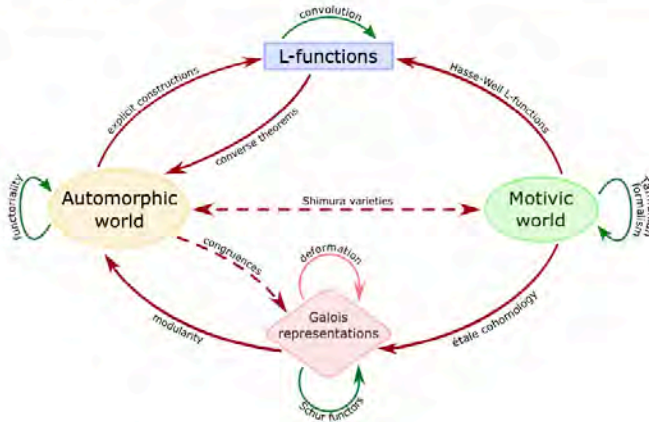
julia>
    
```

- 数論統合システム LMFDB 上のデモプログラムに OSCAR コードが新規採用

LMFDB

The LMFDB universe

The top half of the diagram is based on the [Langlands program](#), which predicts that any motivic object corresponds to an automorphic object via their L-functions.



LMFDB

Elliptic curve with LMFDB label 100.a2 (Cremona label 100a4)

Minimal Weierstrass equation Show commands: Magma / Oscar / PariGP / SageMath

$$y^2 = x^3 - x^2 - 908x - 15688 \quad (\text{homogenize, simplify})$$

```
oscar: E = EllipticCurve([0, -1, 0, -908, -15688])
oscar: short_weierstrass_model(E)
```

Mordell-Weil group structure

$$\mathbb{Z}/2\mathbb{Z}$$

Torsion generators

$$(37, 0)$$

```
oscar: torsion_structure(E)
```

Properties	
Label	100.a2
Conductor	100
Discriminant	-62500000000
j-invariant	$-\frac{20720484}{15625}$
CM	no
Rank	0
Torsion structure	$\mathbb{Z}/2\mathbb{Z}$

Related objects

OSCAR

- すべての情報は OSCAR の web ページから参照：

<https://www.oscar-system.org/>



- インストールが少し難しいので注意：
 - Julia に加えて **Xcode ツール** が必要 (Mac OS の場合)
 - Julia を立ち上げてから **Pkg ツール** を用いる
 - インストール・ダウンロード (アーティファクト)・プリコンパイルに少々時間がかかるので気長に待つ
- 終わったら using Oscar で、あとは楽しむ
ただしパス設定がシビアなので、慣れるまで頑張る
※ **Jupyter Notebook** 環境を使うと少しは導入がラク

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

1

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

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

現実存在するあらゆるものが対象

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 ->そもそもまともに計算できない

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

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

有名なFORTTRAN

FORTTRAN77 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。FORTTRAN77もたまに見かける
 大学学部で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

個別の研究でもFortranが多く使われている

WIEN2k

Fortranが使われている実例

少し古いけど 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: 'val0a'), 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ってきた。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

小さなコードは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

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）

先生がFortran(FORTRAN)使い

研究室秘伝のコードがFortran

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ツ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

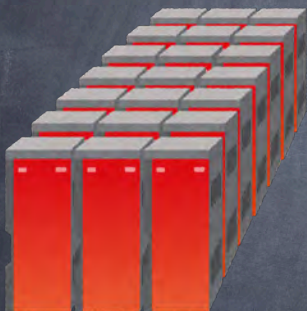
一度に大量の計算を行うこと

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

プロセス並列

人が多い計算



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



それぞれがそれぞれの仕事をする

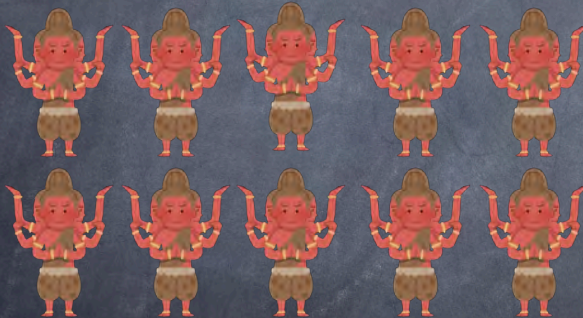
高速に行うには：
それぞれのデータのやり取りは時間がかかるので、通信を減らす

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

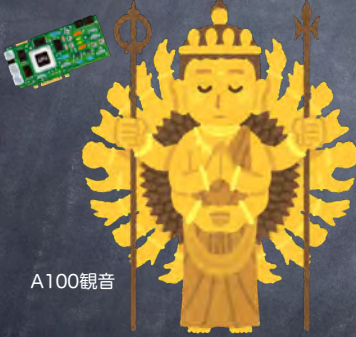
サーバー間：プロセス並列

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

マルチGPU並列



A100観音

GPU:シンプルな演算器
がすごくたくさんある



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

プロセス並列:この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



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

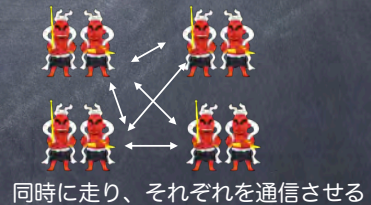
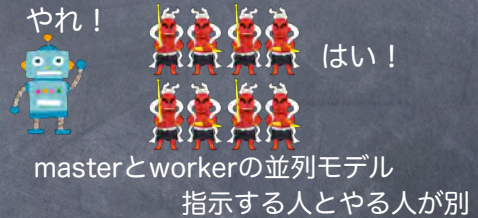
単純で一つ一つが重たい並列計算ならこれが楽

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



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： $Ax = b_i$ 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

$a = \text{pmap}(i \rightarrow \text{goodfunc}(i), 1:N)$ 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_{ij} は、 $e_i^T (z-H)^{-1} e_j$ 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-shift CG for $(\sigma_j + A)x_j = b$ with a hermitian matrix $A \in \mathbb{C}^{m \times m}$, $\forall \sigma_j \in \mathbb{C}^m$, $\sigma_j, r_j, p_j \in \mathbb{C}^m$, $\alpha_j(\sigma_j), \beta_j(\sigma_j), \rho_j(\sigma_j), \Sigma_j(\sigma_j), \Pi_j(\sigma_j), \Xi_j(\sigma_j) \in \mathbb{C}^m$ and $\alpha_j, \beta_j \in \mathbb{C}$, $\Sigma_j(\sigma_j) \in \mathbb{V}_j(\sigma_j)$, $V^T = (v_1, v_2, \dots, v_m)$, $v_i \in \mathbb{C}^m$. Here, the symbol T represents transposition.

1. Input $\sigma_j (j = 1, 2, \dots, N)$
2. Set $\alpha_j = 0, r_j, p_j = b, \alpha_{j-1} = 1, \beta_{j-1} = 0$
3. Compute $\Sigma_j = Vb$
4. Set $\Sigma_j(\sigma_j) = 0, \Pi_j(\sigma_j) = \Sigma_j, \rho_j(\sigma_j) = \rho_j(\sigma_j) = 1 (j = 1, 2, \dots, N)$
5. For $k = 0, 1, \dots$ until convergence Do:
6. $\alpha_k = (r_j, r_j) / (r_j, A r_j)$
7. $r_{k+1} = r_j + \alpha_k A r_j$
8. $\rho_{k+1} = r_j + \alpha_k A r_j$
9. $\beta_k = (r_{k+1}, r_{k+1}) / (r_k, r_k)$
10. $\rho_{k+1} = \beta_k \rho_k$
11. Compute $\Sigma_{k+1} = V r_{k+1}$
12. For $j = 1, 2, \dots, N$ Do:
13. $\rho_{j+1}(\sigma_j) = \rho_{j+1}(\sigma_j) + \alpha_k (r_j, r_{k+1}) / (r_j, A r_j) - \rho_j(\sigma_j)$
14. $\alpha_j(\sigma_j) = \rho_{j+1}(\sigma_j) / \rho_j(\sigma_j)$
15. $\Xi_{k+1}(\sigma_j) = \Sigma_{k+1}(\sigma_j) + \alpha_j(\sigma_j) \Pi_k(\sigma_j)$
16. $\rho_j(\sigma_j) = \left(\frac{\rho_{j+1}(\sigma_j)}{\rho_j(\sigma_j)} \right)^2 \beta_k$
17. $\Pi_{k+1}(\sigma_j) = \rho_{j+1}(\sigma_j) \Xi_{k+1} + \beta_k \Pi_k(\sigma_j)$
18. End Do
19. End Do

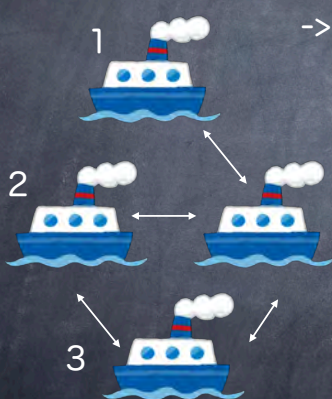
```

for k=0:maximumsteps
    mul!(Ap, A, -r)
    p1, m1, r1 = (p, A, -r)
    normr = dot(r, r)
    a = normr / dot(r, Ap)
    @. x += a * r
    @. r = -Ap * r
    @. p = r * r / normr
    @. p = r * p
    for j = 1:N
        if abs(p[j]) > eps
            pk[j] = p[j] * (pk[j] / norm(p[k])) + a * beta * (pk[j] - pk[j])
            dk[j] = a * pk[j] / (pk[j] * r1)
            R[j] = dk[j] * (1 + beta)
            pk[j, :1] = (pk[j] * (pk[j]^2) + beta * normr) / (pk[j] * (pk[j]^2) + beta * normr)
            pk[j] = pk[j]
            pk[j] = pk[j]
        end
    end
    m = a
    beta = beta
    h1 = real(normr * maximum(abs, pk))
    if h1 < eps
        return 0
    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)



-> CPUもメモリもそれぞれ別を持っている
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を集める
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

母艦を指定してもよいしみんな計算してもよい

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。でも新規参加者はコンパイルできなかつたり…

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

しなくても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れない

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、ということを示す必要がない

```
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x2 x272-> 13056コア
 CPU演算部:2.801PFLOPS
 Xeon Gold 20コアx2 x706-> 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 20 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 差分された2階微分演算子D+対角要素V

```
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
```

```
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/n)*imag(Gii)
  end
  ldosall = reshape(MPI.Allgather(ldos,comm),(nene,n))
  myrank = MPI.Comm_rank(comm)
end
```

$$H = D+V$$

$e_i^T (z - H)^{-1} e_i$ をRSCG.jlで計算

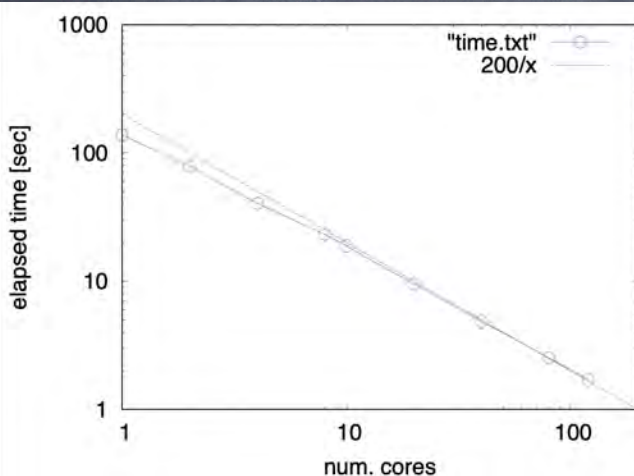
iのループを並列化

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 差分された2階微分演算子D+対角要素V



$$H = D+V$$

$e_i^T (z - H)^{-1} e_i$ をRSCG.jlで計算

iのループを並列化

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

並列計算について

MPI.j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れば
先行研究の技法が使えるのになあ……

機械学習

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



シミュレーション分野での機械学習の使用例

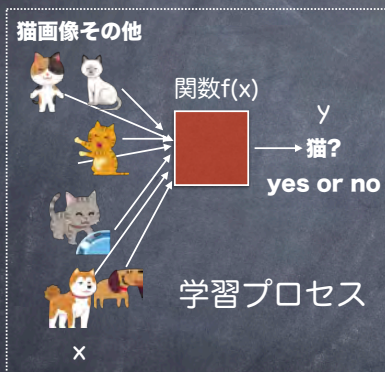
これまで計算が大変だった部分を機械が行う



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

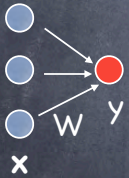
大量の入力データxを用いて、 $f(x)=y$ を満たす関数 $f(x)$ を決める

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

$$y = Wx + b$$



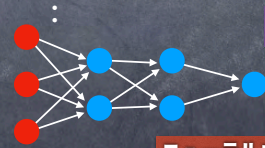
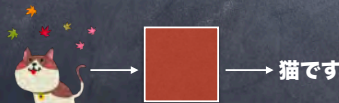
もっと表現力を高めたい

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

f: 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 θ と入力 x の関数 $y = f(\theta, x)$

“loss関数” $L(\theta, f)$ を最小化する θ 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))
    dfdx = gradient(model,x)[1]
    println(dfdx)
end
```

隠れ層2層の全結合ニューラルネットワーク

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           構造体を作る
    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”

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

エネルギー固有値 E_n 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 of 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”

ハミルトニアンを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
        for j=i-1
            if i <= j <= M-1
                T[i,j] = 1
            end
        end
        j = i + 1
        if i <= 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, w, xs)
    V = spzeros(Float64, M-1, M-1)
    for i=1:M-1
        x = xs[i]
        V[i,i] = (1/2)*w^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
```

```
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

PyTorchやTensorFlowと違い、
テンソルにする必要がない

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

エネルギーの定義

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

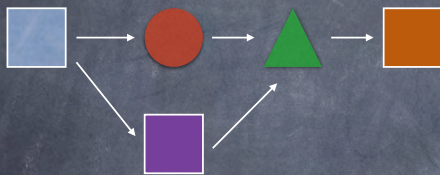
```
opt = Flux.setup(Adam(), model)
for i=1:50000
    Flux.train!(energy, model, (xs,), opt)
    e = energy(model, xs)
    if i % 1000 == 0
        println("i = $i energy = $e")
    end
end
```

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

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

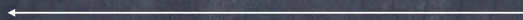
例 df/dx を計算したい

$$\begin{aligned}
 a(x) &= \sin(x) \\
 b(a) &= 0.2 + a \\
 c(b) &= \sqrt{1 + b} \\
 f(x) &= c(b(a(x)))
 \end{aligned}$$

連鎖律を使う

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

誤差逆伝播法と呼ばれている



自動微分とは？

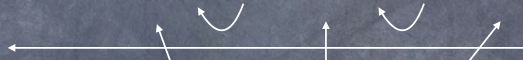
例 df/dx を計算したい

$$\begin{aligned}
 a(x) &= \sin(x) \\
 b(a) &= 0.2 + a \\
 c(b) &= \sqrt{1 + b} \\
 f(x) &= c(b(a(x)))
 \end{aligned}$$

連鎖律を使う

誤差逆伝播法と呼ばれている

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



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

Zygoterという
パッケージを
使えば

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

```
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

$B_y(\bar{y}, x) \equiv \frac{\partial y}{\partial x} \bar{y}$ 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

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

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

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

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

->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 α 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

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

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

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

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

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

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

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

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

$$\check{S}_{ki} = \sum_l \alpha_l (S_{ki+l} + S_{ki-l})^{48}$$

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

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

rruleにpullbackを返す関数
を実装すればよい

```
dHdtheta_a = gradient(theta -> re(theta)(S), theta)[1]
for i=1:length(theta)
    println("$i dHdtheta = ", dHdtheta_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、

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

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, Kashima, Chiba 277-0871, Japan

Akiyo 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がいない。どんな型でも自動微分できる



素粒子物理におけるJuliaの活用 ～格子QCDの大規模計算に向けて～

Akio Tomiya (IPUT Osaka, Assistant Prof.)
akio_at_yukawa.kyoto-u.ac.jp



MLPhyS Foundation of "Machine Learning Physics"
Grant-in-Aid for Transformative Research Areas (A)

Outline

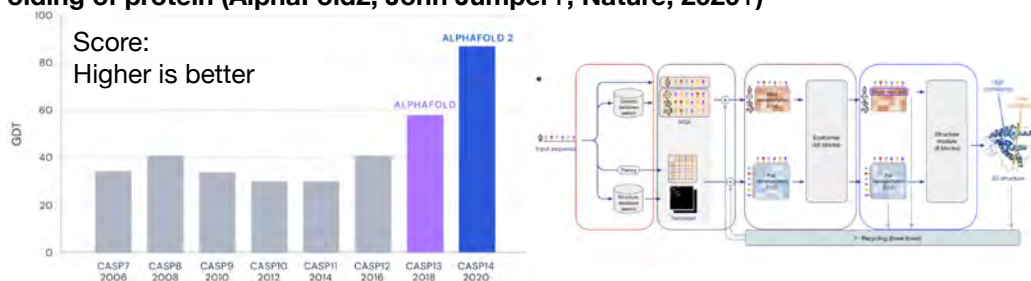
- 素粒子物理学? 格子QCD?
- Julia言語での取り組み
- 何ができれば良いのか

How to treat gauge fields with neural networks?

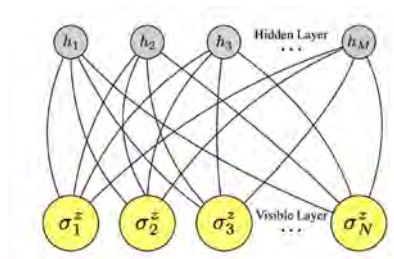
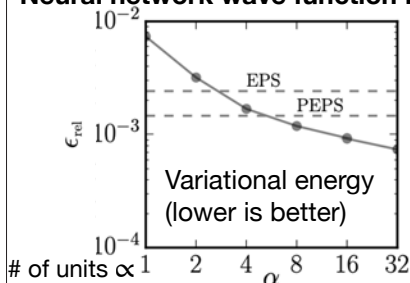
Applications of neural network in natural science Neural network have been good job

Akio Tomiya

Folding of protein (AlphaFold2, John Jumper+, Nature, 2020+)



Neural network wave function for many body (Carleo Troyer, Science 355, 602 (2017))



Neural net + Expert knowledge \rightarrow Best performance

What is neural networks?

Akio Tomiya

Affine transformation + element-wise transformation

Layers of neural nets $l = 2, 3, \dots, L, \vec{u}^{(l)} = \vec{x}$

$$\begin{cases} \vec{z}^{(l)} = W^{(l)}\vec{u}^{(l-1)} + \vec{b}^{(l)} & \text{Affine transf.} \\ & (\text{b=0 called linear transf.}) \\ u_i^{(l)} = \sigma^{(l)}(z_i^{(l)}) & \text{element-wise (local)} \end{cases}$$

A fully connected neural net

$$f_{\theta}(\vec{x}) = \sigma^{(3)}(W^{(3)}\sigma^{(2)}(W^{(2)}\vec{x} + \vec{b}^{(2)}) + \vec{b}^{(3)})$$

θ is a set of parameters: $w_{ij}^{(l)}, b_i^{(l)}, \dots$

Neural network = map between vectors and vectors

Physicists terminology: Variational transformations

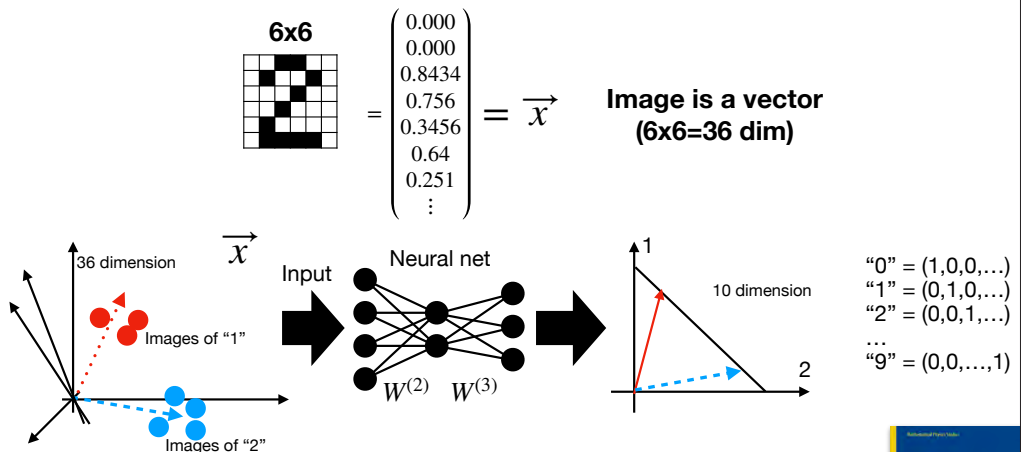
5

What is the neural networks?

Akio Tomiya

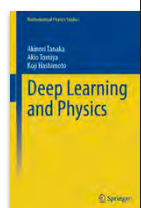
Neural network is a universal approximator

Example: Recognition of hand-written numbers



Fact: Neural network can mimic any function = A systematic variational function.

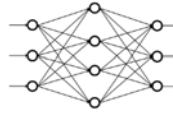
In this example, NN mimics image (36-dim vector) and label (10-dim vector)



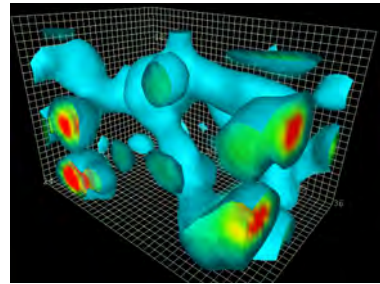
6

ML for LQCD is needed

- Neural networks
 - Data processing techniques mainly for 2d image (a picture = pixels = a set of real #)
 - Neural network helps data processing e.g. AlphaFold2
- Lattice QCD requires numerical effort but is more complicated than pictures
 - 4 dimension
 - **Non-abelian gauge d.o.f. and symmetry**
 - Fermions (Fermi-Dirac statistics)
 - Exactness of algorithm is necessary
- Q. How can we deal with neural nets?



thispersondoesnotexist.com



<http://www.physics.adelaide.edu.au/theory/staff/einweber/VisualQCD/QCDvacuum/>

7

Gauge covariant neural network = trainable smearing

Akio Tomiya

AT Y. Nagai arXiv: 2103.11965

Smearing = Gauge covariant way of transform gauge configurations

$$U_\mu(n) \rightarrow U_\mu^{\text{smr}}(n) = \mathcal{N} \left[(1 - \alpha)U_\mu(n) + \frac{\alpha}{6} V_\mu^\dagger[U](n) \right] \quad \text{Staple}$$

$$V_\mu^\dagger[U](n) = \sum_{\mu \neq \nu} U_\nu(n) U_\mu(n + \hat{\nu}) U_\nu^\dagger(n + \hat{\mu}) + \dots$$

$$\mathcal{N}[M] = \frac{M}{\sqrt{M^\dagger M}} \quad \text{Normalization or projection}$$

Gauge covariant neural network = General smearing with tunable parameters w

$$\begin{cases} z_\mu^{(l)}(n) = w_1^{(l)} U_\mu^{(l-1)}(n) + w_2^{(l)} \mathcal{G}_\theta^{(l)}[U] \\ \mathcal{N}(z_\mu^{(l)}(n)) \end{cases} \quad \text{Train (tune, fitting)}$$

point-wise (local)

$$\text{Gauge covariant NN: } U_\mu^{\text{NN}}(n)[U] = U_\mu^{(4)}(n) [U_\mu^{(3)}(n) [U_\mu^{(2)}(n) [U_\mu(n)]]]$$

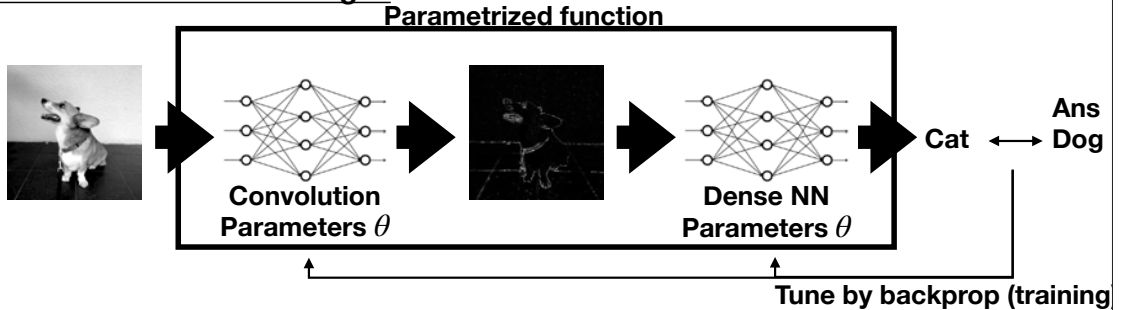
$$\text{Gauge covariant variational map: } U_\mu(n) \mapsto U_\mu^{\text{NN}}(n) = U_\mu^{\text{NN}}(n)[U]$$

Gauge covariant neural network

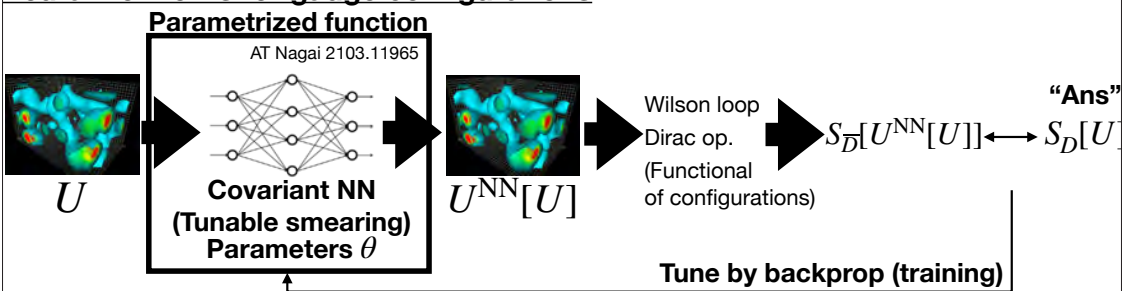
Schematic illustrations for neural networks (NN)

Akio Tomiya

Neural networks for images



Neural networks for gauge configurations



<http://www.physics.adelaide.edu.au/theory/staff/einweber/VisualQCD/QCDvacuum>

Alternative realization of gauge symmetric neural net: gauge equivariant neural net -> Tilo Wettig's talk

9

Gauge covariant neural network

= trainable smearing

Akio Tomiya

AT Y. Nagai arXiv: 2103.11965

Dictionary	(convolutional) Neural network	Gauge Covariant Neural network	
Input	Image (2d data, structured)	gauge config (4d data, structured)	
Output	Image (2d data, structured)	gauge config (4d data, structured)	
Symmetry	Translation	Translation, rotation(90°), Gauge sym.	
with Fixed param	Image filter	(APE/stout ...) Smearing	
Local operation	Summing up nearest neighbor with weights	Summing up staples with weights	
Activation function	Tanh, ReLU, sigmoid, ...	projection/normalization in Stout/HYP/HISQ	
Formula for chain rule	Backprop	“Smeared force calculations” (Stout)	Well-known
Training?	Backprop + Delta rule	AT Nagai 2103.11965	

(Index i in the neural net corresponds to n & μ in smearing. Information processing with NN is evolution of scalar field)

10

SLHMC = Exact algorithm with ML

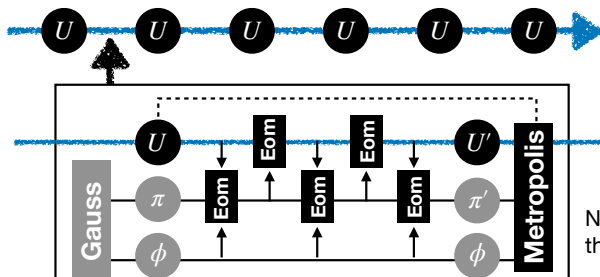
Akio Tomiya

SLHMC for gauge system with dynamical fermions

Gauge covariant neural network can mimics gauge invariant functions

-> It can be used in simulation? -> **Self learning HMC!**

HMC



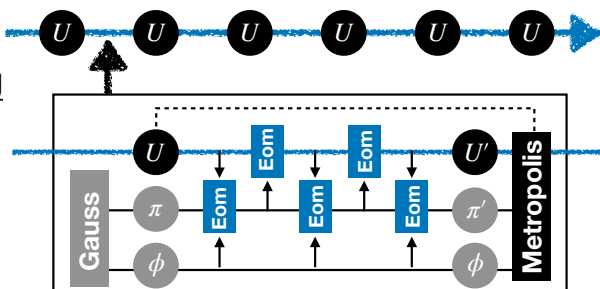
Eom Metropolis

Both use

$$H_{\text{HMC}} = \frac{1}{2} \sum \pi^2 + S_g + S_f$$

Non-conservation of H cancels since the molecular dynamics is reversible

Self Learning HMC



Metropolis

$$H = \frac{1}{2} \sum \pi^2 + S_g + S_f[U]$$

Eom

$$H = \frac{1}{2} \sum \pi^2 + S_g + S_f[U^{\text{NN}}[U]]$$

Neural net approximated fermion action but exact

arXiv: 2103.11965 and reference therein

SLHMC works as an adaptive reweighting!

Application for the staggered in 4d

Akio Tomiya

Problems to solve

arXiv: 2103.11965

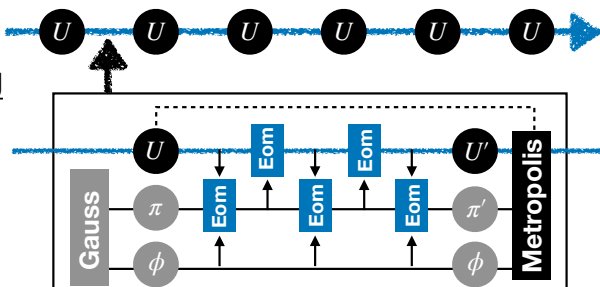
Mimic different action (Dirac operator):

(Final target: Domain-wall vs overlap)

A toy problem: Staggered (heavy) vs Staggered (light)

$$\left\{ \begin{array}{l} \text{Target action (Metropolis)} \\ \text{Action in MD} \end{array} \right. \begin{array}{l} S[U] = S_g[U] + S_f[\phi, U; m = 0.3], \\ S_\theta[U] = S_g[U] + S_f[\phi, U_\theta^{\text{NN}}[U]; m_h = 0.4], \end{array} \quad \leftarrow \text{mimic}$$

Self Learning HMC



Metropolis

$$H = \frac{1}{2} \sum \pi^2 + S_g + S_f[U]$$

Eom

$$H = \frac{1}{2} \sum \pi^2 + S_g + S_f[U^{\text{NN}}[U]]$$

Neural net approximated fermion action but exact

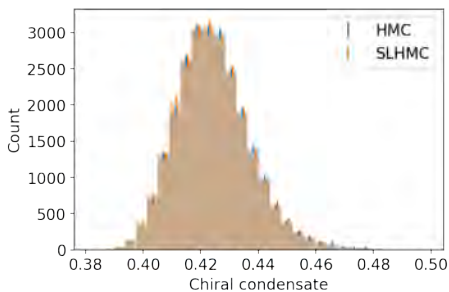
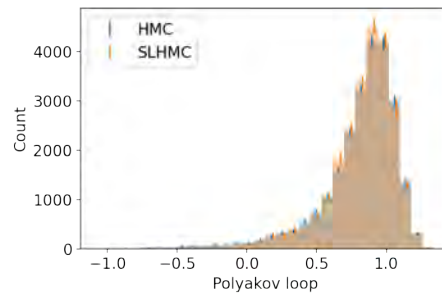
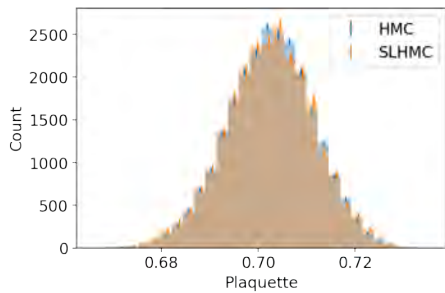
SLHMC works as an adaptive reweighting!

Application for the staggered in 4d

Akio Tomiya

Results are consistent with each other

arXiv: 2103.11965



Expectation value		
Algorithm	Observable	Value
HMC	Plaquette	0.7025(1)
SLHMC	Plaquette	0.7023(2)
HMC	Polyakov loop	0.82(1)
SLHMC	Polyakov loop	0.83(1)
HMC	Chiral condensate	0.4245(5)
SLHMC	Chiral condensate	0.4241(5)

Implemented by  LatticeQCD.jl 

13

Julia language on Fugaku, Lattice code in Julia

(LatticeQCD.jl and GomalizingFlow.jl)

Benchmark

Speed of Julia ~ Clang

Akio Tomiya



1. Open source scientific language (Just in time compiler/LLVM backend)
2. Fast as C/Fortran (faster sometimes), Practical as Python
3. Machine learning friendly

	Compiler	Benchmark (sec) single core	Type	Parallelism	GPU	Pros 🍌	Cons 🍌	Column, row	Note
Julia (1.8)	JIT, LLVM	0.0014	Dynamic & Static	MPI, others	CUDA	Fast Practical ML feiendly	not major	column-major	
C	Clang (LLVM)	0.0033	Static	MPI, others	CUDA	Fast	Long codes	row-major	
Python +Numba	(CPython) JIT, LLVM	0.0131	Dynamic	Available	Numba-CUDA	Practical ML feiendly	Not fully supported	row-major (Numpy)	(Rosetta2 is used in benchmark)

C and Julia have similar speed

Benchmarks are performed on m1 mac mini (similar tendency on Xeon)

Benchmark: Multiplications for 12dim vector and 12x12 complex matrix for 10^4 times (repeated 10 times)

15

Benchmark

Code comparison

Akio Tomiya

```
using Random

function main()
    T = 10
    K = 10^4
    N = 12
    #
    A = zeros(ComplexF64, (N,N))
    V = zeros(ComplexF64, N)
    W = zeros(ComplexF64, N)

    function myprod(A,V,W)
        for k = 1:N
            for i = 1:N
                W[i] += A[i, k]*V[k]
            end
        end
    end
end
...(cut)...
```



Attached in backup

```
#include <stdio.h>
#include <complex.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define T 10
#define K 10000
#define N 12

...(cut)...
void myprod(double complex A[N][N], double complex *V,
double complex *W) {
    for (int k = 0; k < N; k++) {
        for (int i = 0; i < N; i++) {
            W[i] += V[k] * A[k][i];
        }
    }
}
...(cut)...
```



Attached in backup

- Complex matrix (12x12) times complex vector (d=12)
 - One set= 10^4 times, and repeated 10 times and averaged
- Code of Julia looks like Python (short, simple) but fast as C
Julia: 0.0014 (sec), C: 0.0033 (sec). Single core performance is similar

Benchmark

Why Julia? (My personal opinion)

[1] https://akio-tomiya.github.io/julia_in_physics/

[2] <https://qr.ae/prgSG5>

- Modern scientific programming language
- **Easy to make codes. Fast as C/C++** (Julia& C use LLVM)
- Fewer compiling/dependency issues.
- Many people are potentially interested in. (More than 400 people registered to “Julia in physics 2022 online workshop” [1]). 4,923 public repo on Github
- No two Language problem. “The fact that while the users are programming in a high-level language such as R and Python, the performance-critical parts have to be rewritten in C/C++ for performance”. [2]

- **Neural network friendly (Flux.jl)**. Tensor networks also (iTensor.jl).

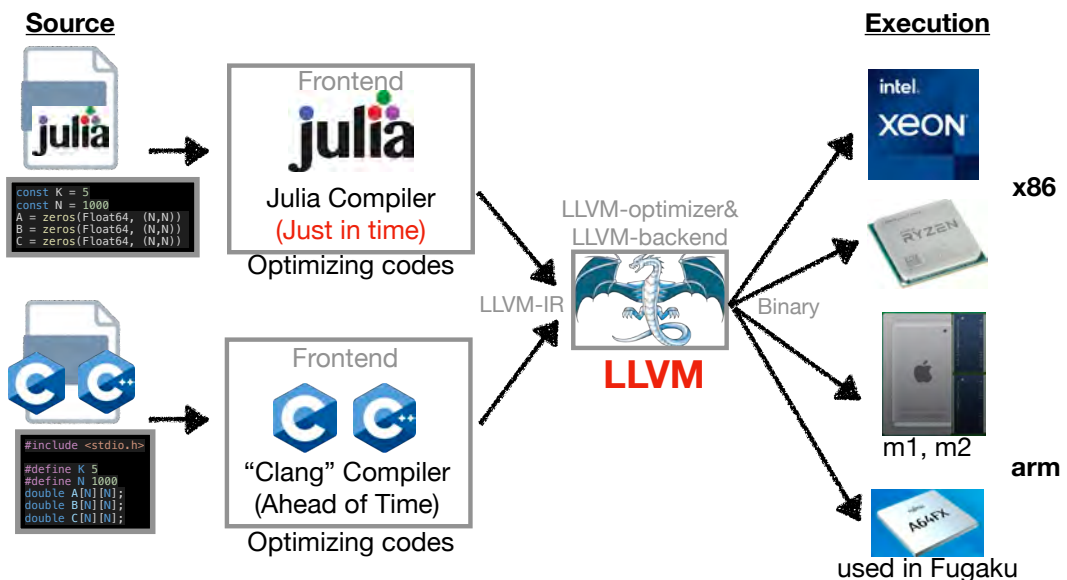
- Works on/with

- Xeon, Radeon/Apple silicon/A64FX
- MPI, GPU



LLVM?

LLVM = common backend for making binaries on multiple architectures



<https://www.fujitsu.com/jp/about/businesspolicy/tech/fugaku/>

https://ja.wikipedia.org/wiki/Apple_M1

<https://ja.wikipedia.org/wiki/Ryzen>

<https://ja.wikipedia.org/wiki/Xeon>

<https://gigazine.net/news/20200629-japan-fugaku-fastest-supercomputer/>

See: <https://en.wikipedia.org/wiki/LLVM> and related pages

Julia is ready on Fugaku(?)

Parallelization with A64FX/Fugaku

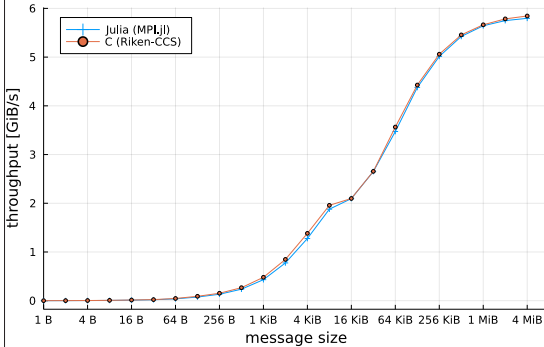
M. Giordano, arXiv:2207.12762v1 [cs.DC] 26 Jul 2022

Tests of MPI + on Fugaku

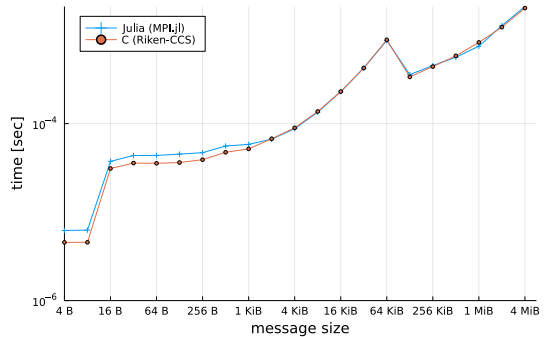



Send-Recv performance

Throughput of MPI PingPong @ Fugaku



Latency of MPI Allreduce @ Fugaku (384 nodes, 1536 ranks)



 has similar scaling of MPI with C
(no obvious overhead)

19

Lattice QCD code

Open source LQCD code in Julia Language

Akiyo Tomiya



AT & Y. Nagai in prep



Open source (Julia Official package, Now updated to **v1.0**)

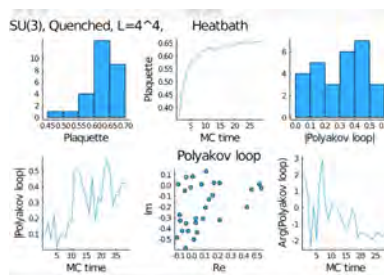
Machines: Laptop/desktop/**Jupyter/Supercomputers**

Functions: SU(Nc)-heatbath, (R)HMC, **Self-learning HMC**, SU(Nc) Stout
Dynamical Staggered, Dynamical Wilson, **Dynamical Domain-wall**
Measurements

Start LQCD
in 5 min

1. Download Julia binary
2. Add the package through Julia package manager
3. Execute!

<https://github.com/akio-tomiya/LatticeQCD.jl>



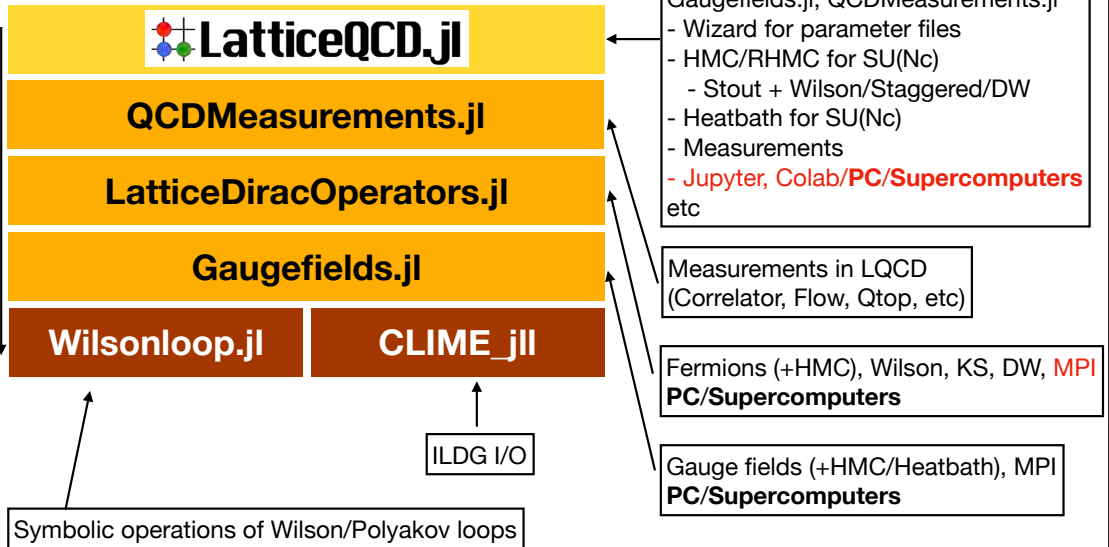
20

Package structure

Akio Tomiya

Our lattice QCD codes are constructed by following repositories

Dependency (Automatically solved)



See <https://github.com/akio-tomiya/LatticeQCD.jl> in detail

21

Benchmark of Julia + QCD

Wilson inversion / MPI parallel, Strong Scaling

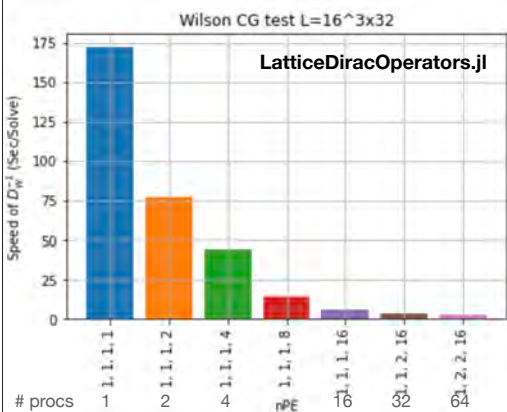
Akio Tomiya



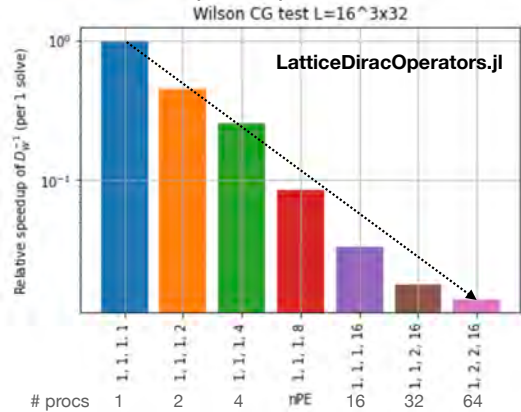
AT & Y. Nagai in prep

Tested on Yukawa-21@YITP

Absolute execution time



Relative speed up



It looks scaling well

We need more contributors!
Please help us

We thank to H. Ohno & Issaku Kanamori

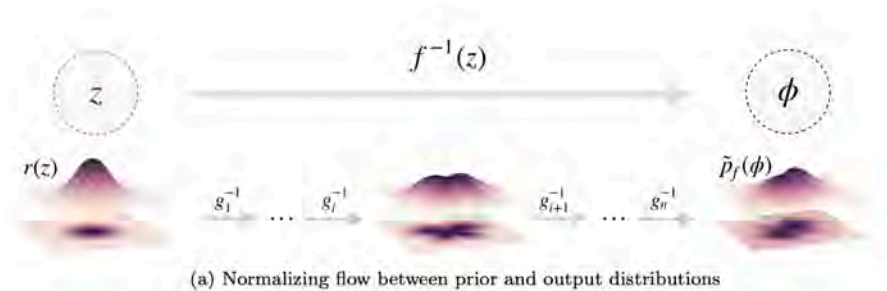
22

Flow based sampling algorithm

Akio Tomiya

Trivializing map realized using neural network

Normalizing flow? = Trivializing map, exact MCMC with ML



Change of variable by a neural network (Normalizing flow)

$$\phi = F^{(\text{NN})}[\varphi]$$

Sampling from Gaussian

- Inverse trivializing map (neural net)
- QFT configurations → Tractable Jacobian (by even-odd strategy)
- After sampling, Metropolis-Hastings test → exact!

arXiv:1904.12072, 2003.08413, 2008.05456

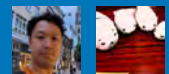
Related talk: Nobuyuki Matsumoto (Feb 16)

23

Normalizing flow in Julia

Akio Tomiya

A public code in Julia Language



arXiv:2208.08903v1 [hep-lat] 18 Aug 2022

GomalizingFlow.jl: A Julia package for Flow-based sampling algorithm for lattice field theory

Akio Tomiya

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

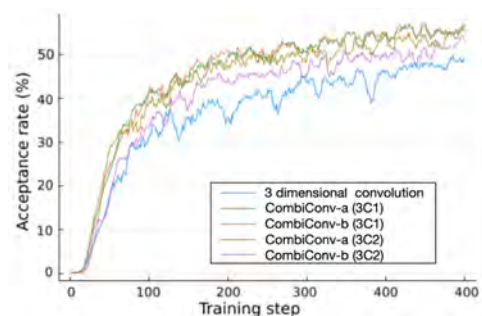
Satoshi Terasaki

AtelierArith, 980-0004, Miyagi, Japan

<https://arxiv.org/abs/2208.08903>

A public Julia code for the flow-based sampling algorithm for scalar field.
This supports not only 2d but also 3d.
CPU/GPU with Docker.

<https://github.com/AtelierArith/GomalizingFlow.jl>



A new type of convolution improves acceptance rate (~ shorten the autocorrelation)

I reported in NeurIPS 2022 workshop
<https://ml4physicalsciences.github.io/2022/>



24

Summary

Akio Tomiya

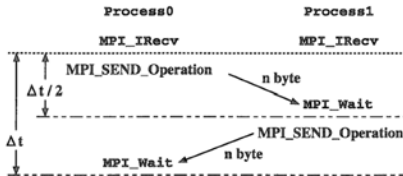
LQCD + Machine learning by Julia language

arXiv: 2103.11965
arXiv: 2208.08903

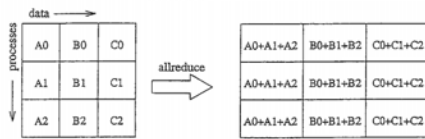
- Machine learning for LQCD
 - Neural net (NN) + expert knowledge -> Best performance
e.g. AlphaFold2, NN wave functions
 - NN can deal with 4d non-abelian gauge symmetric data now
 - **Self-learning HMC with NN works for dynamical fermions**
- Julia language for LQCD/HPC/ML
 - **Julia has similar speed with C (w/ & w/o MPI), and machine learning friendly**
 - Two Julia codes for lattice field theory
 - LatticeQCD.jl: A suite lattice QCD code, machine learning
 - GomalizingFlow.jl: Trivializing map via a neural network



PingPong

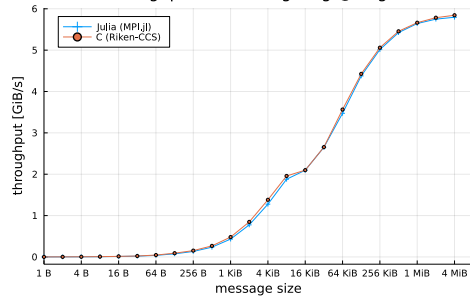


This section describes the performance of PingPong as a one-to-one communication. PingPong sends data between two ranks by Send communication from one rank, receives the data at the other rank, and then sends the data back to the original rank by Recv communication.

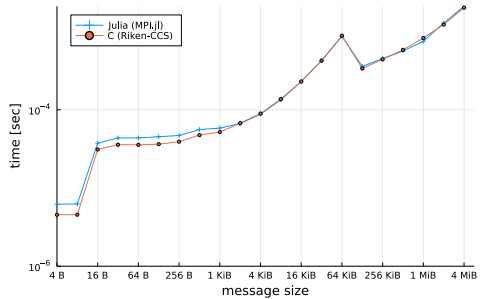


Allreduce collects data from all ranks, performs a set operation, and transfers the result to and transfers the result to all ranks.

M. Giordano, arXiv:2207.12762v1 [cs.DC] 26 Jul 2022
Throughput of MPI PingPong @ Fugaku



Latency of MPI Allreduce @ Fugaku (384 nodes, 1536 ranks)



<https://jopss.jaea.go.jp/pdfdata/JAERI-Data-Code-2001-010.pdf>

Introduction

Configuration generation with machine learning is developing

Year	Group	ML	Dim.	Theory	Gauge sym	Exact?	Fermion?	Reference
2017	AT, Akinori Tanaka	RBM + HMC	2d	Scalar	-	No	No	arXiv: 1712.03893
2018	K. Zhou+	GAN	2d	Scalar	-	No	No	arXiv: 1810.12879
2018	J. Pawłowski +	GAN + HMC	2d	Scalar	-	Yes?	No	arXiv: 1811.03533
2019	MIT+	Flow	2d	Scalar	-	Yes	No	arXiv: 1904.12072
2020	MIT+	Flow	2d	U(1)	Equivariant	Yes	No	arXiv: 2003.06413
2020	MIT+	Flow	2d	SU(N)	Equivariant	Yes	No	arXiv: 2008.05456
2020	AT, Akinori Tanaka +	SLMC	4d	SU(N)	Invariant	Yes	Partially	arXiv: 2010.11900
2021	M. Medvidovic+	A-NICE	2d	Scalar	-	No	No	arXiv: 2012.01442
2021	S. Foreman	L2HMC	2d	U(1)	Yes	Yes	No	
2021	AT+	SLHMC	4d	QCD	Covariant	Yes	YES!	This talk
2021	L. Del Debbio+	Flow	2d	Scalar, O(N)	-	Yes	No	
2021	MIT+	Flow	2d	Yukawa	-	Yes	Yes	
2021	S. Foreman, AT+	Flowed HMC	2d	U(1)	Equivariant	Yes	No but compatible	arXiv: 2112.01586
2021	XY Jing	Neural net	2d	U(1)	Equivariant	Yes	No	
2022	J. Finkenrath	Flow	2d	U(1)	Equivariant	Yes	Yes (diagonalization)	arxiv: 2201.02216
2022	MIT+	Flow	2d, 4d	U(1), QCD	Equivariant	Yes	Yes	arXiv:2202.11712 +
2022	AT+	Flow	2d, 3d	Scalar		Yrs		

+ ...

Benchmark

Code comparison

Akio Tomiya

```
using Random
function main()
T = 10
K = 10^4
N = 12
#
A = zeros(ComplexF64, (N,N))
V = zeros(ComplexF64, N)
W = zeros(ComplexF64, N)

function myprod(A,V,W)
for k = 1:N
for i = 1:N
W[i] += A[i, k]*V[k]
end
end
end

function test(A,V,W)
for jj=1:T
runtimes=[]
for r=1:K
A .= rand(N,N) + im*rand(N,N)
V .= rand(N) + im*rand(N)
W .= 0
tmp = @elapsed myprod(A,V,W)
push!(runtimes,tmp)
end
println("$jj=1) $(sum(runtimes)) #W[1] = $(W[1])")
end
end
test(A,V,W)
end

if abspath(PROGRAM_FILE) == @__FILE__
main()
end
```

```
#include <stdio.h>
#include <complex.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define T 10
#define K 10000
#define N 12

double urand(){
double m, a;
m = RAND_MAX + 1;
a = (rand() + 0.5)/m;
a = (rand() + a)/m;
return (rand() + a)/m;
}

void myprod(double complex A[N][N], double complex *V, double complex *W){
for (int k = 0; k < N; k++) {
for (int i = 0; i < N; i++) {
W[i] += V[k] * A[k][i];
}
}
}

void test(double complex A[N][N], double complex *V, double complex *W){
for (int jj = 0; jj < T; jj++) {
double runtimes = 0;
for (int r = 0; r < K; r++) {
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {
A[i][j] = urand() + urand() * I;
}
}
V[i] = urand() + urand() * I;
W[i] = 0.0 + 0.0 * I;
}
clock_t start = clock();
myprod(A, V, W);
clock_t end = clock();
runtimes += (double)(end - start) / CLOCKS_PER_SEC;
}
printf("%d %f # W[0] = %f %f\n", jj, runtimes, creal(W[0]), cimag(W[0]));
}
}

int main() {
double complex A[N][N];
double complex V[N];
double complex W[N];

test(A, V, W);

return 0;
}
```

29

「マス・フォア・インダストリ研究」シリーズ刊行にあたり

本シリーズは、平成 23 年 4 月に設立された九州大学マス・フォア・インダストリ研究所 (IMI) が、平成 25 年 4 月に共同利用・共同研究拠点「産業数学の先進的・基礎的共同研究拠点」として、文部科学大臣より認定を受けたことにもない刊行するものである。本シリーズでは、主として、マス・フォア・インダストリに関する研究集会の会議録、共同研究の成果報告等を出版する。各巻はマス・フォア・インダストリの最新の研究成果に加え、その新たな視点からのサーベイ及びレビューなども収録し、マス・フォア・インダストリの展開に資するものとする。

令和 4 年 10 月
マス・フォア・インダストリ研究所
所長 梶原 健司

数学と物理における Julia の活用

マス・フォア・インダストリ研究 No.27, IMI, 九州大学

ISSN 2188-286X

発行日 2024 年 1 月 18 日

編集 富谷昭夫, 横山俊一, 永井佑紀, 寺崎敏志

発行 九州大学マス・フォア・インダストリ研究所

〒819-0395 福岡市西区元岡 744

九州大学数理・IMI 事務室

TEL 092-802-4402 FAX 092-802-4405

URL <https://www.imi.kyushu-u.ac.jp/>

印刷 城島印刷株式会社

〒810-0012 福岡市中央区白金 2 丁目 9 番 6 号

TEL 092-531-7102 FAX 092-524-4411

シリーズ既刊

Issue	Author / Editor	Title	Published
マス・フォア・インダストリ 研究 No.1	穴田 啓晃 安田 貴徳 Xavier Dahan 櫻井 幸一	Functional Encryption as a Social Infrastructure and Its Realization by Elliptic Curves and Lattices	26 February 2015
マス・フォア・インダストリ 研究 No.2	滝口 孝志 藤原 宏志	Collaboration Between Theory and Practice in Inverse Problems	12 March 2015
マス・フォア・インダストリ 研究 No.3	笈 三郎	非線形数理モデルの諸相：連続，離散，超離散， その先 (Various aspects of nonlinear mathematical models (: continuous, discrete, ultra-discrete, and beyond)	24 March 2015
マス・フォア・インダストリ 研究 No.4	穴田 啓晃 安田 貴徳 櫻井 幸一 寺西 勇	Next-generation Cryptography for Privacy Protection and Decentralized Control and Mathematical Structures to Support Techniques	29 January 2016
マス・フォア・インダストリ 研究 No.5	藤原 宏志 滝口 孝志	Mathematical Backgrounds and Future Progress of Practical Inverse Problems	1 March 2016
マス・フォア・インダストリ 研究 No.6	松谷 茂樹 佐伯 修 中川 淳一 上坂 正晃 濱田 裕康	結晶のらせん転位の数理	10 January 2017
マス・フォア・インダストリ 研究 No.7	滝口 孝志 藤原 宏志	Collaboration among mathematics, engineering and industry on various problems in infrastructure and environment	1 March 2017
マス・フォア・インダストリ 研究 No.8	藤原 宏志 滝口 孝志	Practical inverse problems based on interdisciplinary and industry-academia collaboration	20 February 2018
マス・フォア・インダストリ 研究 No.9	阿部 拓郎 高島 克幸 縫田 光司 安田 雅哉	代数的手法による数理暗号解析 Workshop on analysis of mathematical cryptography via algebraic methods	1 March 2018
マス・フォア・インダストリ 研究 No.10	阿部 拓郎 落合 啓之 高島 克幸 縫田 光司 安田 雅哉	量子情報社会に向けた数理的アプローチ Mathematical approach for quantum information society	26 December 2018

Issue	Author / Editor	Title	Published
マス・フォア・インダストリ 研究 No.11	松谷 茂樹 佐伯 修 中川 淳一 濱田 裕康 上坂 正晃	結晶転位の先進数理解析 Advanced Mathematical Investigation for Dislocations	7 January 2019
マス・フォア・インダストリ 研究 No.12	滝口 孝志	Non-destructive inspection for concrete structures and related topics	13 February 2019
マス・フォア・インダストリ 研究 No.13	宇波 耕一 長野 智絵 吉岡 秀和 田上 大助 白井 朋之	数理農学における時系列データのモデル化と解析 Modeling and Analysis of Time Series Data in Math- Agro Sciences	28 February 2019
マス・フォア・インダストリ 研究 No.14	佐久間 弘文 大津 元一 小嶋 泉 福本 康秀 山本 昌宏 納谷 昌之	ドレスト光子に関する基礎的数理解析	18 March 2019
マス・フォア・インダストリ 研究 No.15	松谷 茂樹 佐伯 修 中川 淳一 濱田 裕康 富安 亮子	結晶の界面, 転位, 構造の先進数理解析	2 December 2019
マス・フォア・インダストリ 研究 No.16	Takuro Abe Yasuhiko Ikematsu Koji Nuida Yutaka Shikano Katsuyuki Takashima Masaya Yasuda	Quantum computation, post-quantum cryptography and quantum codes	17 January 2020
マス・フォア・インダストリ 研究 No.17	河村 彰星 津曲 紀宏 西澤 弘毅 溝口 佳寛	代数・論理・幾何と情報科学—理論から実世界への 展開	10 February 2020
マス・フォア・インダストリ 研究 No.18	Takashi Takiguchi	New technologies for non-destructive and non- invasive inspections and their applications	21 February 2020
マス・フォア・インダストリ 研究 No.19	Hirofumi Sakuma Motoichi Ohtsu Masayuki Naya Izumi Ojima Yasuhide Fukumoto	Basic mathematical studies on dressed photon phenomena	19 March 2020

マス・フォア・インダストリ 研究 No.20	松谷 茂樹 井上 和俊 加葉田雄太朗 佐伯 修 垂水 竜一 内藤 久資 中川 淳一 濱田 裕康	材料科学における幾何と代数 I	24 November 2020
マス・フォア・インダストリ 研究 No.21	Daisuke Sakurai Shigeo Takahashi Naoki Hamada Osamu Saeki Hamish Carr Takahiro Yamamoto	Fiber Topology Meets Applications	10 March 2021
マス・フォア・インダストリ 研究 No.22	巴山 竜来 中丸 啓 大垣 真二	数理的生成手法による CG とデジタルファブリケーション	15 March 2021
マス・フォア・インダストリ 研究 No.23	松谷 茂樹 井上 和俊 落合 啓之 佐伯 修 垂水 竜一 内藤 久資 中川 淳一 濱田 裕康 松江 要	材料科学における幾何と代数 II	11 November 2021
マス・フォア・インダストリ 研究 No.24	櫻井 大督 佐伯 修 高橋 成雄 Hamish Carr 山本 卓宏 濱田 直希	Fiber Topology Meets Applications 2	15 March 2022
マス・フォア・インダストリ 研究 No.25	佐久間弘文 大津 元一 小嶋 泉 福本 康秀	解析から設計に向けたオフシエル数理学	28 March 2022
マス・フォア・インダストリ 研究 No.26	松谷 茂樹 緒方 勇太 落合 啓之 加葉田雄太朗 佐伯 修 濱田 裕康 松江 要	材料科学における幾何と代数 IV	28 November 2023



Institute of Mathematics for Industry
Kyushu University

九州大学マス・フォア・インダストリ研究所

〒819-0395 福岡市西区元岡744
<https://www.imi.kyushu-u.ac.jp>