講義 1（80 分）

基本 + ベンチ（**BenchmarkTools / FLOPs**）

Hiroshi Shinaoka

導入

自己紹介：品岡 寛

- **所属**: 埼玉大学学術院・大学院理工学研究科
- **GitHub**: `https://github.com/shinaoka`
- **研究/開発**: 計算物理・量子多体・凝縮系（実装を伴う研究 OSS）
- **代表的 OSS（例）**:
  - `SparseIR.jl`（Julia） / `sparse-ir`（Python） / `sparse-ir-rs`（Rust）
  - `DCore`（DMFT software）
  - `TensorCrossInterpolation.jl`（Julia）
  - `CT-HYB` / `SpM` など
- **言語**: Julia, Python, Rust, C++/C, Fortran…
- **最近の活動**: `https://tensor4all.org`, Rust による計算物理エコシステム構築

自己紹介：品岡 寛

- **補足（一次情報）**：https://shinaoka.github.io

Julia の得意なこと

- **OSS**: 誰でも入手・配布・改善でき、教育でも環境を揃えやすい（対比：MATLAB は強力だが商用）
- **コミュニティの道具箱**: 微分方程式、可視化、最適化、自動微分…パッケージが豊富
- **パッケージシステム**: パッケージを簡単にインストール・配布可能
- **速いコードも書ける**: JIT + 型情報で最適化しやすい
- **ガーベージコレクション**: 自動でメモリ管理（`segmentation fault` が起きにくい）

Julia が苦手なこと

- **最高速追求の難しさ**: GC のため、細粒度の並列計算で遅くなりやすい
- **大規模開発の難しさ**: 動的型付けゆえ、設計・保守コスト増
- **他言語からの再利用の難しさ**: Python 以外は現状ハードル高め

私が思う Julia の現状

- 人間とのインターフェースとしての強み。ただ、生成 AI が発展し、人間とのインターフェースは段々不要になってきている。
- 大規模案件：他言語連携の検討（例：Rusty Julia）。Julia で全部エコシステムを作ろうという試みがあるが、個人的には必要ないと思う。

この講義の方針

- Julia の基本を学ぶ
- 正しくプロジェクトを管理する
- 最適化

最近の AI コーディングエージェントの発展は素晴らしい。 AI エージェントを操るのに十分な基礎知識を身につけることが目標。自分で今後成長していけるように。

Outline

型

型とはなにか？（Julia の「動的型づけ」）

- **変数に型**ではなく、**値に型**がつく
- 関数は「入力の型」に応じて最適化されうる
- 型は性能と表現力の両方に効く

型とはなにか？(C++との比較・イメージ)

- C++：`int x`（変数が型を持つ）
- Julia：`x = 1`（値 `1::Int` が型を持つ）
- Julia でも **局所変数**は `x::Float64` のように型を制約できる

julia

```julia
x = 1           # Int
x = 1.0         # Float64 に置き換わる（変数の型固定ではない）

y::Float64 = 1.0 # Float64 に型を制約できる（Python では不可）
y = 1.0 + 2.0im # InexactError（Float64 へ変換できない）

```

柔軟な記述と静的型付けのメリットの両立

抽象型と具体型

- **具体型（concrete type）**：実体（値）を持てる型（例：`Int64`, `Float64`, `Vector{Float64}`）
- **抽象型（abstract type）**：分類ラベル（値そのものは作れない）（例：`Number`, `Real`, `AbstractFloat`）
- Julia の「継承」＝ **部分型関係**（`<:`）
  - `Any` は全ての型のスーパータイプ（一番上）

julia

```julia
isabstracttype(Number)   # true
isconcretetype(Float64)  # true
typeof(1.0)              # Float64

```

代表的な型階層（例）

```text
Any
└─ Number
   └─ Real
      └─ AbstractFloat
         └─ Float64

```

ポイント：自作型で「分類」を作るときは `abstract type` を使う（`struct` は通常その下に置く）

構造体とデータ

構造体とはなにか？

- 名前のある「型」を自分で作る仕組み
- フィールド（属性）をまとめ、意味を持たせる
- 型がはっきりすると、コードも速く・安全に

構造体（最小例）

julia

```julia
struct Square
    side_length::Float64  # 辺の長さ
end

struct Circle
    radius::Float64       # 半径
end

Square(2.0).side_length  # 2.0
Circle(1.0).radius       # 1.0

```

ポイント：**データの塊に名前と型を与える**（設計が明確に、最適化もしやすく）。

Parametric type（パラメトリック型）

- `Vector{Float64}` の `{Float64}` のように、型にパラメータを持たせられる
- 目的：実装の重複を防ぎつつ、要素型などの情報を型に埋め込む → 速いコードになりやすい
- `SquareGeneric{Float64}` と `SquareGeneric{Int64}` は **別の具体型（別の型）**

julia

```julia
struct SquareGeneric{T<:Real}
    side_length::T
end

SquareGeneric(2.0)    # SquareGeneric{Float64}
SquareGeneric(2)      # SquareGeneric{Int64}

```

mutable / reference / GC

Immutable, Mutable

- Julia では型（type）自体に「mutable / immutable」の違いがある
- `struct` はデフォルトで **immutable（不変）な型**を定義（`p.x = ...` は不可）
- `mutable struct` は **mutable（可変）な型**を定義（フィールド更新が可能）
- `Array` は mutable（`a[i] = ...` ができる）

julia

```julia
ismutabletype(Int64)       # false
ismutabletype(Float64)     # false
ismutabletype(Vector{Int}) # true

```

Immutable / Mutable（最小例）

julia

```julia
struct Point
    x::Float64
    y::Float64
end

mutable struct MPoint
    x::Float64
    y::Float64
end

p = Point(1.0, 2.0)   # p.x = 3.0 はエラー
mp = MPoint(1.0, 2.0) # mp.x = 3.0 は OK

```

Reference（参照）と aliasing（共有）

julia

```julia
# mutable の例（配列：共有される）
a = [1, 2, 3]
b = a           # 参照のコピー（同じ配列を指す）
b[1] = 99
a               # [99, 2, 3] になる

c = a
c = [0]         # 変数 c の付け替え（a は変わらない）

# immutable の例（値：共有されない）
x = 1
y = x
y = 2
x               # 1

```

Stack / Heap / GC（直感）

- **stack（スタック）**：関数呼び出し中の「一時置き場」（関数が終わるとまとめて片付くイメージ）
- **heap（ヒープ）**：長く生きるデータの置き場（手動で片付けないと溜まるイメージ）
- Julia ではどこに置かれるか（stack/heap）は原則コンパイラが決める（ユーザが直接は制御しない）
- ざっくり：
  - 小さな immutable（isbits）は値として扱われやすい
  - mutable（Array など）は参照として扱われやすい → GC の対象になりやすい

Stack / Heap / GC（直感）

- **GC（garbage collection）**：参照されなくなったオブジェクトのメモリを自動回収（手動 `free` 不要）

関数・多重ディスパッチ

関数とはなにか？（メソッドの集合）

- Julia の「関数」は **メソッドの集合**
- 引数の型に応じて多重ディスパッチで呼び分け
- 「型 × 振る舞い」の設計が自然

多重ディスパッチ（人為的な最小例）

julia

```julia
f(x) = x                # 汎用（fallback）
f(x::Real) = x + 1      # 抽象型で定義して OK
f(x::Int) = x + 1       # より具体型（同じ意味のまま上書きもできる）

f(1)      # 2
f(1.0)    # 2.0 (Real)
f("a")    # "a" (fallback)

```

多重ディスパッチ（最小例）

julia

```julia
area(s::Square) = s.side_length^2
area(c::Circle) = π * c.radius^2

area(Square(2.0))  # 4.0
area(Circle(1.0))  # 3.1415...

```

JIT / 計測

Just-In-Time Compilation (JIT)

- 最初の呼び出しでコンパイル（遅い）
- コンパイル単位：**メソッド × 実引数の型**（specialization）
  - 型注釈の有無に関係なく、実際に渡された「具体型」で決まる
  - 引数型に `x::Real` のような抽象型を指定すること自体は可能（ただし実行時は具体型で specialize）
- 同じ「メソッド×型」なら、2 回目以降はキャッシュ済み（速い）
- TTFX: Time To First Execution（最初の呼出し時の待ち時間）

（補足）Julia 1.9 以降：事前コンパイルでネイティブコードも保存（pkgimage）→次回以降が速い
（注意）通常の実行中に JIT された分は基本セッション限り（永続化したいなら sysimage）

JIT と計測（BenchmarkTools の意味）

- “一回だけ実行” はコンパイル時間が混ざりがち
- `@btime` は繰り返して安定した時間を測る（＋`$` 補間）

（→ `projects/01_linalg_bench/`, `projects/02_broadcast_bench/` で体験）

配列

よく使う構造体の例: Array

julia

```julia
a = [1, 2, 3]
a[1] = 4
a # Array{Int64,1} / Vector{Int64}

A = [1 2; 3 4]   # 2x2 行列
A[1, 2]          # 2
A[2, 1] = 30
A # Array{Int64,2} / Matrix{Int64}

```

配列の添字は 1-based であることに注意。
配列の Element の型は、最初に決まったら変更できない。

配列の Element の型

高速な計算のために、配列の Element の型は Int64, Float64, Bool, Char などの基本的な型にするのが望ましい。

Array{Any}になる例

julia

```julia
v = [1.0, 1]       # Vector{Float64}（数値は昇格して揃う）
eltype(v)          # Float64

w = [1.0, "a"]     # Vector{Any}（揃えられないと Any）
eltype(w)          # Any

```

ハンズオン

ここからハンズオン（Project と計測）

以降は `projects/` を動かしながら進めます。

使う Project（通し番号）

- `projects/01_linalg_bench/`: 行列積ベンチ（BenchmarkTools、FLOPs）
- `projects/02_broadcast_bench/`: `sin.(x)`ベンチ（BenchmarkTools、broadcast）
- `projects/03_core_basics/`: 基礎文法・配列
- `projects/04_functions_dispatch/`: 関数・多重ディスパッチ

0. まず動かす（Project 環境）

例：`projects/01_linalg_bench/`

julia

```julia
using Pkg
Pkg.activate("projects/01_linalg_bench")
Pkg.instantiate()
Pkg.test()

```

Notebook 標準：各 Project の `notebook.ipynb` を開く。

1. 線形代数ベンチ：行列積（BenchmarkTools + FLOPs）

使う Project：`projects/01_linalg_bench/`

- `BenchmarkTools.@btime` の作法（`$` 補間）
- `A*B` の実測
- FLOPs/GFLOPs（目安 `2n^3`）の概念

2. broadcast の速度感：`sin.(x)`

使う Project：`projects/02_broadcast_bench/`

- `sin.(x)` を測る（要素/秒）
- dot/broadcast の直観

3. 基本：配列と制御構文（最低限）

使う Project：`projects/03_core_basics/`

- 配列、range、for/if
- view とコピーの違い（軽く）

4. 関数と多重ディスパッチ

使う Project：`projects/04_functions_dispatch/`

- 関数定義
- メソッド追加（`f(::Int)`など）

5. 予告：GC / allocation

allocation/GC の詳説は講義 3 で扱う（「なぜ allocation を減らすと効くか」もそこで回収）。