CHAPTER 1 Juliaをはじめよう

1-1 Juliaの特徴

この本を手に取っている方は、何らかの数値計算に興味があると思います。ここではまず、数値計算を行うためのプログラミング言語という観点でJuliaの特徴について述べようと思います。Juliaの特徴を箇条書きにすると以下の通りです。

1. Pythonのように書きやすい（学習コストが低い）
2. 数式をそのままの見た目に近い形でコーディングできる
3. 動的コンパイル言語（実行前にコンパイルが必要ない）
4. FortranやCのように高速
5. 多重ディスパッチ（コードの拡張がしやすい）
6. 充実したパッケージ管理システム

それぞれの点について簡単に述べます。

● 1-1-1 Pythonのように書きやすい（学習コストが低い）

Juliaは直感的な書き方が可能な文法となっています。特に、MatlabやPythonを使ったことがある方であれば、すんなりと勉強しやすいと思います。また、C言語を学び始めたときに言われるような「おまじない」的なものが少ないのも特徴です。例えば、

```
x = [1,2,3]
y = sum(x)
```

などとすれば、「配列を定義してその和を計算する」ことが可能です。文法がわかりやすいおかげで、コーディングに費やす時間を短くすることができ、自分のやりたい数値計算そのものに注力することが可能です。

● 1-1-2 数式をそのままの見た目に近い形でコーディングできる

先述した書きやすいこととも関連しますが、JuliaではUnicodeが使えるため、ギリシャ文字等を使うことができます。例えば、

```
f(α,θ) = exp(im*α*θ)
```

これは、関数 $f(\alpha,\theta)=e^{i\alpha\theta}$ を定義したことになります。さまざまな数値計算ではギリシャ文字が変数として使われるのはよくあるため、この特徴はコードの可読性をあげてくれます。

● 1-1-3 動的コンパイル言語（実行前にコンパイルが必要ない）

プログラミング言語には、書いたコードを機械が実行可能な形にする作業「コンパイル」を行った後に実行する言語（FortranやC言語など）と、実行時にコードをその場で解析しながら実行するインタプリタ言語（PythonやRubyなど）があります。

コンパイルという作業は人間が読めるコードを機械が読めるコードに翻訳する作業です。この作業を通じて使っているマシンに応じたコードの最適化が行われ、コードを高速に実行することができます。複数のコードと複数のライブラリからなるプログラムの場合、コンパイルによってそれらの依存関係が正しく考慮された一つの実行ファイルが生成されます。

逆にいえば、プログラムの開発者が存在を想定しているライブラリが自分のマシンにおいて正しく動かない場合、コンパイルに失敗し、実行できないことになります。どのライブラリをどのようにリンクするか、というのは初心者やそのプログラムに慣れていない人にとっては難しく、「コードはあるのに実行できない」という事態がたびたび生じます。Pythonなどのインタプリタ言語の場合、実行時にコンパイルという作業が必要なく、コードをそのまま走らせることができます。もちろん、複数のコードからなるパッケージもありますが、それぞれの個別のパッケージが動くことがわかっていれば、大抵の場合問題なく動かすことができます。しかし、コンパイルが必要な言語と比べて速度が出ない傾向にあります。

Juliaの場合、Pythonのようにコードをそのまま走らせることができ、明示的なコンパイルが必要ありません。つまり、Juliaは静的なコンパイル言語ではなく、動的なコンパイル言語です。しかし、コードが実行される過程で動的にコンパイルが行われ、最適化されたコードによって高速に動作します。これによって、実行前コンパイル作業という（初心者にとって）煩雑な工程を行うことなしにコード開発が可能とな

っており、気軽にコードの開発や機能追加を行うことができます。

● 1-1-4 FortranやCのように高速

数値計算を行うためのプログラミング言語を選択する際に重要となる要素はいくつか考えられますが、特に重視されるのは計算速度です。最先端の物理学のシミュレーションでは、スーパーコンピュータがよく用いられますが、計算速度が速いC言語やFortranが未だに現役として使われています。

Juliaは、これらのC言語やFortranに匹敵する速度を出すことができます。これは、動的コンパイルによって、計算機に最適なコードが生成されているためです。学びやすさと書きやすさを持ちながらも、高速に動作するということで、現代の数値計算では非常に有利な性質を持っています。

● 1-1-5 多重ディスパッチ(コードの拡張がしやすい)

Juliaの大きな特徴の一つは、多重ディスパッチです。これは、同じ名前の関数を複数定義することができ、どの定義を呼び出すかを引数の型に応じて動的に変えることができる、という機構です。例えば、「犬が n 回"ワン"と鳴く関数」と「蛙が n 回"ケロ"と鳴く関数」を、

```
001 struct 犬 end
002 struct 蛙 end
003 function 鳴く(C::犬,n)
004     println("ワン"^n)
005 end
006 function 鳴く(C::蛙,n)
007     println("ケロ"^n)
008 end
009 n=4
010 c = 犬()
011 鳴く(c,n)
012 d = 蛙()
013 鳴く(d,n)
```

のように両方「鳴く」で定義（JuliaはUnicodeが使えるので日本語の変数や関数を定義できます）すると、引数に「犬」が入るか「蛙」が入るかで挙動を変えることができます。これは多くのプログラミング言語で同様のことが可能ですが、Juliaの場合「犬と蛙が両方引数で入ったとき、 n 回"ゲゲゲワン"と鳴く」という関数も、

```
001 function 鳴く(A::犬,C::蛙,n)
002     println("ゲゲゲワン"^n)
003 end
004 鳴く(c,d,n)
```

のように定義することができます。この性質は既存のコードの機能を拡張したい場合にも有用です。

例えば、すでに「犬」に関するパッケージと「蛙」に関するパッケージがある場合に、犬と蛙の両方を同時に扱うパッケージを開発したいとします。このとき、二つのパッケージの中身を全くいじることなく『鳴く(A::犬,C::蛙,n)』を定義することで新しいパッケージを開発することができます。そして、犬に関する新しい関数『激しく鳴く(A::犬)』なども、犬のパッケージをいじることなく新しいパッケージで定義して利用することができます。この多重ディスパッチという機構によって、Juliaは書きやすい言語となっています。

● 1-1-6 充実したパッケージ管理システム

Juliaは2018年にバージョン1がリリースされた新しいプログラミング言語で、最新のプログラミング技術の知見が活かされています。その一つとして、Juliaはパッケージ管理システムを標準で備えている、という特徴があります。

「パッケージ」とは、ある機能が含まれたコード一式のことを意味します。あるパッケージが別の外部パッケージを必要とする（例えば、線形代数のパッケージを用いて固有値を計算することなど）ことはどのプログラミング言語でもよくあります。そして、あるパッケージの開発者は、開発したときに使った別の外部パッケージの特定バージョンでのみ動作確認をしていることもよくあります。Juliaの場合、そのパッケージが動作するために使う外部パッケージのバージョンの範囲を指定することができます。そして、Juliaは、パッケージのインストール時に動作確認したバージョンの外部パッケージを自動的にインストールしてくれます。これによって、利用者はバージョン違いに悩まされることなく安心してパッケージを導入することができます。

また、パッケージの作成は非常に気軽に行うことができ、自分で書いたコードをパッケージ化して再利用することもできます。

1-2 インストール方法

Juliaのインストール方法について、主要なOSごとに解説します。2024年5月現在、Juliaの最新バージョンは1.10です。他のバージョンでも基本的なインストール方法は変わらないと思います。

● 1-2-1 Juliaupを使用する方法

Juliaのインストール方法はいくつかの方法がありますが、2024年5月現在、初心者にもわかりやすいインストール方法はJuliaupを使用する方法かと思います。それぞれのOSごとに、インストール方法について解説します。

Windows OSの場合(Windows 11　Pro 22H2で確認)

JuliaはWindowsストアにあるため、ストアの検索で「Julia」と入れ、指示に従ってインストールすることができます。Windows PowerShellを使う場合には

```
winget install julia -s msstore
```

とすることでインストールすることができます。どちらのインストール方法もやっていることは同じで、最新版のJuliaをインストールし、環境変数も設定してくれます。これで、Windows PowerShellで

```
julia
```

と入れれば実行する準備が整います。

Mac OSやLinux OSの場合

端末（Macであればターミナル、Linuxであれば好きな端末）において、

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

を実行することでインストールができます。そのあとは.zshrcをリロードするか、新しい端末を開き直し

```
julia
```


と入れれば立ち上げることができます。

どのOSであれ、Juliaのバージョンアップをしたい場合には

```
juliaup update
```

とすることで最新版のJuliaにアップデートすることができます。

● 1-2-2 バイナリをダウンロードする方法

Juliaupを使わないでインストールすることもできます。例えば、クラスターやスーパーコンピュータなどにJuliaにインストールする場合には、下記に書かれてある方法を用いて自分のhomeディレクトリにJuliaを置いて使用することができます。しかし、クラスターやスーパーコンピュータにおいてもJuliaupを使う方法でインストールすることは基本的には可能なため、まずJuliaupを使った方法を試し、それができなかった場合には下記のインストール方法を試すといいでしょう。

どのOSでも共通なことは、公式サイト(https://julialang.org)に行き「Download」をクリックしてダウンロードページ(https://julialang.org/downloads/)に行くことです。検索で「Julialang」と入れることで公式ページを見つけられると思います。

Windows OSの場合(Windows 11　Pro 22H2で確認)

Windowsの場合は「Windows」の欄の「64-bit（installer）」をクリックし、「julia-1.10.2-win64.exe」をダウンロードしましょう。32ビット用のインストーラーもありますが、最近のWindows PCの場合は64ビットの方を選んでおけば間違いはありません。その後、ダウンロードしたインストーラーを開き、指示に従ってインストールします。基本的には「Next」のボタンを押していけばインストールは完了します。インストールが完了すると、プログラムの中に「Julia 1.10」が現れます。そちらをクリックして無事に開くことができればインストールは完了となります。

環境設定についてです。「設定」の「システム」「バージョン情報」のところの「システムの詳細設定」をクリックし「システムのプロパティ」画面を開きます。「詳細設定」タブの右下の「環境変数」をクリックし、「環境変数」画面を開きます。上の「ユーザーの環境変数」の「Path」をダブルクリックし「環境変数名の編集」画面を開きましょう。そして「新規」ボタンをクリックし、

「C:¥Users¥yuki¥AppData¥Local¥Programs¥Julia-1.10.2¥bin」と入れてください（ユーザー名yukiはご自分のユーザー名に変更してください。また、1.10.2の

部分は先ほどインストールしたバージョンに合わせてください)。あとは、「OK」を押してそれぞれのウィンドウを閉じてください。

Mac OSの場合(macOS 13.5で確認)

最近のMac(M1やM2などというApple製のCPUが使われているもの)の場合は、「macOS (Apple Silicon)」の欄の「64-bit (.dmg)」をクリックします。すると、「julia-1.10.2-macaarch64.dmg」というファイルがダウンロードされます(1.10の部分はインストールしたいバージョンに合わせて適宜読み替えてください)。Intel製のCPUを持つMacの場合には「macOS x86 (Intel or Rosetta)」の欄の「64-bit (.dmg)」をクリックすることで、「julia-1.10.2-mac64.dmg」というファイルがダウンロードされます。どちらにせよ、ダウンロードしたファイルを開き、中の「Julia-1.10」というアイコンを「アプリケーション」フォルダにドラッグアンドドロップしてください。これでインストールは完了です。Julia-1.10をダブルクリックして開ければ、インストールは成功しています。

環境設定についてです。Mac OS Catalina以降であれば、ターミナルを開き、ホームディレクトリにあるファイル`.zshrc`を開き、`export PATH=/Applications/Julia-1.10.app/Contents/Resources/julia/bin:$PATH`を追記します。Mac OS Catalina未満の場合には、`.bashrc`を開き同様に追記してください。

Linux OSの場合

Linuxの場合も自分のPCのCPUの種類に合わせたファイルをダウンロードページからダウンロードします。その後、解答するとbinディレクトリにJuliaのアプリケーションが入っているため、実行できればインストールは完了しています。なお、インストールする際にRoot権限は必要ないため、Homeディレクトリなどで解凍しておけばいいでしょう。Root権限が必要ないため、共用のクラスターマシンやスーパーコンピュータ上でも気軽にインストールしてJuliaを使用することができます。

環境設定についてです。解凍してできたbinディレクトリをPATHに追加します(bashであればMacと同様に.bashrcに記述します)。

● 1-2-3 実行方法

どのようにインストールしたとしても、上に書いたように作業すればJuliaは使えるようになります。このとき

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.2 (2024-03-01)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

のようなものが表示されていると思いますが、これはJuliaの「REPL」モードと呼ばれているものです。Pythonを「python」と立ち上げたときと似ており、そのまま計算をはじめることができます。一方、Juliaは他の方法でも実行することができます。例えば、端末(ターミナルあるいはWindows PowerShell等)において

```
julia test.jl
```

のような形で実行する場合、コードをあらかじめtest.jlに書き込み、それを走らせることができます。これもPythonで「python test.py」とするのと似ています。

この他の方法としては、Jupyter notebookを使う方法もあります。その方法は巻末の付録に書きました。この本ではREPLあるいは端末でのJuliaの実行を念頭に置くことにします。

1-3 あったら便利なツール

Juliaでコードを書く際に便利なツールを紹介します。コードを書く際にはどのエディタでも問題ありませんが、特にこだわりがなければ筆者はVisual Studio Code(VScode)をおすすめします。このエディタはMicrosoftが開発しているフリーのエディタで、軽量で高機能です。

CHAPTER 0 Julia的数値計算

0-1 自分の数値計算パッケージ　MyNumerics.jlの作成

さて、【第I部　基礎編】では、数値計算をするためのJuliaの基本的な機能について解説を行いました。【第II部　実践編】では、具体的な数値計算の問題をJuliaで解くことにします。

この実践編ではさまざまなアルゴリズムをfunctionとして実装しますが、それらをまとめて一つのパッケージを作成し、必要に応じて他のコードでも使える形にしましょう。

まず、パッケージを作るためのパッケージPkgTemplatesを導入します。REPLを立ち上げ、REPLで]キーを押してからadd PkgTemplatesとするか、

```
using Pkg
Pkg.add("PkgTemplates")
```

としてください。どちらの方法でもPkgTemplatesをインストールすることができます。

次に、パッケージの雛形を作ります。

```
using PkgTemplates
t = Template(;user="hogehoge",authors=["Seimei Abe"],dir="./",julia=v"1.9")
generate("MyNumerics.jl",t)
```

ここで、user名やauthorsは、今回一般公開するのではなく自分用ですので適当で構いません。これで、ディレクトリ直下./にMyNumericsというディレクトリが作成できました。

このディレクトリの中身は、


```
MyNumerics
├── LICENSE
├── Manifest.toml
├── Project.toml
├── README.md
├── src
│   └── MyNumerics.jl
└── test
    └── runtests.jl
```

のようになっているはずです。srcディレクトリの中にはMyPackage.jlというファイルがあると思います。このファイルの中身は

```
module MyNumerics
# Write your package code here.
end
```

となっているはずです。これでパッケージMyNumericsを作成する準備ができました。このファイルの中にコードを書いていけば、自分のパッケージMyNumericsの機能を増やすことができます。例えば、

```
module MyNumerics
function hellomypackage()
    println("こんにちは！")
end
end
```

のようにhellomypackage()を作成しておきましょう。 このMyNumericsを使う方法はいくつかありますが、ここでは一番シンプルな方法を紹介します。

まず、MyNumericsのディレクトリに移動し、JuliaのREPLを立ち上げます。その後、]キーを押して、activate .とすると、

```
(MyNumerics) pkg>
```

という状態になり、まっさらなJulia環境に自分のパッケージMyNumericsだけが入った環境を作ることができます。なお、activate .の.は自分の現在のディレクトリの場所を指定しているため、別の場所から呼ぶ場合にはその場所を指定してください。

この状態でdeleteキーを押して通常のjulia>の状態に戻すと、MyNumerics以外何も入っていないJuliaを使うことができます。この状態で、

```
using MyNumerics
MyNumerics.hellomypackage()
```

とすると、hellomypackage()を呼び出すことができて、こんにちは！が出力されます。

REPLではなく、端末からJuliaを実行したい場合には、

```
julia --project=./ test.jl
```

のように--projectでディレクトリの場所を指定することで、まっさらな状態のJuliaを呼び出すことができます。

このactivate .した状態には追加のパッケージが入っていませんので、別のパッケージを入れる場合にはいつも通りaddをしてください。

0-2 パッケージのテスト

作ったディレクトリの中にはsrcディレクトリがあり、その中にコードがありました。この他に、testというディレクトリの中にはruntests.jlがあります。このファイルの中身は、

```
using MyNumerics
using Test
@testset "MyNumerics.jl" begin
    # Write your tests here.
end
```

です。このファイルに書かれたコードは、REPLで]キーを押してパッケージモードにした後にactivate .し、testで実行することができます。そのため、試しに


```
using MyNumerics
using Test
@testset "MyNumerics.jl" begin
    MyNumerics.hellomypackage()
    a = 2*2
    @test a == 4
end
```

として、testを実行してみてください。

```
こんにちは！
Test Summary: | Pass  Total  Time
MyNumerics.jl |    1      1  0.0s
     Testing MyNumerics tests passed
```

以上のような出力が得られていると思います。このように、testによってパッケージの動作確認をできます。なお、@test a == 4というものをコードに書きましたが、これは、条件が満たされていればテストがPassになります。そのため、あらかじめ答えがわかっている場合には、ここに値を入れておくことで正しい動作になっているかを調べることができます。

以後、作成した関数はMyNumerics内に定義し、テストを適宜行うことにします。

0-3 ファイルの構成

次の章以降、具体的にfunctionを定義していきますが、すべての章の内容を一つのファイルにすると煩雑です。そのため、それぞれ以下のようなファイル名で保存し、includeを用いてMyNumericsに入れることとします。

具体的には、srcディレクトリに01.jlから07.jlという空のファイルを作成し、MyNumerics.jlは、

```
001 module MyNumerics
002 include("./01.jl")#連立方程式
003 include("./02.jl")#非線形方程式
004 include("./03.jl")#固有値
005 include("./04.jl")#数値微分
006 include("./05.jl")#補間と近似
007 include("./06.jl")#常微分方程式
008 include("./07.jl")#偏微分方程式
009 function hellomypackage()
010     println("こんにちは！")
011 end
012 end
```

とします。また、runtests.jlは、

```
001 using MyNumerics
002 using Test
003 @testset "連立方程式" begin
004     include("./01test.jl")
005 end
006 @testset "非線形方程式" begin
007     include("./02test.jl")
008 end
009 @testset "固有値" begin
010     include("./03test.jl")
011 end
012 @testset "数値微分" begin
013     include("./04test.jl")
014 end
015 @testset "補間と近似" begin
016     include("./05test.jl")
017 end
018 @testset "常微分方程式" begin
019     include("./06test.jl")
020 end
021 @testset "偏微分方程式" begin
022     include("./07test.jl")
023 end
```

とします。ここでtestディレクトリには01test.jlから07test.jlという空のファイルを作っておきます。ディレクトリ構造は以下の通りです。

```
├── LICENSE
├── Manifest.toml
├── Project.toml
├── README.md
├── src
│   ├── 01.jl
│   ├── 02.jl
│   ├── 03.jl
│   ├── 04.jl
│   ├── 05.jl
│   ├── 06.jl
│   ├── 07.jl
│   └── MyNumerics.jl
└── test
    ├── 01test.jl
    ├── 02test.jl
    ├── 03test.jl
    ├── 04test.jl
    ├── 05test.jl
    ├── 06test.jl
    ├── 07test.jl
    └── runtests.jl
```

ある章のコードを書いているときに別の章のテストを行いたくない場合には、適宜includeを#でコメントアウトしてください。

CHAPTER 1 連立一次方程式

まず、連立一次方程式を解いてみましょう。初めてのJuliaでの実装例になるので、比較的丁寧に理論とコードを追うことにします。 さて、

$$
\begin{aligned}
9x + 2y + z + 3t &= 20 \\
2x + 9y - 2z + t &= 16 \\
-x - 2y + 8z - 2t &= 8 \\
x - 2y - 3z + 4t &= 17
\end{aligned}
$$

という連立方程式を解く方法について考えましょう。未知数は x, y, z, t の4つで式が4本の連立方程式です。この方程式を、

$$
\begin{pmatrix} 9 & 2 & 1 & 3 \\ 2 & 9 & -2 & 1 \\ -1 & -2 & 8 & -2 \\ 1 & -2 & -3 & 4 \end{pmatrix}
\begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix}
=
\begin{pmatrix} 20 \\ 16 \\ 8 \\ 17 \end{pmatrix}
$$

という形に行列とベクトルを使って整理すると、

$$A\vec{x} = \vec{b}$$

という形の方程式になっていることがわかります。今回は、行列 A は 4×4 行列ですが、 N 本の方程式がある場合には $N \times N$ 行列となります。

1-1 標準機能による方法

Q 例題 1.1

Juliaの基本的な機能を用いて、上記の連立方程式を解き、解を表示せよ。

A 解答

第I部で紹介したように、連立方程式 $A\vec{x} = \vec{b}$ は、

```
x = A \ b
```

で解くことが可能です。そのため、コードは

```
function test01()
    A = [4 1 2 3
    2 9 -2 1
    -1 -2 8 -2
    1 -2 -3 4
    ]
    b = [20,16,8,17]
    x0 = A \ b
    println(x0)
end
test01()
```

となります。出力結果は

```
[-3.9999999999999996, 2.413793103448276, 3.3448275862068964, 8.96551724137931]
```

となります。今回はこの部分をMyNumericsパッケージのテストとして使います。つまり、01test.jlに上のコードを貼り付けます。そして、「0-2　パッケージのテスト」で述べたactivate .からtestを行います。今後行列Aを何度も使うことを考え、01test.jlを、

```
const A = [4 1 2 3
           2 9 -2 1
           -1 -2 8 -2
           1 -2 -3 4
           ]
const b = [20,16,8,17]
function test01()
    x0 = A \ b
    println(x0)
end
test01()
```

とします。ここで、Aやbは関数test01()の外で定義されているため、グローバル変数になっているように見えます。通常、グローバル変数は型が計算中に変更されうるために最適化が効きづらく遅いコードになってしまいますが、上のようにconstをつけるとこれは「定数」になるため、問題なく最適化が行われます。

1-2 反復法

ここでは、ヤコビ法とガウス・ザイデル法という二つの反復法を紹介します。

ヤコビ法

冒頭の連立方程式を以下のように変形してみます。

$$
\begin{aligned}
x &= -(2y+z+3t)/9+20/9\\
y &= -(2x-2z+t)/9+16/9\\
z &= -(-x-2y-2t)/8+8/8\\
t &= -(x-2y-3z)/4+17/4
\end{aligned}
$$

もし、解として x,y,z,t の値を持っていれば、右辺にその値を代入すれば左辺になるはずです。しかし、もし解ではない組 x,y,z,t を持っていた場合、等式は成り立ちません。このとき、右辺に値を代入した結果得られた左辺の値を新しい解の候補として採用し、また右辺に代入する、ということを繰り返すことを考えます。つまり、i 番目の候補 $x^{(i)},y^{(i)},z^{(i)},t^{(i)}$ を右辺に代入して、

$$
\begin{aligned}
x^{(i+1)} &= -(2y^{(i)}+z^{(i)}+3t^{(i)})/9+20/9\\
y^{(i+1)} &= -(2x^{(i)}-2z^{(i)}+t^{(i)})/9+16/9\\
z^{(i+1)} &= -(-x^{(i)}-2y^{(i)}-2t^{(i)})/8+8/8\\
t^{(i+1)} &= -(x^{(i)}-2y^{(i)}-3z^{(i)})/4+17/4
\end{aligned}
$$

のように $i+1$ 番目の解の候補を得ることとします。これを何度も繰り返すと、最後には左辺と右辺が等しい解が得られても良さそうです。この方法をヤコビ法と呼びます。

今回考えているのは変数が4つの4成分ベクトルですが、一般的には N 成分ベクトルを使って解くことになります。そこで、$\vec{x}=(x,y,z,t)^T$ の代わりに、$\vec{x}=(x_1,x_2,x_3,\cdots,x_N)^T$ という形で書いておきます。ベクトル $\vec{b}$ についても、$\vec{b}=(b_1,b_2,b_3,\cdots,b_N)^T$ としておきます。すると、上の式は

$$
x_j^{(i+1)} = \frac{1}{A_{jj}}\left[-\sum_{k=1,k\neq j}^{N} A_{jk}x_k^{(i)}+b_j\right] \tag{1}
$$

という形になります。

Q 例題 1.2

ヤコビ法を用いて連立方程式を解く関数を作成し、得られた解がたしかに解となっていることを確かめよ

A 解答

ヤコビ法で計算するコードは以下のようになります。

```
function hanpuku_Jacobi(A,b;eps = 1e-7,printx=true)
    n,m = size(A)
    @assert n==m "正方行列である必要があります。$n x $m 行列です。"
    x = zeros(Float64,n) #解の候補のベクトル。ここでは実数を仮定。
    x_temp = zero(x) #計算用の一時ベクトル。適宜更新される
    imax = 1000 #繰り返し代入の最大の数
    converged = false
    for i =1:imax
        err = 0.0 #誤差
        for j=1:n
            x_temp[j] = b[j] #まずbの値を足しておく
            for k=1:n
                if k==j
                    continue #対角の場合は計算しないのでスキップ
                end
                x_temp[j] -= A[j,k]*x[k]
            end
            x_temp[j] /= A[j,j] #対角要素で割る
            err += abs(x[j]-x_temp[j]) #j番目の要素の誤差を足す
        end

        if err < eps #誤差が設定したeps以下になると終了する
            converged = true
            break
        else
            x .= x_temp #解の候補を更新
        end
        if printx #表示したくない場合にはprintx=falseとしておく
            println("$i ",x)
        end
    end

    if  converged
        println("収束しました")
    else
        println("収束しませんでした")
    end
    return x
end
```


この関数を01.jlに書き込むことで、MyNumericsパッケージの関数の一つとなります。01.jlに上のコードを書く場合には、最後のendの後にexport hanpuku_Jacobiとすることで、MyNumerics.hanpuku_Jacobi(A,b)ではなくhanpuku_Jacobi(A,b)で呼び出せるようになります。これ以後、01.jlに追加する関数はすべてexport 関数名をendの後につけるようにしましょう。この関数をテストするには、01test.jlを、

```
using LinearAlgebra
function test01_Jacobi()
    x0 = A \ b
    println(x0)

    x = hanpuku_Jacobi(A,b)
    println("Jacobi: 残差ノルム：",norm(A*x-b))
end
test01_Jacobi()
```

のようにします。なお、ここで使った関数normはベクトルのノルムを計算する関数ですが、LinearAlgebraパッケージに入っている関数です。そのため、using LinearAlgebraを行なっています。そして、MyNumericsパッケージにはLinearAlgebraが入っていませんので、activate .の後にadd LinearAlgebraをする必要があります。このコードを実行すると、得られた解が真の解にどのくらい近いかがわかります。

i 番目のベクトル $\vec{x}^{(i)}$ を計算するためには、このベクトルの j 番目の要素 $\vec{x}_j^{(i)}$ を計算する必要がありますが、この要素は式(1)より変数 k に関する和が必要です。また、更新前のベクトル $\vec{x}^{(i)}$ のためにx_temp、更新後のベクトル $\vec{x}^{(i+1)}$ のためにxを用意しています。

例題 1.2 追加課題 1

1. 上記のヤコビ法のコードを、ループ最大数imaxを設定しなくても済むようにwhile文を用いて書き直せ。
2. 行列の要素が複素数の場合に拡張せよ。

理論的背景

ヤコビ法が収束する条件について調べておきましょう。行列 A を上三角行列 U と下三角行列 L と対角行列 D

$$U = \begin{pmatrix} 0 & 2 & 1 & 3 \\ 0 & 0 & -2 & 1 \\ 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & 0 \end{pmatrix},\ L = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ -1 & -2 & 0 & 0 \\ 1 & -2 & -3 & 0 \end{pmatrix},\ D = \begin{pmatrix} 9 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

を用いて、 $A = U + L + D$ とします。これらを用いると、反復する式は

$$D\vec{x}^{(i+1)} = -(U+L)\vec{x}^{(i)} + \vec{b}$$
$$\vec{x}^{(i+1)} = -D^{-1}(U+L)\vec{x}^{(i)} + D^{-1}\vec{b}$$

となります。真の解を $\vec{x}$ とすると、 i 番目の解の候補 $\vec{x}^{(i)}$ と真の解とのずれ（残差ベクトル）は、

$$\vec{e}^{(i)} \equiv \vec{x}^{(i)} - \vec{x} = -D^{-1}(U+L)(\vec{x}^{(i-1)} - \vec{x}) = -D^{-1}(U+L)\vec{e}^{(i-1)}$$

となります。最初に決めた初期の解候補 $\vec{x}_0$ の残差ベクトルを $\vec{e}_0 = \vec{x}_0 - \vec{x}$ とすると、

$$\vec{e}^{(i)} = M^i \vec{e}_0$$

と書くことができます。ここで $M \equiv -D^{-1}(U+L)$ です。したがって、何度も行列 M をかけていったときに $\vec{e}^{(i)}$ がどんどん小さくなっていけば、ヤコビ法は収束することがわかります。

もし M が対角化可能だった場合、 $M = P^{-1}\Lambda P$ のように固有値 λ_i が対角に並んだ対角行列 $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \cdots, \lambda_N)$ を用いて対角化でき、このとき、

$$M^i = P^{-1}\Lambda^i P$$

となります。つまり、行列 M のすべての固有値の絶対値が1以下であれば、ヤコビ法は収束します。そして、絶対値最大の固有値が0に近ければ近いほどすぐに収束することもわかります。

しかし、ある行列 A が与えられたときに、連立方程式を解くために行列 M の固有値を求めることは計算コストの意味で現実的ではありません。実際には、行列 A がすべての i について、

$$|A_{ii}| > \sum_{j, j \neq i} |A_{ij}|$$

を満たすとき（これを対角優位と呼びます）、ヤコビ法が収束することが言えます（証明は省略）。

例題 1.2 追加課題 2

上で定義された行列Aよりも早く収束するような行列Bを考え、試してみよ。

追加課題1.1略解

for i =1:imaxの代わりにwhile converged == falseとする。

追加課題1.2略解

xの定義のところでComplexF64を用いるか、if eltype(A) <: Complexのときにそう定義する。

追加課題2略解

行列 M は行列 A の対角要素が大きいほど値が小さくなっていくので固有値も小さくなる。そのため、対角要素が行列 A より大きい行列を考えればよい。

ガウス・ザイデル法

ヤコビ法を少し改良して、より少ない回数で収束する方法を考えてみましょう。式(1)は解の j 番目の要素を更新する式です。例えば、 $j=1$ の場合、 $x_1^{(i+1)}$ は、 $x_1^{(i)},\cdots,x_N^{(i)}$ を使って計算されます。この計算が終わった後、 $j=2$ の計算を行います。このとき、 $j=1$ と同じように $x_1^{(i)},\cdots,x_N^{(i)}$ を使って更新します。しかし、すでに $x_1^{(i+1)}$ が計算済みであることを考えれば、 $x_1^{(i)}$ の代わりに $x_1^{(i+1)}$ を使った方が良さそうな気がします。つまり、 $x_1^{(i+1)},x_2^{(i)},\cdots,x_N^{(i)}$ を使って $x_2^{(i+1)}$ を計算した方が良さそうです。同様に、 $x_1^{(i+1)}$ の計算が終わった後の $j=3$ の場合は、 $x_1^{(i+1)},x_2^{(i+1)}$ を使った方が良さそうです。つまり、式(1)の代わりに、

$$x_j^{(i+1)}=\frac{1}{A_{jj}}\left[-\sum_{k=1}^{j-1}A_{jk}x_k^{(i+1)}-\sum_{k=j+1}^{N}A_{jk}x_k^{(i)}+b_j\right] \tag{2}$$

とすると、より少ない回数で収束しそうです。この方法をガウス・ザイデル法と呼びます。

Q 例題 1.3

ガウス・ザイデル法を用いて連立方程式を解く関数を作成し、得られた解がたしかに解となっていることを確かめよ。

A 解答

実は、ヤコビ法のコードを少しだけ変えるだけでガウス・ザイデル法のコードになります。ガウス・ザイデル法の式はヤコビ法の式よりも一見複雑に見えますが、そうではありません。というのは、更新済みの $x_k^{(i+1)}$ （ $k<j$ ）を $k>j$ の $x_k^{(i+1)}$ の更新に使うだけだからです。つまり、ヤコビ法のコードの16行目を、

```
016 x_temp[j] -= A[j,k]*x_temp[k]
```

に置き換えればOKです。収束を止める条件を同じに設定したときに必要なループ回数が半分ほどになっていることを確認してみてください。

理論的背景

式(2)を

$$x_j^{(i+1)}=\frac{1}{A_{jj}}\left[-\sum_{k=1,k<j}^{N}A_{jk}x_k^{(i+1)}-\sum_{k=1,k>j}^{N}A_{jk}x_k^{(i)}+b_j\right]$$

のように書き換えると、括弧内の第一項目は行列 A のうち $k<j$ となる下三角行列、第二項目は $k>j$ となる上三角行列となっていることがわかります。すなわち、

$$x^{(i+1)}=D^{-1}(-Lx^{(i+1)}-Ux^{(i)})+D^{-1}b$$

となっていることがわかります。これを整理すると、

$$\begin{aligned}(D+L)\vec{x}^{(i+1)}&=-U\vec{x}^{(i)}+\vec{b}\\ \vec{x}^{(i+1)}&=-(D+L)^{-1}U\vec{x}^{(i)}+(D+L)^{-1}\vec{b}\end{aligned} \tag{3}$$

となります。そのため、ヤコビ法と同様に考えると、 $M=-(D+L)^{-1}U$ として、この行列の固有値の絶対値が1以下であれば収束することになります。また、ヤコビ法と同様に、ガウス・ザイデル法も「対角優位」な行列であれば収束することが知られています。

Q 例題 1.3 追加課題

例題の解答のコードを修正し、一時配列x_tempを使わずにxのみを使うコードに書き換えよ。

追加課題 解答

```
function hanpuku_GaussSeidel(A,b;eps = 1e-7,printx=true)
    n,m = size(A)
    @assert n==m "正方行列である必要があります。$n x $m 行列です。"
    x = zeros(Float64,n) #解の候補のベクトル。ここでは実数を仮定。
    imax = 1000 #繰り返し代入の最大の数
    converged = false
    for i =1:imax
        err = 0.0 #誤差
        for j=1:n
            xjold = x[j] #誤差の評価のため前の値を保持しておく
            x[j] = b[j] #まずbの値を足しておく
            for k=1:n
                if k==j
                    continue #対角の場合は計算しないのでスキップ
                end
                x[j] -= A[j,k]*x[k]
            end
            x[j] /= A[j,j] #対角要素で割る
            err += abs(xjold-x[j]) #j番目の要素の誤差を足す
        end
        if err < eps
            converged = true
            break
        end
        if printx
            println("$i ",x)
        end
    end
    if  converged
        println("収束しました")
    else
        println("収束しませんでした")
    end
    return x
end
```


このコードを使うと、長さ N の配列一つ分のメモリを節約することができます。この関数を01.jlに追加しておきましょう。

1-3 ガウスの消去法

上で述べたヤコビ法とガウス・ザイデル法は反復法と呼ばれ、計算を繰り返すことによって徐々に解に近づいていきます。ここでは、反復法とは異なる解法である「ガウスの消去法」について学ぶことにします。 ガウスの消去法は、中学数学で行う連立方程式を手で解くやり方と非常に似ています。つまり、係数を揃えて徐々に変数を消していく方法です。

冒頭に出てきた連立方程式について考えます。

1番目の式 $9x+2y+z+3t=20$ を定数倍して、2,3,4番目の式から x を消去することを考えます。具体的には、1番目の式を、2番目の式に対しては $2/9$ 倍、3番目の式に対しては $-1/9$ 倍、4番目の式に対しては $1/9$ 倍してそれぞれの式から引きます。つまり、

$$
\begin{aligned}
9x+2y+z+3t&=20\\
\left(9-2\frac{2}{9}\right)y+\left(-2-\frac{2}{9}\right)z+\left(1-3\frac{2}{9}\right)t&=16-20\frac{2}{9}\\
\left(-2+2\frac{1}{9}\right)y+\left(8+\frac{1}{9}\right)z+\left(-2+3\frac{1}{9}\right)t&=8+20\frac{1}{9}\\
\left(-2-2\frac{1}{9}\right)y+\left(-3-\frac{1}{9}\right)z+\left(4-3\frac{1}{9}\right)t&=17-20\frac{1}{9}
\end{aligned}
$$

です。2番目から4番目の式には x が無くなっているため、3本の式だけに着目すると、 y,z,t という3つの変数に対して式が3本という連立方程式となっていることがわかります。この3本からなる連立方程式に対しても同様のことを行うことができるため、2番目の式を定数倍して、3,4番目の式から y を消去することができます。これを繰り返すと、

$$
\begin{aligned}
a_{11}x+a_{12}y+a_{13}z+a_{14}t&=b_1\\
a_{22}y+a_{23}z+a_{24}t&=\tilde{b}_2\\
a_{33}z+a_{34}t&=\tilde{b}_3\\
a_{44}t&=\tilde{b}_4
\end{aligned}
$$

というような形になります。ここで a_{ij} と $\tilde{b}$ は上の操作を繰り返して得られた係数です。4番目の式は変数が t だけの方程式なので、 $t = \tilde{b}_4/a_{44}$ と直ちに t が求まります。

次に、この t を3番目の式に代入すれば $z = (\tilde{b}_3 - a_{34}t)/a_{33}$ と z が求まります。これを繰り返せば、 y も x も求まるため、連立方程式の解が得られたことになります。これがガウスの消去法です。

このガウスの消去法を N 本の連立方程式に適用することを考えます。まず、 i 番目の変数 x_i を消去するためには、 i 番目の方程式の定数倍を差し引きする必要があります。この i 番目の方程式は、

$$A_{i1}x_1 + \cdots + A_{ii}x_i + \cdots + A_{iN}x_N = \tilde{b}_i$$

という形をしています。このとき、変数 x_i を消去したい k 番目の方程式は、

$$A_{k1}x_1 + \cdots + A_{ki}x_i + \cdots + A_{kN}x_N = \tilde{b}_k$$

となっています。変数 x_i をこの k 番目の方程式から消去したい場合には、 A_{ki}/A_{ii} を i 番目の方程式にかけて、引けばよいことがわかります。つまり、 k 番目の方程式は、

$$\sum_{j=1,j\neq i}\left[A_{kj} - A_{ij}\frac{A_{ki}}{A_{ii}}\right]x_j = \tilde{b}_k - \tilde{b}_i\frac{A_{ki}}{A_{ii}}$$

という形になります。

これを行列の形で書くと、行列 A の k,j 成分 A_{kj} は k 番目の方程式の j 番目の変数の係数ですが、これが、

$$A_{kj} \to A_{kj} - \frac{A_{ki}}{A_{ii}}A_{ij}$$

と置き換わり、 $\tilde{b}$ が、

$$\tilde{b}_k \to \tilde{b}_k - \frac{A_{ki}}{A_{ii}}\tilde{b}_i$$

と置き換わったことになります。ガウスの消去法の全体をコーディングする前に、この i 番目の変数を消去するコードを書いてみましょう。

Q 例題 1.4

NxN行列Aとベクトルbを入力として、i番目の変数を消去して得られる行列A'と修正されたベクトルb'を出力するコードを作成し、得られた行列A'のi行以外の行のi列目が0になっていることを確認せよ。引数はAとbとiとせよ。

A 解答

数式そのままに実装してみると、以下のような関数となります。

```
001 function syokyo(A,b,i)
002     n,m = size(A)
003     @assert n==m "正方行列である必要があります。$n x $m 行列です。"
004     Ap = zeros(Float64,n,n)
005     bp = zeros(Float64,n)
006
007     for k=1:n
008         a = A[k,i]/A[i,i]
009         for j=1:n
010             if k==i #i番目の方程式からは引かない
011                 Ap[k,j] = A[k,j]
012             else
013                 Ap[k,j] = A[k,j] - A[i,j]*a
014             end
015         end
016         bp[k] = b[k] - b[i]*a
017     end
018     return Ap,bp
019 end
```

これを01.jlに追記してください。また、export syokyoを最後につけておきます。そして、01test.jlには新しい関数test01_syokyo():

```
function test01_syokyo()
    i = 1
    Ap,bp = syokyo(A,b,i)
    display(Ap)
    println("\t")

    i = 2
    Ap,bp = syokyo(A,b,i)
    display(Ap)
    println("\t")
end
test01_syokyo()
```

をつけておきます（以前のtest01()はコメントアウトしても構いません）。この出力結果は、

```
4×4 Matrix{Float64}:
 4.0   1.0    2.0   3.0
 0.0   8.5   -3.0  -0.5
 0.0  -1.75   8.5  -1.25
 0.0  -2.25  -3.5   3.25
4×4 Matrix{Float64}:
  3.77778   0.0   2.22222   2.88889
  2.0       9.0  -2.0       1.0
 -0.555556  0.0   7.55556  -1.77778
  1.44444   0.0  -3.44444   4.22222
```

となります。実はこのコードはもう少し高速化することができます。

4×4 行列のような小さな場合にはほとんど変化はありませんが、1000×1000 行列のように大きくなってくると顕著に速度差が現れます。

Q 例題 1.4 追加課題

例題の解答のコードを修正し、より高速なコードに変更せよ。また、速度差を比較せよ。

A 解答

高速なコードの名前をsyokyo_fastとします。速度差を比較するコードは

```
function test01_syokyo_fast()
    N = 1000
    A = rand(N,N)
    b = rand(N)

    i = 1
    Ap1,bp1 = syokyo(A,b,i)
    Ap2,bp2 = syokyo_fast(A,b,i)
    println(sum(Ap1 .- Ap2))

    i = 2
    @time Ap1,bp1 = syokyo(A,b,i)
    @time Ap2,bp2 = syokyo_fast(A,b,i)
    println(sum(Ap1 .- Ap2))
    i = 3
    @time Ap1,bp1 = syokyo(A,b,i)
    @time Ap2,bp2 = syokyo_fast(A,b,i)
    println(sum(Ap1 .- Ap2))
end
test01_syokyo_fast()
```

とします。ここでは 1000×1000 の行列を作成し、その関数の計算時間を@timeというマクロで測定しています。この関数はtest01.jlに追加しておきます。test01_syokyo_fastでは、println(sum(Ap1 .- Ap2))で、元の関数と今作った関数の出力が同じになっているかを比較しています。syokyo_fastは

```
function syokyo_fast(A,b,i)
    n,m = size(A)
    @assert n==m "正方行列である必要があります。$n x $m 行列です。"
    Ap = zeros(Float64,n,n)
    bp = zeros(Float64,n)
    Ap .= A
    bp .= b

    @inbounds for k=1:n
        a = A[k,i]/A[i,i]
        bp[k] -= b[i]*a
        if k != i
            for j=1:n
                Ap[k,j] -= A[i,j]*a
            end
        end
    end
    return Ap,bp
end
```

として、01.jlに追加しておきます。ここで、@inboundsというマクロを使っていますが、これは配列を扱う際にfor文の前につけておくと、速度が上がります。通常、Juliaでは、配列の中身を取り出す際A[i]などとしますが、引数iがちゃんと配列のインデックスの最小と最大の中に収まっているかをチェックしています。しかし、@inboundsをつけると、そのチェックをしなくなります。チェックをしないのでその分早くなります。通常は、ちゃんと計算が回っているかを確認した後、@inboundsをつけることが多いです。test01_syokyo_fastを実行した結果、出力は

```
0.0
  0.019362 seconds (3 allocations: 7.637 MiB, 35.47% gc time)
  0.009733 seconds (3 allocations: 7.637 MiB)
0.0
  0.012740 seconds (3 allocations: 7.637 MiB)
  0.010162 seconds (3 allocations: 7.637 MiB)
0.0
```


となります。劇的な高速化を達成しているわけではありませんが、速くなっているのはわかると思います。なお、行列のサイズが大きくなるほど差が開いていきます。

さて、ガウスの消去法を実行するためには、すべての変数を順番に消去していく必要があります。つまり、N 行の連立方程式であれば $N-1$ 回の消去作業が必要です。すべての変数を消去すると、4成分の連立方程式の時と同様に行列は上三角行列になります。この件を踏まえて、以下のコードを作成してみましょう。

Q 例題 1.5

$N \times N$ 行列Aとベクトルbを入力として、変数を消去することにより行列Aを上三角行列に変形せよ。また、bも同様に変形せよ。入力はAとbとし、Aとbは計算後に変形されたものに変わっているものとする。

A 解答

例題4.1のiを1から順番に行なっていけばいいため、

```
function gauss!(A,b)
    n,m = size(A)
    @assert n==m "正方行列である必要があります。$n x $m 行列です。"
    for i=1:n #i番目の方程式
        for k=(i+1):n #iよりも下にある方程式。i番目の方程式を使って消す
            a = A[k,i]/A[i,i] #k番目の方程式のi個目の変数を消すため。
            for j=i:n
                A[k,j] -=  A[i,j]*a #j=iの時、係数を消してくれている
            end
            b[k] -=  b[i]*a
        end
    end
end
```

となります。ここで、順番に消去する場合は i 番目の次の方程式の係数を消せばいいということで、k=(i+1):nとなっています。この関数を01.jlに記入し、01test.jlには

```
001 function test01_gauss()
002     n,m = size(A)
003     At = zeros(n,m)
004     At .= A
005     bt = zeros(m)
006     bt .= b
007     gauss!(At,bt)
008     display(At)
009     println("\t")
010     display(bt)
011 end
012 test01_gauss()
```

を定義しておきます。ここで、At = zeros(n,m)とAt .= Aのように新しい行列Atを定義している理由ですが、gauss!(At,bt)の中ではAtやbtの中身を変更していますが、Aはもともと整数の行列と定義していたため、そこに実数を入れようとするとエラーが出るからです。At = zeros(n,m)では、Atを倍精度実数の行列と定義しているため、gauss!(At,bt)を実行しても問題ありません。

testを実行すると、AとbはそれぞれAt

```
4×4 Matrix{Float64}:
 4.0  1.0   2.0       3.0
 0.0  8.5  -3.0      -0.5
 0.0  0.0   7.88235  -1.35294
 0.0  0.0   0.0       2.3806
4-element Vector{Float64}:
 20.0
  6.0
 14.235294117647058
 21.34328358208955
```

となります。ちゃんと上三角行列になっていますね。

ここで、関数の名前がgauss!となっており、エクスクラメーションマーク!がついていることに注意してください。この!は付けなくても構いません。Juliaでは、入力した引数が関数の内部で変更される場合、慣例として!をつけることにしています。Juliaの標準的な関数を見ると、どれも基本的にはこのルールに従っていて、引数として入れた配列が変更されたかどうかがわかります。今回は、AとbはJuliaの関数gauss!を

呼び出すことで変更されています。

上三角行列になってしまえば、あとは一番下の方程式から順番に答えを求めていけばいいです。得られた上三角行列を $\tilde{A}$ 、変形された右辺ベクトルを $\tilde{b}$ とすると、N 番目の方程式から、

$$x_N = \tilde{b}_N / \tilde{A}_{NN}$$

と直ちに求まります。$N-1$ 番目は2項しかないため、

$$\tilde{A}_{N-1N-1} x_{N-1} + \tilde{A}_{N-1N} x_N = \tilde{b}_{N-1}$$

より、

$$x_{N-1} = (\tilde{b}_{N-1} - \tilde{A}_{N-1N} x_N) / \tilde{A}_{N-1N-1}$$

となりますね。あとはこれを繰り返していけばいいので、解ベクトルの i 成分は、

$$x_i = \left(\tilde{b}_i - \sum_{j=i+1}^{N} \tilde{A}_{ij} x_j \right) / \tilde{A}_{ii}$$

となります。これをコーディングしましょう。

Q 例題 1.6

例題 1.5のコードを改良し、解ベクトルが出力されるように変更せよ。