Julia による数値解析の基礎のキソ

あるいは

数値解析による Julia の基礎のキソ

(β 版)

川平　友規

一橋大学

はじめに

この講義ノートは 2024 年度に一橋大学で行われた『数値解析』の講義（および演習）で配布した講義ノート（プリント）を書籍風にまとめたものです．演習用のプログラム言語としては Julia を用いました．Julia と数値解析，そのいずれにも馴染みがないみなさんを主な対象として想定したため，Julia の教科書としても，数値解析（数値計算法）の教科書としても物足りないものではありますが，その点は「基礎のキソ」ということでご容赦ください．

とくに，Julia のプログラム（コード）の例については，以下のような方針で作成しています．

- 泥臭いコードであっても数学的な自然さと読みやすさを優先．
- Julia 特有の表現や機能はできるだけ避ける．とくに，グラフ描画に必要な Plots 以外のパッケージは利用しない．
- 一方で，「変数の型宣言をしなくても適宜解釈してくれる」という Julia の利点は最大限活用する．

つまり，これは Julia と数値解析両方のクラッシュコースということになります．

Julia のプログラミング環境について

現在この講義ノート（β 版）は，一橋大学での講義，ゼミ，演習等で利用する目的で作成されているため，一橋大学学内の情報端末に構築されているプログラミング環境のみを想定して書かれています．具体的には，

(1) プログラミング言語 Julia の最新バージョン
`https://julialang.org/`
(2) Julia のパッケージ Plots の最新バージョン
`https://docs.juliaplots.org/stable/`
(3) Visual Studio Code（VSCode, エディター）
`https://code.visualstudio.com/`
(4) VSCode の Julia 拡張
`https://code.visualstudio.com/docs/languages/julia`

の 4 つがインストールされた環境です．第 1 講や第 2 講あたりまではとくに，そのような環境を意識した記述がなされていますが，実際にはお使いのパソコンに上の 4 つのう

ち (1) と (2) さえインストールされていれば，お好みのエディタやターミナルを用いて本講義ノートにあるコードや練習問題に取り組むことができます．インストール方法については，インターネットから十分な情報が得られると思います．

よく使う記号など：数の集合

(1) $\mathbb{C}$: 複素数全体　(2) $\mathbb{R}$: 実数全体　(3) $\mathbb{Q}$: 有理数全体
(4) $\mathbb{Z}$: 整数全体　(5) $\mathbb{N}$: 自然数全体　(6) $\emptyset$: 空集合

ギリシャ文字

(1) α: アルファ　(2) β: ベータ　(3) γ, Γ: ガンマ　(4) δ, Δ: デルタ
(5) ϵ: イプシロン　(6) ζ: ゼータ　(7) η: エータ　(8) θ, Θ: シータ
(9) ι: イオタ　(10) κ: カッパ　(11) λ, Λ: ラムダ　(12) μ: ミュー
(13) ν: ニュー　(14) ξ, Ξ: クシー　(15) o: オミクロン　(16) π, Π: パイ
(17) ρ: ロー　(18) σ, Σ: シグマ　(19) τ: タウ　(20) υ, Υ: ウプシロン
(21) ϕ, Φ: ファイ　(22) χ: カイ　(23) ψ, Ψ: プサイ　(24) ω, Ω: オメガ

その他

(1) $x \in X$ と書いたら，「x は集合 X に属する」すなわち「x は X の元 」という意味．
(2)「…をみたす X の元全体の集合」を

$$\left\{x \in X \;\middle|\; (\ x \text{ に関する条件})\right\}$$

の形で表す．たとえば「$\mathbb{N} = \left\{n \in \mathbb{Z} \;\middle|\; n > 0\right\}$ 」
(3) $X \subset Y$ と書いたら，「集合 X は集合 Y に含まれる」という意味．$X \subseteqq Y$ と同じ意味．
(4) $A := B$ と書いたら A を B で定義する，という意味．たとえば

$$e := \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n$$

．
(5)（文章 1）$:\Longleftrightarrow$（文章 2）と書いたら，（文章 1）を（文章 2）と定義する，という意味．たとえば「数列 $\{a_n\}$ が上に有界 $:\Longleftrightarrow$ ある実数 M が存在して，すべての自然数 n に対し $a_n \leq M$.」

改訂履歴等

- 20250412 一橋大学での講義ノートをまとめたものを β 版として公開.

目次

第1講　Julia とは？

Julia は Python なみの「書きやすさ」と C 言語なみ「高速さ」を備えるフリー（無料）のプログラミング言語です．2018 年に最初のバージョンがリリースされたあと，種々のライブラリも充実し，研究者の間で着実に広まっているように思います．

本講義では，冒頭（ii ページ）の「Julia のプログラミング環境について」の部分で述べたようなプログラミング環境が構築されたパソコンが手元にあることを想定しています．各自，インターネットなどを参照しながらそのような環境をご準備ください[*1]．

1.1　さっそく動かしてみよう

REPL による Julia の実行　Julia には **REPL** (Read-Eval-Print Loop) というモードがあります．これは「対話型実行環境」で，命令の入力と実行結果の出力を交互に繰り返しながら，次第に複雑な計算をこなしていくことができます．

REPL の起動方法はお使いの環境によって異なります．基本的には，ターミナル（コマンドプロンプト）を起動し，`julia` と入力するだけです[*2]．下のような画面が出てきたでしょうか：

以下，原則として 緑の `julia>` の右側にある式（命令）を自分で入力し，shift + enter

[*1] 一橋大学の図書館や情報教育棟のコンピューター端末（Windows マシン）には，Julia 言語とグラフ描画用のライブラリである `Plots`，汎用的なプログラミング用エディターである VSCode (Visual Studio Code) がすでにインストールされています．

[*2] 一橋大学の端末では，以下の方法で REPL が起動できます．その 1：画面左下のスタートメニュー → Julia のアイコンがあれば，それをクリック．なければ，検索バーの中に “Julia” と書くことでアイコンが表示されます．それをクリック．その 2：画面左下のスタートメニュー → VSCode をクリック → ウィンドウ上部のメニューの ··· からターミナル（Terminal）→ New Terminal → julia と入力．

(Shift キーを押しながら Enter を押す) とすることで実行していきます．ただし，1 行だけの命令のときは enter だけでも実行できます．

練習 1.1

```
julia> 1 + 2 - 3
0
julia> 4*(5 + 6)/7
6.285714285714286
julia> 4*(5 + 6)//7
44//7
julia> 3^10
59049
```

割り算 / と // の出力の違いに注目しましょう．前者は数値的な割り算であり，後者は「分数」としての割り算です．Julia は有理数を認識できるのです．

練習 1.2

```
julia> 2*pi
6.283185307179586
julia> 2pi
6.283185307179586
```

円周率 π の値は pi とすれば利用できます[*3]．また，積の * は省略できますが，**本講義では積に対し明示的に * を書くことを推奨します**．なお，2␣pi のように空白を入れても積とは認識されません．

複素数を扱うこともできます．虚数単位 i は im となります：

練習 1.3

```
julia> im^2
-1 + 0im
julia> (1-2im)*(1+2im)
```

[*3] \pi と入力して Tab を押すと π が入力でき，\euler と入力して Tab を押す自然対数の底 e が入力できます．しかし，**本講義での利用は非推奨とします**．

```
5 + 0im
```

組み込み関数の利用　私たちがよく使う初等的な関数は「組み込み関数」として利用できます．

練習 1.4

```
julia> sin(pi/4)
0.7071067811865475
julia> log(exp(3))
3.0
julia> exp(pi*im)
-1.0 + 1.2246467991473532e-16im
```

最後の式はオイラーの等式 $e^{\pi i} = -1$ です．虚部は本来 0 となるはずなのですが，計算誤差のために $1.2246467991473532 \times 10^{-16} i$ という値がでてしまっています．誤差については別の機会に学びましょう．

よく使う組み込み関数を挙げておきます：

絶対値 $\lvert x \rvert$	`abs(x)`
指数関数 e^x	`exp(x)`
b を底とする対数 $\log_b x$	`log(b,x)`
$\sin x, \cos x, \tan x$	`sin(x), cos(x), tan(x)`
$\mathrm{Re}\, z$, $\mathrm{Im}\, z$	`real(z), imag(z)`
平方根 $\sqrt{x}$	`sqrt(x)`
自然対数 $\log(x)$	`log(x)`
常用対数 $\log_{10} x$	`log10(x)`
階乗 $n!$	`factorial(n)`
x 以下の最大整数	`floor(x)`

問題 1.1 e^{π}, π^{e}, $\pi+20$ のうち，もっとも小さいものはどれか？（Hinte は $e = exp(1)$ とすれば得られる．）

定数や関数を自作する　1 回のセッション（Julia の起動から終了まで）の間，定数や関数を自作して利用することができます．

練習 1.5

```
julia> a = 2
2
julia> a + a^2
6
julia> b = a^5; b - 1
31
```

1 行目で a に 2 という値を**代入**しています．Julia（というか，大概のプログラミング言語）における記号 = は「等号」ではなく，「左辺に右辺の値を代入します」という意味です．3 行目ではそれを利用して $2+2^2$ という計算をしたことになります．5 行目では，b に a^5，すなわち $2^5 = 32$ を代入したあと，b-1，すなわち $32-1$ の値を出力します．途中のセミコロン ; は，「この結果は出力しない」という意味です．そのため，6 行目のように 31 だけが出力されたのです．

問題 1.2 $a = \sqrt{3}+\sqrt{2}\,i$, $b = \sqrt{3}-\sqrt{2}\,i$ のとき，$c = \dfrac{a}{b}+\dfrac{b}{a}$ と $d = a^4-2a^2$ の値を求めよ．**（センター追試）**

問題 1.3 $\sqrt{42+12\sqrt{6}}$ の整数部分を p，小数部分を q とするとき，p, q, $r = \dfrac{p}{q(q+4)}$ の値を求めよ．**（成蹊大）**

次に関数を自作します．$f(x) := x^3$ と定義し，$f(5) = 5^3 = 125$，$f(i) = i^3 = -i$ を計算させます：

練習 1.6

```
julia> f(x) = x^3
f (generic function with 1 method)
```

```
julia> f(5)
125
julia> f(im)
0 - 1im
```

1.2　グラフの描画

REPL からグラフ描画用のパッケージ Plots を呼び出して，簡単なグラフを書いてみましょう．

まずは Plots 召喚のおまじない：

練習 1.7

```
julia> using Plots
```

これには少しだけ時間がかかります．次に好きな関数を定義しましょう：

練習 1.8

```
julia> f(x)=sin(x^2)/x
f (generic function with 1 method)
```

関数にさきほどと同じ `f(x)` を利用しました．この場合，以前の $f(x) = x^3$ を上書きされてしまいます．上書きを避けたければ `g(x)` や `myFunction(x)` など，自由に名前をつけても構いません．

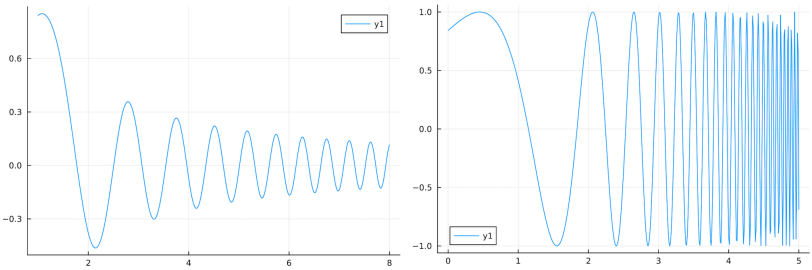
あとは描画の操作です：

練習 1.9

```
julia> x=1:0.01:8
1.0:0.01:8.0
julia> plot(x,f)
```

1 行目は，「1 から 8 まで，0.01 刻みで動かしたリスト（数列）を作り，`x` とせよ」という命令です（出力はセミコロン ; で抑制してもかまいません）．2 行目の `plot(x,f)`

(plots としがちなので注意) は，「直前で作ったリスト x について関数 f(x) の値を計算し，描画せよ」という意味です．



複数の行を一度に実行させる　以上のように，1 行書いては実行する，という操作を繰り返すのは面倒です．その場合は，命令を let と end で挟むことで複数行にまたがる命令を書くことができます（enter で改行，end まで書いたところで enter あるいは shift + enter で実行）：

練習 1.10

```
julia> let
       g(x)=sin(exp(x))
       x=0:0.01:5
       plot(x,g)
       end
```

問題 1.4 以下の方法でも $g(x) = \sin(\exp(x))$ のグラフを描くことができるので，実行してみよ．

(1) g(x)=sin(exp(x)); plot(g)
(2) g(x)=sin(exp(x)); plot(g, 0, 5)
(3) g(x)=sin(exp(x)); plot(g, 0:0.1:5)
(4) plot(x -> sin(exp(x)),-5,10)

第2講　Julia 入門つづき / 指数関数の計算

今回はまず， VS Code で REPL を実行しましょう．

VS Code について　VS Code（Visual Studio Code）は高機能かつ無料のプログラミング用のエディターです．REPL は長いコード（プログラム）を書くことには適していないので，テキストエディタを用いて Julia のコードを書き，実行することができます[*1]

VS Code の中でターミナルを開いて julia を起動させることもできます．画面（ウィンドウ）上部にあるメニューから「ターミナル」を選んで起動し，julia と入力してください．

2.1　式の真偽

Julia に与えられた条件（命題）に対し，真 (`true`) か偽 (`false`) を判定してもらいます．

たとえば，$2 = 3$ という等式が成り立つかを判定してもらいましょう．等号 $=$ は `==` のようにイコールを 2 つ並べて表現します[*2]．

練習 2.1

```
julia> 2 == 3
false
```

その他にも，Julia は次のような条件式を判定できます：

`x == y`	x は y と等しい	`x >= y`	x は y 以上
`x != y`	x は y と等しくない	`x <= y`	x は y 以下
`x > y`	x は y より大	`(x == y)&&(y > z)`	x == y かつ y > z
`x < y`	x は y より小	`(x == y)||(y > z)`	x == y または y > z

[*1] 画面左下のスタートメニュー → VSCode のアイコンをクリック → 左上のメニューから「新しいファイル」→ Julia File．1 行 1 行を実行するときは shift + enter，すべてをまとめて実行するには alt + enter．

[*2] 1 つだけの = は代入に使うのでした．

練習 2.2

```
julia> 2 == 2
julia> 2 < 3
julia> 3 <= 5
julia> (3 < 4) && (4 < 3)
julia> (3 < 4) || (4 < 3)
julia> 3 < 4 < 5
julia> 3 < 7 > 5
```

4 行目と 5 行目の丸括弧（ ）は省略できるのですが，本講義では他の言語との兼ね合いや読みやすさを考慮して，括弧をつけることを推奨します．同様の理由で，6 行目と 7 行目のような条件式も本講義では推奨しません．

2.2 if 文による条件分岐

条件分岐とは，ようするに「場合わけ」のことです．プログラミング言語では，ふつう **if 文** (if statement) といわれる構文をもちいて条件分岐を実現します．

簡単な例として，機械学習でよく用いられる **ReLU**(Rectified Linear Unit) とよばれる関数を定義し，そのグラフを描いてみましょう．ReLU を数学風に書くと，こんな感じです：

$$f(x) = \begin{cases} 0 & (x < 0) \\ x & (x \geq 0) \end{cases}$$

ここからは VS Code で新たに.jl ファイルを作成し，複数行にまたがるコード（プログラム）を書いていきましょう（1 行ごとに実行する場合は Shift + Enter，全体を実行するときは Alt + Enter）．以下のコード（プログラム）では，**緑色の # で始まる部分はコメントなので入力不要です．**

練習 2.3

```
using Plots      # Plots の読み込み
function f(x)    # 関数 f(x) の定義はじめ
    if x < 0     # if 文はじめ，x < 0 のとき
        return 0     # 値 0 を返す
    else         # それ以外（x >= 0）のとき
```

```
6           return x     # 値 x を返す
7       end              # if 文おわり
8   end                  # 関数の定義おわり
9   plot(f)              # グラフの描画
```

4 行目・6 行目の return は省略してもかまいません．また，次のように，elseif を用いて条件分岐を増やしていくこともできます：

練習 2.4

```
1   function g(x)        # 関数 g(x) の定義はじめ
2       if x < 0         # if 文はじめ，x < 0 のとき
3           0            # 値 0 を返す
4       elseif x < 1     # 0 <= x < 1 のとき
5           x            # 値 x を返す
6       else             # x > 1 のとき
7           sqrt(x)      # 値 sqrt(x) を返す
8       end              # if 文おわり
9   end                  # 関数の定義おわり
10  plot(g, -2, 5)        # グラフの描画
```

問題 2.1 if 文をもちいて関数 $h(x) = \min\{x+1, -2x+3\}$ を定義し，グラフを描け．

2.3　ループと文字の出力

特定の作業を一定の回数くり返し行いたいときは，**for 文**（for statement）や **while 文**（while statement）を用いてループ処理の命令を利用します．今回は for 文の使い方を練習しましょう．

最初に，1 から 10 までの整数を順に書き出すプログラムです：

練習 2.5

```
1   for i = 1:10      # for 文はじめ，i は 1 から 10 まで
```

```
    println(i)    # i を出力して改行，を繰り返す
end               # for 文おわり
```

1 行目の `i = 1:10` は正確にいうと，`i` を 1 から 10 まで「刻み幅 1 で」変化させよ，ということです．「刻み幅 0.5」にしたければ，1 行目を `for i = 1:0.5:10` とします．

刻み幅は負の数でもかまいません：

練習 2.6

```
for i = 10:-1:1   # for 文はじめ
    println(i)    # i を出力して改行，を繰り返す
end               # for 文おわり
```

次に，for 文を用いて 1 から 100 までの和を計算してみましょう．

練習 2.7

```
a = 0             # a の初期値を設定
for i = 1:100     # for 文はじめ
    a = a + i     # a + i の値を a に代入
end               # for 文おわり
a                 # a の値を表示
```

5050 という答えが出てきたでしょうか．3 行目はプログラミング言語特有の（等式ではない）書き方で，「以前の `a` に `i` を足したものを，新たに `a` に代入せよ」という意味です．

問題 2.2 for 文を用いて 1 から 100 までの奇数の和を計算せよ．

ひとつ，for 文を用いて「階乗もどき」関数 `ff(x)` (fake factorial) を定義してみましょう．

練習 2.8

```
function ff(x)    # 関数定義はじめ
    a = 1
    for i = 1:x   # for 文はじめ
        a = a * i
```

```
    end                 # for 文おわり
    return a
end                     # 関数定義おわり
```

問題 2.3 上で定義した関数 `ff` に対し，`x = 4, 5, 5.3, -2` での値を求め，その値がでた理由を考えよ．

2.4 誤差とは

以下では，自然対数の底 $e = 2.718281828459\cdots$ の数値計算法について考えてみます．e は無理数であることが知られているので，その小数展開は循環小数ではありません．したがって，e を正確に計算することは**不可能**なのです．ただし，「計算する」とは「有限の時間内に，有限回数の計算で真の値を求めること」という意味で使っています．

仕方ないので，私たちは e を有理数，とくに有限小数によって近似値＼します．たとえば，有限小数である

$$2,\ 2.7,\ 2.71,\ 2.718,\ \cdots$$

はそのような近似値の列になっています．

以下では，そのような近似値の列を 2 つの公式

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!} + \cdots$$

に基づいて数値計算してみましょう．

さまざまな誤差 その前に，誤差というものについて考えます．いま，求めたい真の値 A に対し近似値 a が与えられたとき，

$$a - A$$

を a の A からの**誤差** (error) といい，

$$|a - A|$$

を**絶対誤差** (absolute error, あるいは，単に **誤差**) といいます．もし $A \neq 0$ のときは，誤差の割合（何 % ぐらいの誤差か）が重要になるので，

$$\frac{|a - A|}{|A|}$$

という量を考えます．これを a の A に対する**相対誤差** (relative error) といいます．

問題 2.4 Julia の REPL を用いて以下の a の A に対する相対誤差を求めよ．

(1) $a = 1.41$，$A = \sqrt{2}$.　　(2) $a = 3.0$，$A = \pi$.

ただし，便宜的に `sqrt(2)` と `pi` をそれぞれ A の真の値とみなしてよい．

2.5　浮動小数点数と数の型

コンピューターで数値計算をやるにあたって，コンピューターが実際に扱うことができる数について少し知っておきましょう．

浮動小数点数　ひとつの記憶素子（0 または 1 の値をとる半導体回路）が蓄えることができる情報量を 1 **ビット**といい，8 ビットを普通 1 **バイト**といいます．現在のパソコンは何十ギガバイトものデータを扱うことができますが，1 ギガバイトは 2^{30} バイト，すなわち 2^{33} ビット（約 85.9 億ビット）に相当します．莫大な量でいまひとつ実感がわきませんが，それでも有限の情報量であることには違いありません．

さて，数値計算で主に用いられるのは，**浮動小数点数**とよばれる

$$\pm\Big(1 + \frac{p_1}{2} + \frac{p_2}{2^2} + \frac{p_3}{2^3} + \cdots + \frac{p_N}{2^N}\Big) \times 2^e$$

(ただし p_k は 0 もしくは 1，e は整数）の形の数です．これは 2 進数による「指数表記」であり，括弧内を**仮数** (significand)，e を**指数** (exponent) といいます．もちろん N や e は有限の値です．数値計算でよく用いられるのは**倍精度** (double precision) とよばれる浮動小数点数の体系で，通常 $N = 52$，e は -1022 から 1023 までの整数とします．符号（±）に 1 ビット，仮数に 52 ビット，指数に 11 ビットが用いられるので，合計 64 ビットで表現される数ということになります．

浮動小数点数と誤差　倍精度浮動小数点数の体系では結局，高々 2^{64} 種類の数しか扱うことができません．すなわち，実数全体を高々 2^{64} 個の浮動小数点数（有理数）で近似しているわけです．

あらゆる四則演算をこの範囲で行うのですから，必然的に誤差が生じます．仮に A と B が浮動小数点数でも，積 $A \cdot B$ や商 A/B は浮動小数点数でないかもしれません．この場合は適当にその答を有限個の浮動小数点数の中から選んで近似することになります．$\sin A$ や e^B といった関数の値を考えるときも同様です．このような近似によって発生する誤差を**丸め誤差** (rounding error) といいます．

Julia における数の取り扱い　多くのプログラミング言語では，データをメモリ内に効率よく保存したり加工したりするために，データに**型** (type) というものを与えて最適化しています．C 言語や Java のような言語だと，コードを書く際に明示的に変数の型を指定しなくてはならず，それが結構面倒だったりするのですが，Python や Julia はそれを自動的に「解釈」する機能が備わっているため，楽にプログラミングを進めることができるのです．したがって，本講義ではあまり変数の型について細かい注意はいらない（しない）のですが，念のために変数の型の種類について大雑把に知っておきましょう．

たとえば，「2」という実数が Julia の中でどのように扱われるかをみてみます．REPL で構いませんので，次のコードを打ってみましょう：

練習 2.9

```
julia> typeof(2)        # 64 ビット整数
Int64
julia> typeof(2.0)      # 64 ビット浮動小数点数（倍精度）
Float64
julia> typeof(2.0im)    # 64 ビット浮動小数点複素数
ComplexF64 (alias for ComplexFloat64)
```

ここで使っている `typeof` 関数は，その値が Julia の中でどのような数として認識されているかを教えてくれる関数です．1 行目の `typeof(2)` というのは，2 と入力された文字が 2 行目で出力された `Int 64` という型の情報として処理されれていることを意味します．`Int 64` は 64 ビットの情報量で表現された整数です．3 行目では 2 を `2.0` を変えただけですが，型が `Float 64` になっています．これはいわゆる倍精度（64 ビット）の浮動小数点数で，もっとも一般的な実数の型です．つまり，このように書き方を変えることで，コードを書く側が明示的に型の違いを表現できるわけです．5 行目は純虚数 $2i$ ですが，この場合は実部と虚部がそれぞれ倍精度（64 ビット）の浮動小数点で表される複素数の型である `ComplexF64` として認識されています[*3]．

[*3] 第 4 講では，任意の精度を扱うことができる `BigFloat` 関数について学びます．

2.6　e の計算

では，e の数値計算をやってみましょう．数列

$$a_n = \left(1 + \frac{1}{n}\right)^n$$

と

$$b_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

を e の近似列として採用し，比較してみます．最初に $n = 1$ から 10 で比べてみましょう．

a_n の場合　まず単純に，

練習 2.10

```
for n = 1:10
    println((1+1/n)^n)
end
```

としてみます．結果をみるとわかりますが，$n = 10$ でも 2.6 弱で，あまり良い数字がでません．

練習 2.11

```
for n = 1:100
    println((1+1/n)^n)
end
```

と変えてみても，$n = 100$ でせいぜい 2.7 程度です．

b_n の場合　こちらは和の計算ですから，さきほどのテクニックが使えます：

練習 2.12

```
b = 1
for n = 1:10
    b = b + 1/factorial(n)
```

```
    println(b)
end
```

こちらはかなり優秀です．$n = 4$ で早くも 2.7 超え，$n = 10$ では 8 桁ぐらい数値が一致しています．じつは，絶対誤差でいうと $|a_n - e|$ は $1/n$ に比例する程度であり，$|b_n - e|$ は $1/n!$ に比例する程度であることが知られています．したがって，

> **e の定義式にはふつう数列 a_n を用いるが，その数値計算には数列 b_n を用いるべき**

だとわかります．数値解析の理論が目指すのは，b_n のような良い数列を見つけ出し，さらに誤差の評価を与えることだといえるでしょう．

問題 2.5 次のようなコードを書け：

(1) $n = 1$ から 10 に対し，絶対誤差 $|a_n - e|$ を出力する
(2) $n = 1$ から 10 に対し，絶対誤差 $|b_n - e|$ を出力する

n	a_n	$\lvert a_n - e\rvert$	b_n	$\lvert b_n - e\rvert$
1	2.000000000	0.7182818285	2.000000000	0.7182818285
2	2.250000000	0.4682818285	2.500000000	0.2182818285
3	2.370370370	0.3479114581	2.666666667	0.05161516179
4	2.441406250	0.2768755785	2.708333333	$9.948495126 \times 10^{-3}$
5	2.488320000	0.2299618285	2.716666667	$1.615161792 \times 10^{-3}$
6	2.521626372	0.1966554567	2.718055556	$2.262729035 \times 10^{-4}$
7	2.546499697	0.1717821314	2.718253968	$2.786020508 \times 10^{-5}$
8	2.565784514	0.1524973145	2.718278770	$3.058617775 \times 10^{-6}$
9	2.581174792	0.1371070367	2.718281526	$3.028858530 \times 10^{-7}$
10	2.593742460	0.1245393684	2.718281801	$2.731266076 \times 10^{-8}$

第3講　方程式の数値解法 1：二分法

3.1　方程式の数値解法

「方程式 $x^2-2=0$ を解け」といわれたら，反射的に $x=\pm\sqrt{2}$ と式変形をして，それで満足してしまうものです．しかし，$\sqrt{2}$ という記号は「$x^2=2$ を満たす正の数」がその定義ですから，これで本当に方程式を解いたことになっているのか，疑問が残ります．

より実用的な答えは，「$\sqrt{2}=1.4142\cdots$」といった具体的な数値を与えることでです．もちろん，$\sqrt{2}$ は無理数なので，何かしら合理的な判断のもと，有限の桁数で切った値を「近似値」として採用する必要があります．

今回から数回にわけて，「与えられた x に対し，関数 $f(x)$ の値は任意の精度で計算できる」という仮定のもとで，方程式 $f(x)=0$ の解の値を任意の精度で数値計算する方法を学びます．

なお，「任意の精度で計算する」というのは，「誤差の許容範囲を自由に設定し，その範囲におさまるような近似値を計算する」という意味です．

3.2　二分法

以下では，

連続関数 $y=f(x)$ に対し，方程式 $f(x)=0$ の解 α を数値的に求めよ

という問題を考えていきます．もちろん，付随して誤差の評価も行いましょう．

そのような数値計算で，もっとも初歩的な方法なのが，**二分法**(bisection method) とよばれる次のアルゴリズムです：

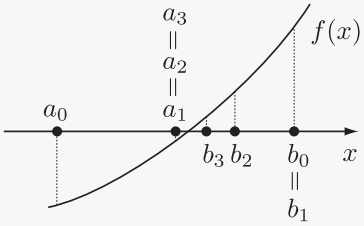
二分法のアルゴリズム．　$y=f(x)$ を連続関数とする．

(1) $f(a_0)<0, f(b_0)>0$ となるペア (a_0,b_0) を見つける．必要であれば $f(x)$ のかわりに $-f(x)$ を考えることで，$a_0<b_0$ と仮定してよい．

(2) $f(a_n)<0, f(b_n)>0, a_n<b_n$ を満たすペア (a_n,b_n) が与えられているとき，その中点を $m_n:=\dfrac{a_n+b_n}{2}$ とおく．

(3) $f(m_n)$ の値を計算し，

- $f(m_n) < 0$ ならば $(a_{n+1}, b_{n+1}) := (m_n, b_n)$ とおいて (2) に戻る．
- $f(m_n) > 0$ ならば $(a_{n+1}, b_{n+1}) := (a_n, m_n)$ とおいて (2) に戻る．
- $f(m_n) = 0$ ならば $\alpha := m_n$，計算終了．



二分法は解の存在まで保証するアルゴリズムです．理論的には，次のように正当化されます：

定理 3.1 (二分法の収束性) 上記の二分法に関して，次のいずれかが成り立つ：

(a) ある自然数 n に対し $f(m_n) = 0$ となる．よって $\alpha := m_n$ は解のひとつ．
(b) すべての自然数 n で $f(m_n) \neq 0$ となるが，数列 $\{a_n\}$ と $\{b_n\}$ は $n \to \infty$ のときそれぞれ同じ極限 α に収束し，$f(\alpha) = 0$ を満たす．

いずれの場合も，次のような誤差の評価式が成り立つ．

$$\max\{|a_n - \alpha|,\ |b_n - \alpha|\} \leq \frac{|b_0 - a_0|}{2^n}.$$

したがって，十分大きな n に対し a_n もしくは b_n を α の近似値として用いることができるわけです．

【証明】 (a) でなければ，すべての自然数 n で $f(m_n) \neq 0$ である．このときアルゴリズムの (2) と (3) を繰り返すと，数列 $\{a_n\}$ は単調増加かつ $a_n < b_0$ （上に有界），数列 $\{b_n\}$ は単調減少かつ $a_0 < b_n$ （下に有界）である．よって「実数の連続性」[*1]より，それぞれある実数に収束する．$\alpha = \lim_{n\to\infty} a_n$ とすると，$n \to \infty$ のとき $|b_n - a_n| = |b_0 - a_0|/2^n \to 0$ より $b_n = (b_n - a_n) + a_n \to 0 + \alpha = \alpha$ $(n \to \infty)$. $f(x)$ は連続であったから，$f(\alpha) = \lim_{n\to\infty} f(a_n) \leq 0$ かつ $f(\alpha) = \lim_{n\to\infty} f(b_n) \geq 0$. よって $f(\alpha) = 0$ となる．

また，(a)，(b) いずれの場合も $a_n \leq \alpha \leq b_n$ であるから，

$$\max\{|a_n - \alpha|,\ |b_n - \alpha|\} \leq |b_n - a_n| = \frac{|b_0 - a_0|}{2^n}.$$ ■

[*1] 「単調増加（減少）かつ上に（下に）有界な数列はある実数収束する」という実数のもつ性質のこと．

例 1（$\sqrt{2}$ の計算）　連続関数 $f(x) = x^2 - 2$ に対し，$(a_0, b_0) = (1.0, 2.0)$ として二分法のアルゴリズムに従って計算したのが次の表です[*2]．

n	a_n	b_n
0	1.00000	2.00000
1	1.00000	1.50000
2	1.25000	1.50000
3	1.37500	1.50000
4	1.37500	1.43750
5	1.40625	1.43750
6	1.40625	1.42188

n	a_n	b_n
7	1.41406	1.42188
8	1.41406	1.41797
9	1.41406	1.41602
10	1.41406	1.41504
11	1.41406	1.41455
12	1.41406	1.41431
13	1.41418	1.41431

n	a_n	b_n
14	1.41418	1.41425
15	1.41418	1.41422
16	1.41420	1.41422
17	1.41421	1.41422
18	1.41421	1.41422
19	1.41421	1.41422
20	1.41421	1.41421

定理 3.1 より，数列 a_n, b_n は $\alpha^2 - 2 = 0$ を満たす正の実数 α に収束します．すなわち，$\sqrt{2} = 1.41421356\cdots$ の近似値を与えるわけです．また，その誤差は $|b_0 - a_0|/2^n = 1/2^n$ 以下であることもわかります[*3]．

3.3　中間値の定理と二分法

方程式の数値解法としての二分法は収束も遅く，それほど有用とはいえないのですが（次回学ぶ「ニュートン法」も参照），**中間値の定理** (Intermediate Value Theorem, IVT) の実質的な証明を与えてくれる点は特筆すべきでしょう．

この定理は，連続関数のグラフが「つながっている」ことの根拠とされるものです：

定理 3.2（中間値の定理） 閉区間 $[a, b]$ 上で定義された連続関数 $y = f(x)$ が $f(a) \neq f(b)$ を満たすとき，$f(a)$ と $f(b)$ の間にある任意の実数 ℓ に対し，$f(c) = \ell$ を満たす c が区間 (a, b) に少なくともひとつ存在する．

【証明（定理 3.2）】 $f(x)$ は連続なので，$F(x) = f(x) - \ell$ も連続関数である．これに $a = a_0$, $b = b_0$ として二分法のアルゴリズムを適用すると，定理 3.1 より $F(c) = 0$ を満たす c が区間 (a, b) に見つかる．よって $f(c) = \ell$．■

[*2] ここに現れる (a_n, b_n) はすべて $k/2^n$ の形の有理数なのですが，表の中では収束の様子が分かりやすいように小数で表現しています，

[*3] 厳密にいうと，定理 3.2 や定理 3.1 からわかるのは「$\alpha^2 = 2$ を満たす α が区間 $[1, 2]$ 内に少なくともひとつ存在する」ということだけです．この α が本当に，私たちが $\sqrt{2}$ とよぶ唯一の数であることを示すには，別の根拠が（関数の単調性）が必要となります．

3.4　二分法の Julia による実装（$\sqrt{2}$ の計算）

二分法に限ったことではありませんが，アルゴリズムの数学的な定式化をそのままプログラムとして書き下すと冗長になってしまうことが多いものです．そこで，アルゴリズムを適切に解釈して，効率的にプログラミングする必要があります[*4]．

以下では例 1 にしたがって，$\sqrt{2}$ を計算してみましょう．具体的には $f(x) = x^2 - 2$ に対し，二分法のアルゴリズムを適用するわけですが，

- 最初は特殊な例に対する最小限のコードを書き，
- 次第に汎用性の高いプログラムに改良していく

というプロセスをとります．

$\sqrt{2}$ の計算：その 1　まずは $f(1) = 1^2 - 2 = -1 < 0$，$f(2) = 2^2 - 2 = 2 > 0$ であることと，$f(x)$ が区間 $[1, 2]$ で単調増加であることを既知として，$(a_0, b_0) = (1.0, 2.0)$，$n = 10$ に対し二分法のアルゴリズムを適用するコードを書いてみましょう．

練習 3.1

```
f(x)= x^2 - 2            # 関数の定義
a = 1.0; b = 2.0         # 区間の初期値は [1.0, 2.0]
for i = 1:10             # 反復回数の設定，a_i と b_i を決める for 文
    m = (a + b)/2        # 中点の計算
    if  f(m) < 0         # f(m) が負のとき
        a = m
    else                 # f(m) が 0 以上のとき
        b = m
    end
end
println(a)               # a の値を書き出し
println(b)               # b の値を書き出し
```

[*4] 一方で，効率を追い求めてあまりに技巧的になってしまうと，コードの可読性が損なわれてしまうので，注意が必要です．あとで自分でコードを読み返すときのことを考えて，適宜コメントを入れるなど，わかりやすいコードを書く習慣をつけましょう．

数列が出てこないので変な感じがしますが，プログラムを追っていくと a と b が数列 $\{a_n\}$ と $\{b_n\}$ を順に計算していることがわかります．

実行するとターミナルに a_{10} と b_{10} に対応する

```
1.4140625
1.4150390625
```

という結果が出力されるはずです．丸め誤差を無視すれば $\sqrt{2}$ が $1.4140625 \leq \sqrt{2} \leq 1.41455078125$ を満たすことがわかります．定理 3.1 によれば，理論上これらの値は $\sqrt{2}$ から $|b_0 - a_0|/2^n = 2^{-10} < 0.001$ 以下の絶対誤差だということがわかります．つまり，必要な精度に合わせて n を事前に調整することも可能なわけです．

$\sqrt{2}$ の計算：その 2　さて最初の改良ポイントですが，途中どういう計算を経て結果の出力が出てきたのかがわからず面白くないので，次のように変更してみましょう．

問題 3.1 練習 3.1 の 11 行目と 12 行目を削除し，

```
    println("Step 0: [$a, $b]")
```

と

```
    println("Step $i: [$a, $b]")
```

という 2 つの行を挿入する．それぞれをどこに挿入すれば，次のような実行結果が出せるか？

```
Step 0: [1.0, 2.0]
Step 1: [1.0, 1.5]
Step 2: [1.25, 1.5]
Step 3: [1.375, 1.5]
Step 4: [1.375, 1.4375]
Step 5: [1.40625, 1.4375]
Step 6: [1.40625, 1.421875]
Step 7: [1.4140625, 1.421875]
Step 8: [1.4140625, 1.41796875]
Step 9: [1.4140625, 1.416015625]
Step 10: [1.4140625, 1.4150390625]
```

ここで，println("Step 0: [$a, $b]") という命令の意味を説明しておきます．まず，出力のように何らかの文字列を出力したいときは，基本的に println("Step 0: [a, b]") のように引用符 " " を用いればよいのですが，これではただ Step 0: [a, b] と出力されて実際の a と b の値が反映されません．そこ

で，たとえば a を $a と書くことでその時点の a の値が文字列に変換されて画面上に出力されるわけです[*5]．

$\sqrt{2}$ の計算：その 3　次に，プログラムの汎用性を高めていきましょう．具体的には以下のような変更を加えます：

- 最初の区間の端点 a，b を自由に選べるようにする
- 反復回数 n を自由に変えられるようにする
- アルゴリズム全体を関数[*6]として定義する．すなわち，「区間の端点と反復回数を入れると，上のような結果を出力する関数」を作る．

3 番目のポイントは少しわかりづらいのですが，プログラムの高速化や，コードを整理するためには重要なプロセスです．実際のコードを見てみましょう：

練習 3.2

```
function sqrt2(a, b, n)      # 関数の定義はじめ
    f(x)= x^2 - 2
    println("Step 0: [$a, $b]")
    for i = 1:n
        m = (a + b)/2
        if f(m) < 0
            a = m
        else
            b = m
        end
        println("Step $i: [$a, $b]")
    end
end                          # 関数の定義おわり
sqrt2(0, 20, 15)             # a=0, b=20, n=15 の例を出力
```

[*5] println(a) というのは，println("$a") と同じことです．

[*6] 日本語の「関数」はもともと「機能」を意味する function の訳語で，プログラミング言語における「関数」はまさに，入力されたものを操作して何らかの結果を出力をする「機能」という意味で用いられます．

まず sqrt2(a, b, n) という関数を定義してから，最後の 14 行目でその関数を実行しています．いろいろと値を変えて実験できるので，わざわざプログラムを書き換える必要がありません．

しかし，値によっては正しく動作しない場合もあります：

問題 3.2 sqrt2(5, 10, 15) を実行し，なぜそのようなその結果が出力されたか考えよ．

ポイントは，「二分法は，区間の端点における f(x) の符号が異なるからこそうまくいく」ということです．つまり f(a) < 0 かつ f(b) > 0 でないかぎり，7–11 行目の if 文の意味が数学的に不明瞭になっています．動作保証外ということです．

3.5　さらなる汎用化

高校数学でもおなじみですが，中間値の定理によれば，

> $f(a)f(b) < 0$ であれば，a と b の間に $f(x) = 0$ となる x が少なくとも 1 つ存在する

ことがわかります．したがって，$f(a)f(b) < 0$ を判定条件にして汎用性の高い「二分法を実行する関数」を作ることができます．

次の例では，方程式を定める関数 f も自由に選べるようにして，2 の立方根 $\sqrt[3]{2} = 1.259921\cdots$ を求めてみます：

練習 3.3

```
1  function bisec(f, a, b, n)
2      println("Step 0: [$a, $b]") # 区間の初期値を表示
3      for i = 1:n
4          if f(a)*f(b) > 0      # 中間値の定理が使えないなら
5              println("No root guaranteed in [$a, $b]")
6              break             # for 文を強制終了
7          else
8              m = (a + b)/2
9              if f(m)*f(a) > 0    # f(m) と f(a) 同符号
10                 a = m
```

```
11              else                    # f(m) と f(a) 異符号
12                  b = m
13              end
14              println("Step $i: [$a, $b]")
15          end
16      end                     # for 文 i = 1:n おわり
17  end                         # 関数の定義おわり
18  g(x) = x^3 - 2              # 具体例：2 の立方根で 0 となる関数
19  bisec(g, -1.0, 5.0, 10)
```

4 行目からの if 文では，中間値の定理が利用できる状況かどうかを確認しています．そうでなければ 5 行目で「この区間に解があるかは保証されません」という内容のメッセージを出し，6 行目で for 文の処理を強制終了する break という命令を実行します．

具体例を計算する 18 行目と 19 行目は，まとめて

```
bisec(x -> x^3 - 2, -1.0, 5.0, 10)
```

あるいは文字を変えて

```
bisec(y -> y^3 - 2, -1.0, 5.0, 10)
```

のように書いても大丈夫です．関数をいろいろ変えてみるとよいと思います．

問題 3.3 円周率 π の定義はいろいろあるが，その中に

$\cos x = 0$ となる最小の正の数の 2 倍

というものがある．二分法のアルゴリズム用いて，円周率を小数点以下 5 桁まで求めよ．

ちなみに，余弦関数 $\cos x$ は無限級数（マクローリン展開）を用いて

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

とすれば円（や円周率）を使わずに定義できます．また，「$\cos x = 0$ となる最小の正の数」が存在することは，$\cos 1 > 0$ かつ $\cos 2 < 0$ であることと，中間値の定理によって保証されます[*7]．

[*7] このような円周率の定義は，ドイツの数学者リヒャルト・バルツァー（Richard Baltzer，1818–1887）が導入し，エトムント・ランダウ（Edmund Landau, 1877–1938）が講義や自著に採用し世に広まりました．

第4講　方程式の数値解法 2：ニュートン法

4.1　ニュートン法

前回に引き続き，方程式 $f(x) = 0$ の解の近似値を任意の精度で求める方法を考えます．これは，与えられた関数 $y = f(x)$ に対し，$f(\alpha) = 0$ となる α を数値的に求める，ということに他なりません．

たとえば前回学んだ「二分法」は，単純ですが関数の形に依存しない万能なアルゴリズムです．しかし，解の近似値を n 桁の精度で得るためには，関数 $f(x)$ の値をだいたい n に比例する回数計算しなくてはなりません．これよりもっと効率よく，高速に解の近似値を計算できるのが，今回学ぶ「ニュートン法」です．

ニュートン法のアルゴリズム　ニュートン法は C^1 級関数[*1]なら適用可能なのですが，以下では話を簡単にするために C^∞ 級関数のみを考えることにしましょう．

いま，$f(\alpha) = 0$ を満たす α に十分近い p が与えられたとしましょう．このとき，$x = p$ におけるテイラー展開から

$$0 = f(\alpha) = f(p) + f'(p)(\alpha - p) + \cdots$$

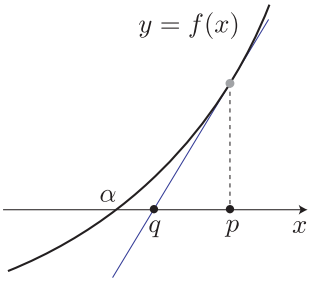
が成立します．右辺の $\cdots$ の部分を誤差として無視すれば，$f'(p) \neq 0$ のとき近似式

$$\alpha \approx p - \frac{f(p)}{f'(p)}$$

が得られます．$q := p - \dfrac{f(p)}{f'(p)}$ とおくと，q は p よりもっと α に近い数だと期待されます．その根拠は，次のように幾何学的に説明することができます．

[*1] n 階導関数が存在し，それが連続関数になっているものを C^n **級関数**というのでした．すべての自然数 n に対し C^n 級である関数は C^∞ **級関数**または**滑らかな関数**といいます．

まず関数のグラフ $y = f(x)$ の形状をある程度調べて，グラフが x 軸と交わるあたりに目星をつけましょう．そして，計算したい未知の解 $x = \alpha$ に対し，その付近の値 $x = p$ をひとつ選びます．次に点 $(p, f(p))$ における関数 $f(x)$ のグラフの接線を引き，x 軸との交点を求め，その x 座標を q とするのです．接線の方程式は $y = f(p) + f'(p)(x - p)$ ですから，$q = p - f(p)/f'(p)$ が成り立ちます．



図からも類推されるように，このとき q は p よりも α に近づいているはずです．したがって，「p から $q = p - f(p)/f'(p)$ を計算する」という操作を反復すれば，α に収束する数列が得られそうです．

この考察を基にしたのが，次の**ニュートン法**のアルゴリズムです：

ニュートン法 (Newton's Method).

(1) $y = f(x)$ のグラフと x 軸との交点 α に目星をつけ，その近くから初期値 x_0 を選ぶ．

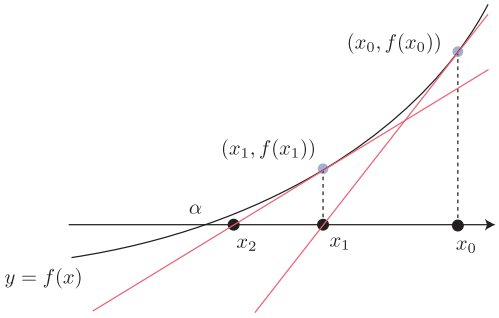
(2) 漸化式

$$\boldsymbol{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}} \quad (n = 0, 1, 2, \cdots) \tag{4.1}$$

で定まる数列を計算する．

(3) $|x_{n+1} - x_n|$ の値が一定以下になったところで計算を打ち切り，x_{n+1} を α の近似値として採用する．

たとえば $|x_{n+1} - x_n| < \dfrac{1}{10^M}$ であれば，x_{n+1} と x_n は「小数点以下 M 桁一致相当」の近さをもつことになります．



解への収束性　ニュートン法が生成する数列 $\{x_n\}$ が本当に解に収束することを証明しましょう[*2]．まず，方程式の「重複度」を定義します[*3]：

定義　(解の重複度) 関数 $y=f(x)$ に対し，$x=\alpha$ が $f(\alpha)=f'(\alpha)=\cdots=f^{(m-1)}(\alpha)=0$ かつ $f^{(m)}(\alpha)\neq 0$ を満たすとき，すなわち，関数 $f(x)$ の $x=\alpha$ におけるテイラー展開が

$$f(x)=A(x-\alpha)^m+o((x-\alpha)^m)$$

($A\neq 0$) の形で書けるとき，α を**関数 $f(x)$ の重複度 m の零点**，もしくは**方程式 $f(x)=0$ の重複度 m の解**とよぶ．

このとき，次が成り立ちます：

定理 4.1 (ニュートン法の収束性) α を関数 $f(x)$ の重複度 m の零点とする．このとき，α に十分近い初期値 x_0 を選べば，漸化式 (5.3) が定める数列 $\{x_n\}$ は α に収束する．より精密に，ある定数 λ $(0<\lambda<1)$ が存在して，すべての n に対し

$$\boldsymbol{|x_{n+1}-\alpha|\leq\lambda|x_n-\alpha|} \tag{4.2}$$

が成り立つ．したがって $n\to 0$ のとき，

$$\boldsymbol{|x_n-\alpha|\leq\lambda^n|x_0-\alpha|\to 0.} \tag{4.3}$$

さらに $m=1$ のときは，ある定数 $K>0$ が存在して，すべての n に対し

$$\boldsymbol{|x_{n+1}-\alpha|\leq K|x_n-\alpha|^2.} \tag{4.4}$$

式 (4.3) より，

$$|x_n-\alpha|\leq\lambda^n|x_0-\alpha|=\frac{1}{10^{n\log_{10}(1/\lambda)-\log_{10}|x_0-\alpha|}}$$

ですから，単純に 1 ステップあたり「$\log_{10}\dfrac{1}{\lambda}$」桁ぐらい精度が上がることがわかります．また，後述する証明からわかるように，この λ はだいたい $\dfrac{m-1}{m}$ に近い値をとる

[*2] 二分法は解の存在まで保証してくれますが，ニュートン法は「解の存在を仮定した上で」成立するアルゴリズムです．実用上は，グラフを描画したり，二分法で解 α の存在をチェックし，位置をある程度特定してからニュートン法を用いるとよいでしょう．

[*3] ここで，$o((x-\alpha)^m)$ は**ランダウの記号**で，$\dfrac{|f(x)-g(x)|}{(x-\alpha)^m}\to 0$ $(x\to\alpha)$ が成り立つとき $f(x)=g(x)+o((x-\alpha)^m)$ と書き表すのでした．すなわち，$f(x)$ と $g(x)$ の誤差が相対的に $(x-\alpha)^m$ よりも早く 0 に近づくことを意味しています．

ことができます[*4].

重要かつ一般的なのは，α が重解ではない $m=1$ の場合です．このとき式 (4.4) より，$|x_n-\alpha|<\dfrac{1}{10^M}$ であれば

$$|x_{n+1}-\alpha|<K\frac{1}{(10^M)^2}=\frac{1}{10^{2M-\log_{10}K}}.$$

大雑把にいって，x_{n+1} は x_n と比べて解と一致する小数点以下の桁数が 2 倍近くになる，という優れたアルゴリズムです[*5].

例 1（平方根）　$A>0$ に対し $\sqrt{A}$ を計算してみましょう．$\sqrt{A}$ は関数 $f(x)=x^2-A$ の（重複度 1 の）零点ですから，$f'(x)=2x$ より漸化式 (5.3) は

$$x_{n+1}=x_n-\frac{x_n^2-A}{2x_n}=\frac{1}{2}\left(x_n+\frac{A}{x_n}\right)$$

となります．たとえば $A=2$ のとき，$x_0=2$ としてニュートン法のアルゴリズムを適用すると，

$$x_0=2,\ x_1=\frac{3}{2},\ x_2=\frac{17}{12},\ x_3=\frac{577}{408},\ x_4=\frac{665857}{470832},\ \cdots.$$

これでは近似の様子が見えてきこないので，数値化して表にすると，次のようになります：

n	近似値 x_n	誤差 $\lvert x_n-\sqrt{2}\rvert$
0	2.0000000000000000000	5.86×10^{-1}
1	1.5000000000000000000	8.58×10^{-2}
2	1.4166666666666666667	2.45×10^{-3}
3	1.4142156862745098039	2.12×10^{-6}
4	1.4142135623746899106	1.59×10^{-12}
5	1.4142135623730950488	8.99×10^{-25}
6	1.4142135623730950488	2.86×10^{-49}

定理 4.1 の式 (4.4) が示すとおりに，誤差は 1 ステップごとに，ほとんど 2 乗される勢いで減少しています．

【証明（定理 4.1，ニュートン法の収束性）】関数 $f(x)$ に対し，$N(x):=x-\dfrac{f(x)}{f'(x)}$ で定まる関数を考えよう．このとき，漸化式 (5.3) は $x_{n+1}=N(x_n)$ と表されることに注意する．

[*4] 実際，$|x_{n+1}-\alpha|/|x_n-\alpha|\to(m-1)/m$ $(n\to\infty)$ がわかります．
[*5] 実際，$|x_{n+1}-\alpha|/|x_n-\alpha|^2\to f''(a)/(2f'(a))$ $(n\to\infty)$ がわかります．

α を関数 $f(x)$ の重複度 m の零点とし，そこでの漸近展開が

$$f(x) = A(x-\alpha)^m + B(x-\alpha)^{m+1} + o((x-\alpha)^{m+1}) \tag{4.5}$$

(ただし $A = f^{(m)}(\alpha)/m! \neq 0$, $B = f^{(m+1)}(\alpha)/(m+1)!$) と表されると仮定しよう．このとき関数 $N(x)$ の α での漸近展開は次のようになる（演習問題）．

$$N(x) = \alpha + \frac{m-1}{m}(x-\alpha) + \frac{B}{m^2 A}(x-\alpha)^2 + o((x-\alpha)^2). \tag{4.6}$$

よって

$$N(x) - \alpha = \left\{ \frac{m-1}{m} + \underline{\frac{B}{m^2 A}(x-\alpha) + o(x-\alpha)} \right\} \cdot (x-\alpha).$$

ここで下線部は $x \to \alpha$ のとき 0 に収束するから，x が十分に α に近ければ中括弧の中身はいくらでも $\frac{m-1}{m}$ に近くなる．たとえば，$\frac{m-1}{m} < \lambda < 1$ となる定数 λ を自由に選んで，「$|x-\alpha| \leq \delta$ であれば中括弧の中身の絶対値が λ 以下」となるように十分に小さな δ を選ぶことができる．このとき $|N(x)-\alpha| \leq \lambda |x-\alpha|$ が成り立つから，ニュートン法の初期値 x_0 が $|x_0-\alpha| \leq \delta$ を満たせば，$|x_1-\alpha| = |N(x_0)-\alpha| \leq \lambda |x_0-\alpha|$．さらに $|x_2-\alpha| = |N(x_1)-\alpha| \leq \lambda |x_1-\alpha| \leq \lambda^2 |x_0-\alpha|$ などを得る．あとは帰納的に，式 (4.2) と式 (4.3) が導かれる．

$m = 1$ のときは，式 (4.6) より

$$N(x) - \alpha = \left\{ \frac{B}{A} + o(1) \right\} \cdot (x-\alpha)^2.$$

ここで $o(1)$ とは，$x \to \alpha$ のとき 0 に収束するという意味である．たとえば，「$|x-\alpha| \leq \delta$ のときこの $o(1)$ 部分の絶対値は 1 以下」となるように十分小さな $\delta > 0$ を選ぶと，三角不等式により $|B/A + o(1)| \leq |B/A| + |o(1)| \leq |B/A| + 1$．よって $K := |B/A| + 1$ とおけば，$|x-\alpha| \leq \delta$ のとき $|N(x)-\alpha| \leq K|x-\alpha|^2$ が成り立つ．必要ならば $K\delta \leq 1$ となるように δ を小さくとり直すことにすれば，$|N(x)-\alpha| \leq K|x-\alpha|^2 \leq K\delta^2 \leq \delta$．よってニュートン法の初期値 x_0 を $|x_0-\alpha| \leq \delta$ となるように選べば，帰納的に $|x_{n+1}-\alpha| = |N(x_n)-\alpha| \leq K|x_n-\alpha|^2$ を得る．■

4.2 Julia による実装

$\sqrt{2}$ の計算 例 1 の計算をそのまま実装し，x_{10} を出力するプログラムです．

練習 4.1

```
1  x = 2.0                    # 初期値の設定，x_0 に相当
2  for i = 1:10
3      x = (x + 2/x)/2        # x_i の計算に相当
```

```
4 end
5 println(x)                    # x_10 の値を書き出し
```

実行するとターミナルに

```
1.414213562373095
```

という結果が出力されるはずです．ちなみに $\sqrt{2} = 1.41421356237309504880168872\cdots$ ですから正しく計算できています．

$\sqrt{2}$ の計算：その 2　上のコードでは途中どういう計算を経て結果の出力が出てきたのかがわからず面白くないので，次のように変更してみましょう．

問題 4.1 練習 4.1 の 5 行目を削除し，どこかに

```
    println("Step 0: $x")
```

と

```
    println("Step $i: $x")
```

という 2 つの行を挿入する．それぞれをどこに入れれば，次のような実行結果が出せるか？

```
Step 0: 2.0
Step 1: 1.5
Step 2: 1.4166666666666665
Step 3: 1.4142156862745097
Step 4: 1.4142135623746899
Step 5: 1.414213562373095
Step 6: 1.414213562373095
Step 7: 1.414213562373095
Step 8: 1.414213562373095
Step 9: 1.414213562373095
Step 10: 1.414213562373095
```

ニュートン法の収束はものすごく速いので，倍精度（Float64）の実数を扱う限り，x_5 以降は結果が改善されません．そこで，任意精度を扱える BigFloat 型を利用してみましょう．

問題 4.2 上の問題 4.1 での修正に加え，練習 4.1 の 1 行目 `x = 2.0` を

```
setprecision(BigFloat, 128)
x = BigFloat(2.0)
```

の 2 行に置き換えて実行せよ．次のような結果が出るはずである：

```
Step 0: 2.0
Step 1: 1.5
Step 2: 1.41666666666666666666666666666666666666666671
Step 3: 1.41421568627450980392156862745098039215
Step 4: 1.41421356237468991062629557889013491011
Step 5: 1.41421356237309504880168962350253024361
Step 6: 1.41421356237309504880168872420969807856
Step 7: 1.41421356237309504880168872420969807856
Step 8: 1.41421356237309504880168872420969807856
Step 9: 1.41421356237309504880168872420969807856
```

最初の `setprecision(BigFloat, 128)` というのは，「以下，`BigFloat` 型として 128 ビットの浮動小数点数を考えよ」という命令です．128 をより大きな数値に変えることもできます．256 や 1024 に変えて，結果を比べてみてください．Newton 法の凄さがわかると思います．なお，`setprecision(BigFloat, 128)` とせずに単に `BigFloat(2.0)` とすると，`2.0` は `BigFloat` 型のデフォルト値である 256 ビットの浮動小数点数として計算されます．

計算コストを抑える 無駄な計算コストを抑えるために，途中で計算をストップさせるプログラムを書く問題を考えてみます：

問題 4.3 必要な精度が有効数字 6 桁だったと仮定する．このとき，必要以上の計算を避けるため，$|x_{n+1} - x_n| < 10^{-5}$ となった時点で計算を終了するようにプログラムを修正せよ．（Hint: `if` 文と `break` を使う．）

平方根の計算 次に，プログラムの汎用性を高めていきましょう．具体的には以下のような変更を加えます：

- 任意の $A > 0$ の平方根を求める．
- 初期値 x_0 を選べるようにする．
- 反復回数 `n` を自由に変えられるようにする．

- アルゴリズム全体を関数化する.

練習 4.2

```
function newton(A, x, n)      # 関数の定義はじめ
    println("Step 0: $x")     # 初期値を表示
    for i = 1:n
        x = (x + A/x)/2
        println("Step $i: $x")
    end
end                           # 関数の定義おわり
newton(3.0, 2.0, 5)           # A=3, x=2, n=5 の例を出力
```

いろいろと値を変えて実験してみてください. しかし, 値によっては正しく動作しない場合もあります:

問題 4.4 `newton(3.0, 0.0, 5)` を実行し, なぜそのようなその結果が出力されたか考えよ.

問題 4.5 (精度を上げる) `setprecision` と `BigFloat` を用いて任意の精度で平方根を計算できるように `newton` 関数を改良せよ.

問題 4.6 与えられた $A > 0$ と $k \in \mathbb{N}$ に対し, $\sqrt[k]{A}$ をニュートン法で計算する関数を作成せよ.

問題 4.7 前回紹介した円周率 π の定義

「$\cos x = 0$ となる最小の正の数の 2 倍」

とニュートン法のアルゴリズム用いて, 円周率を求めよ.

第5講　方程式の数値解法 3：複素数と DKW 法

Julia の良いところは複素数が自由に扱えるところです．今回はニュートン法やその類似を複素数の範疇で考えてみます．

5.1　代数方程式の数値解法

以下では，複素係数の n 次方程式

$$f(z) = A_n z^n + A_{n-1} z^{n-1} + \cdots + A_1 z + A_0 = 0 \quad (A_n \neq 0) \tag{5.1}$$

を考えます．大数学者ガウスが 1799 年に証明した**代数学の基本定理**によれば，複素係数の n 次方程式は重複度込みで n 個の解を持つことがわかります．すなわち，ある複素数 $\alpha_1, \cdots, \alpha_n$ が存在し，

$$f(z) = A_n (z - \alpha_1)(z - \alpha_2) \cdots (z - \alpha_n)$$

と表すことができるのです．以下では，式 (5.1) で与えられる方程式 $f(z) = 0$ に対し，上のような $\alpha_1, \cdots, \alpha_n$ を任意の精度で計算することを考えましょう．

5.2　複素ニュートン法

前回学んだニュートン法のアイディア（テイラー展開を使った式の導出）や定理 4.1 の証明は，複素数でも適用できます．そこで，一般に次のようなアルゴリズムが考えられます：

複素ニュートン法 (Newton's Method in Complex Variables).

(1) 解の近くにあると思われる初期値 z_0 を選ぶ．
(2) 漸化式

$$\boldsymbol{z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}} \quad (n = 0, 1, 2, \cdots) \tag{5.2}$$

で定まる数列を計算する．

(3) $|z_{n+1} - z_n|$ の値が一定以下になったところで計算を打ち切り，z_{n+1} を α の近似値として採用する．

このとき，定理 4.1 とまったく同じ定理（ただし，x_n は z_n に置き換え）が成立します．

例 1（1 のべき根） 1 の 3 乗根を数値計算してみましょう．対応する方程式は $f(z) = z^3 - 1 = 0$ です．高校では

$$z^3 - 1 = (z-1)(z^2+z+1) = 0 \iff z = 1 \text{ または} \frac{-1 \pm \sqrt{3}\,i}{2}$$

と因数分解をして解を求めました．一方，$f'(z) = 3z^2$ より，ニュートン法の漸化式 (5.3) は

$$z_{n+1} = z_n - \frac{z_n^3 - 1}{3z_n^2} = \frac{2z^3 + 1}{3z^2}$$

となりますから，適切な初期値を選ぶことで上の 3 つの解が正しく得られるかどうかが問題です．

早速 Julia でコーディングしてみましょう：

練習 5.1

```
z = im                          # 初期値の設定，im は虚数単位
println("Step 0: $z")
for k = 1:10
    z = (2*z^3 + 1)/(3*z^2)     # 近似値の更新
    println("Step $k: $z")
end
```

これまでは for 文のインデックスは i を用いていましたが，虚数単位 i と紛らわしいので k に変えました．次のような実行結果がでたでしょうか：

```
Step 0: im
Step 1: -0.3333333333333333 + 0.6666666666666666im
Step 2: -0.5822222222222223 + 0.9244444444444446im
Step 3: -0.5087908032893192 + 0.8681655118873493im
Step 4: -0.5000687390673926 + 0.8659822186925402im
Step 5: -0.4999999962890297 + 0.8660253983385867im
```

```
Step 6: -0.5 + 0.8660254037844386im
Step 7: -0.4999999999999999 + 0.8660254037844386im
Step 8: -0.5 + 0.8660254037844386im
Step 9: -0.4999999999999999 + 0.8660254037844386im
Step 10: -0.5 + 0.8660254037844386im
```

問題 5.1 練習 5.1 の初期値をいろいろ変えて，方程式 $z^3 - 1 = 0$ の 3 つの解を数値計算せよ．

問題 5.2 虚数単位 i の平方根（$\alpha^2 = i$ となる複素数 α）をニュートン法により求めよ．（厳密な答えは $\alpha = \pm(1+i)/\sqrt{2}$．）

結果をファイルに記録する ここで，計算結果をファイルに出力する方法も学んでおきましょう．以下は，練習 5.1 の計算結果を `out.txt` というテキストファイルに記録するためのコードです．

練習 5.2

```
1 f = open("out.txt", "w")          # ファイルを作成/開く
2 z = im
3 println(f, "Step $k: $z")     # ファイルへ初期値を書き込み
4 for k=1:10
5     z = (2*z^3 + 1)/(3*z^2)
6     println(f, "Step $k: $z")     # ファイルへ z_k を書き込み
7 end
8 close(f)                          # ファイルを保存し閉じる
```

1 行目の `open` 関数では，引用符（`" "`）の中身の `out.txt` というファイルを作成し，「書き込み可能モード」（`"w"`）で開く命令です．それを，`f` という「変数」に代入（格納）します．もし，すでに `out.txt` という名前のファイルがある場合は，それを開いて `f` に代入します．3 行目と 6 行目では，これまではターミナルに書き出していた出力をファイル `f` に書き込んでいます．8 行目の `close` 関数は `f` を保存し閉じるための命令です．このように，ファイルを読み込むときは原則として `open` 関数と `close` 関数をペアにして使います．

open 関数で指定できる「モード」としては，以下のものがよく利用されます[*1]：

	モード名	ファイルがすでに存在	ファイルが存在しない
"r"	読込みモード	読み込み	エラー
"w"	書込みモード	上書き	新規作成 + 書き込み
"a"	追記モード	追記	新規作成 + 追記

問題 5.3 追記モードを利用し，問題 5.1 で見つけた 3 つの初期値がそれぞれ収束する様子をファイルに記録せよ．

ニュートン法のジュリア集合を描く 複素数の初期値 z_0 に対しニュートン法を適用したとき，解の近くに到達するのに必要な反復回数を数えて複素平面上にプロットしてみると，面白いフラクタル図形が現れます．まず，次のような関数を定義しましょう：

練習 5.3

```
function julia(z)          # 関数 julia の定義はじめ
    for k = 0:20
        w = (2*z^3 + 1)/(3*z^2)
        if abs(z-w) < 0.001      # 収束しつつあるなら
            return k             # k の値を返す
            break                # for 文を出る
        end
        z = w
    end
    return 20              # 20 回で解に収束しなかった
end                        # 関数 julia の定義おわり
```

問題 5.4 この関数 julia に様々な複素数を代入し，その意味を考えよ．

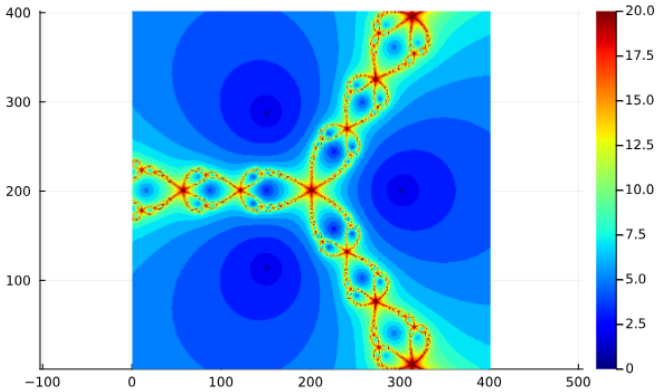
これを 2 次元の密度グラフ（heatmap）として表現してみます．まだ学んでいない関数がいろいろありますが（そのうち説明します），ひとまず無批判的に入力し実行しま

[*1] open 関数のデフォルトは読み込みモードです．つまり，モードを何も指定しない場合は読み込みモードとなります．本講義では（おそらく）読み込みモードは利用しません，

しょう：

練習 5.4

```
matrix = [julia(x + y*im)
            for y = -2.0:0.01:2.0,
                x = -2.0:0.01:2.0]
heatmap(matrix, aspect_ratio=:equal,
                                  color=cgrad(:jet1))
```



このような図が表示されたでしょうか[*2]．図を保存したければ 4 行目で

```
hm =heatmap(matrix, ...)
```

などとおき，そのあと

```
savefig(hm, "newton.png")
```

とすれば良いはずですが，利用している環境（設定）によってどこに newton.png というファイルが作成されるかが変わります．

このグラフから，複素ニュートン法には解に収束できない点が存在し，それは複雑なフラクタル集合をなしていることが示唆されます．このような集合をこのニュートン法が

[*2] 実際には複素平面の，実部と虚部がそれぞれ絶対値 2 以下の場所を描画しているので，このタテヨコの目盛りはおかしい（行列の成分が表示されている）．正しく表示するには練習 5.4 の部分を

```
xx = -2.0:0.01:2.0; yy = -2.0:0.01:2.0
matrix = [julia(x + y*im) for y = yy, x = xx]
heatmap(xx, yy, matrix, aspect_ratio=:equal, color = cgrad(:jet1))
```

とすればよい．

定める**ジュリア集合**といいます．このような集合の性質を記述するのが**複素力学系理論**とよばれる分野です．

5.3 DKW 法

式 (5.1) の解をすべて求めたい場合，

(1) ニュートン法で数値解 α_j を 1 つ見つけ，
(2) 次に多項式の割り算で $f(z) = (z - \alpha_j)g(z)$ を計算し，
(3) $g(z)$ に再びニュートン法を適用する

という作業を繰り返すことが考えられますが，これではどんどん誤差が蓄積してしまい，最初に得られた数値解と最後に得られた数値解では精度に差がでてしまいます．そこで，**すべての解を同じ精度で一斉に求める**方法を考えます．

アイディア まず，z が解 α_j $(j = 1, \cdots, n)$ に十分に近いとします．ニュートン法であれば，ここで

$$N_f(z) = z - \frac{f(z)}{f'(z)}$$

という関数を考えるのですが，$z \approx \alpha_j$ より $f'(z)$ を $f'(\alpha_j)$ で置き換えて

$$N_f(z) \approx z - \frac{f(z)}{f'(\alpha_j)}$$

としてよいでしょう．さらに

$$f'(\alpha_j) = (\alpha_j - \alpha_1)\cdots(\alpha_j - \alpha_{j-1})(\alpha_j - \alpha_{j+1})\cdots(\alpha_j - \alpha_n) = \prod_{j \neq l}(\alpha_j - \alpha_l)$$

であることから（理由を考えてみよ），**各 α_j $(j = 1, 2, \cdots, n)$ に対し近似値 z_j が与えられているとき**，

$$z_j^* := z_j - \frac{f(z_j)}{\prod_{j \neq l}(z_j - z_l)}$$

はニュートン法で得られる $N_f(z_j)$ の近似値を与えているものと期待されます．これを反復していくのが，次の **DKW 法** (デュラン・ケルナー・ワイエルシュトラス法) です：

DKW 法 (the Durand-Kerner-Weierstrass method).

(1) 解の近くにあると思われる初期値の組 $(z_1^{(0)}, \cdots, z_n^{(0)})$ を選ぶ．

(2) 漸化式

$$z_j^{(k+1)} = z_j^{(k)} - \frac{f(z_j^{(k)})}{\displaystyle\prod_{j \neq l} (z_j^{(k)} - z_l^{(k)})} \quad (k = 0, 1, 2, \cdots) \tag{5.3}$$

で定まる数列を計算する．

(3) 各 $|z_j^{(k+1)} - z_j^{(k)}|$ の値が一定以下になったところで計算を打ち切り，$z_j^{(k+1)}$ を α_j の近似値として採用する．

DKW 法はニュートン法と比べて，微分の計算が不要です．

例 2　例 1 と同様に，1 の 3 乗根を求めてみます．以下は，DKW 法の **“間違った”プログラム例です**．どこがおかしいのでしょうか？

練習 5.5

```
f(z) = z^3 - 1
z1 = 2; z2 = im; z3 = -im;
println("Step 0: ($z1, $z2, $z3)")
for k = 1:10
    z1 = z1 - f(z1)/((z1-z2)*(z1-z3))
    z2 = z2 - f(z2)/((z2-z1)*(z2-z3))
    z3 = z3 - f(z3)/((z3-z1)*(z3-z2))
    println("Step $k: ($z1, $z2, $z3)")
end
```

じつは，これでも正しい解たちに収束することが多く，しかもそのほうがより速く解に収束している可能性すらあります．

問題 5.5 練習 5.5 を修正し，下のような「正しい」出力が出るようにせよ．

```
Step 0: (2, im, 0 - 1im)
Step 1: (0.5999999999999999 + 0.0im, -0.3 + 1.1im, -0.3 - 1.1im)
Step 2: (0.9881188118811881 + 0.0im, -0.49405940594059405 + 0.8315031503150315im, -0.49405940594059405 - 0.8315031503150315im)
Step 3: (1.000313655329476 + 0.0im, -0.500156827664738 + 0.8669579360953118im, -0.500156827664738 - 0.8669579360953118im)
Step 4: (1.0000002179449836 + 0.0im, -0.5000001089724918 + 0.8660260513659012im, -0.5000001089724918 - 0.8660260513659012im)
Step 5: (1.0000000000001052 + 0.0im, -0.5000000000000526 + 0.8660254037847513im, -0.5000000000000526 - 0.8660254037847513im)
Step 6: (1.0 + 0.0im, -0.5 + 0.8660254037844387im, -0.5 - 0.8660254037844387im)
Step 7: (1.0 + 0.0im, -0.5 + 0.8660254037844386im, -0.5 - 0.8660254037844386im)
Step 8: (1.0 + 0.0im, -0.5 + 0.8660254037844387im, -0.5 - 0.8660254037844387im)
Step 9: (1.0 + 0.0im, -0.5 + 0.8660254037844386im, -0.5 - 0.8660254037844386im)
Step 10: (1.0 + 0.0im, -0.5 + 0.8660254037844387im, -0.5 - 0.8660254037844387im)
```

問題 5.6 与えられた 3 次関数 `f` と 3 つの初期値 `a`，`b`，`c` に対し，DKW 法を 10 回適用する関数を作成せよ．

第6講　連立 1 次方程式 1：ガウスの消去法

6.1　連立 1 次方程式

n 個の未知数 $x_1, x_2, \cdots, x_n$ に対し

$$m \text{ 個} \begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = b_2 \\ \quad\vdots \qquad\qquad \vdots & \quad\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n & = b_m \end{cases}$$

の形で与えられる n 元連立 1 次方程式は，

$$A := \left[a_{ij} \right]_{1 \le i \le m,\, 1 \le j \le n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \boldsymbol{x} := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \boldsymbol{b} := \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

とおくと，

$$A\boldsymbol{x} \;=\; \boldsymbol{b}$$

と表すことができます．左辺は $m \times n$ 型行列と $n \times 1$ 型行列の積であり，右辺は $m \times 1$ 型行列になっています．このとき，行列 A をこの連立 1 次方程式の**係数行列**といい，行列

$$\left[A \;\middle|\; \boldsymbol{b} \right] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}\right]$$

を**拡大係数行列**といいます．なお，以下では必要に応じ行列の成分 a_{ij} を $a_{i,j}$ と表すことがあります[*1]．

注意！　行列の角カッコや拡大係数行列の縦線は便宜的なものであり，無視してもかまいません．こうした未知数を含む方程式は「人間用」の表現であって，コンピューターに方程式を解かせ

[*1] たとえば，$a_{i,j+1}$ を a_{ij+1} とは書きたくないので．

る（そのためのアルゴリズムを作る）という観点では，データとしてこの拡大係数行列を入力し，答えとして n 次元のベクトル $\boldsymbol{x}$ を出力することにります．

6.2　ガウスの消去法（掃き出し法）

以下では，$m = n$ の場合を考えます．すなわち，A が n 次正方行列となる場合です．

ある n 次正方行列 B が A の**逆行列**であるとは，

$$AB = BA = E_n \text{ (単位行列)}$$

を満たすことをいい，逆行列が存在する n 次正方行列 A を，**正則（行列）**というのでした．1 年生で学んだ線形代数では，正則行列の特徴づけとして次の定理を学んだことかと思います：

定理 6.1 n 次正方行列 A に対し，以下は同値（互いに必要十分条件）：

(1) $\operatorname{rank} A = n$
(2) A は**行基本変形**により E_n となる．
(3) 任意の n 次元列ベクトル $\boldsymbol{b}$ に対し，連立 1 次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ はただ 1 つの解をもつ．
(4) A は逆行列をもつ．すなわち，A は正則行列．
(5) $\det A \neq 0$. ただし，$\det A$ は A の行列式．

詳しくはお手持ちの線形代数の教科書を見てください．今回重要なのは (2) の条件で，これからは本質的に**行基本変形**だけを繰り返して連立 1 次方程式を数値的に解いていきます．復習しましょう：

定義（行基本変形）. 行列 A に対する以下の 3 つの操作を**行基本変形**という：行列の第 i 行目を ⓘ と表すものと約束し，α を 0 でない定数とすると，

(R1) 第 i 行を α 倍する（ⓘ $\longrightarrow$ ⓘ $\times\, \alpha$）.
(R2) 第 i 行を α 倍し，第 i' 行に加える（ⓘ′ $\longrightarrow$ ⓘ′ $+$ ⓘ $\times\, \alpha$）.
(R3) 第 i 行と第 i' 行を入れ替える．（ⓘ $\longleftrightarrow$ ⓘ′）.

6.3　ガウスの消去法

与えられた連立方程式を解く実用的なアルゴリズムとしてよく知られているのが，**ガウスの消去法**（Gaussian elimination）もしくは**掃き出し法**とよばれるものです．

具体例として，連立方程式

$$(*)\quad \begin{cases} 2x & +2y & +z & =0 \\ 3x & -y & & =3 \\ -x & -3y & +2z & =-5 \end{cases}$$

をガウスの消去法で解いてみましょう．以下，対応する拡大係数行列と比較しながら式変形を進めます．その際，方程式 $(*)$ あるいは行列の第 i 行目を $\textcircled{i}$ と表すものと約束しましょう．また，変形「前」の注目ポイントは字を青く，変形「後」の注目ポイントは背景を青くしました．

初期状態：

$$(*)\quad \begin{cases} 2x & +2y & +z & =0 \\ 3x & -y & & =3 \\ -x & -3y & +2z & =-5 \end{cases} \iff \left[\begin{array}{ccc|c} 2 & 2 & 1 & 0 \\ 3 & -1 & 0 & 3 \\ -1 & -3 & 2 & -5 \end{array}\right].$$

$\xrightarrow{(\mathrm{R1})} \textcircled{1} \times \dfrac{1}{2}$ ：

（第 1 行を $1/2$ 倍する）

$$(*)\quad \begin{cases} x & +y & +\frac{1}{2}z & =0 \\ 3x & -y & & =3 \\ -x & -3y & +2z & =-5 \end{cases} \iff \left[\begin{array}{ccc|c} 1 & 1 & \frac{1}{2} & 0 \\ 3 & -1 & 0 & 3 \\ -1 & -3 & 2 & -5 \end{array}\right]$$

$\xrightarrow{(\mathrm{R2})} \textcircled{2} + \textcircled{1} \times (-3),\quad \textcircled{3} + \textcircled{1} \times 1$ ：

（第 2 行に第 1 行の (-3) 倍を加え，第 3 行に第 1 行の 1 倍を加える．以下同様）

$$(*)\quad \begin{cases} x & +y & +\frac{1}{2}z & =0 \\ 0 & -4y & -\frac{3}{2}z & =3 \\ 0 & -2y & +\frac{5}{2}z & =-5 \end{cases} \iff \left[\begin{array}{ccc|c} 1 & 1 & \frac{1}{2} & 0 \\ 0 & -4 & -\frac{3}{2} & 3 \\ 0 & -2 & \frac{5}{2} & -5 \end{array}\right]$$

$\xrightarrow{(\mathrm{R1})} \textcircled{2} \times \left(-\dfrac{1}{4}\right)$ ：

$$(*)\quad \begin{cases} x & +y & +\frac{1}{2}z & =0 \\ & y & +\frac{3}{8}z & =-\frac{3}{4} \\ & -2y & +\frac{5}{2}z & =-5 \end{cases} \iff \left[\begin{array}{ccc|c} 1 & 1 & \frac{1}{2} & 0 \\ 0 & 1 & \frac{3}{8} & -\frac{3}{4} \\ 0 & -2 & \frac{5}{2} & -5 \end{array}\right]$$

$\xrightarrow{\text{(R2)}}$ ③ + ② × 2：

$$(*)\ \begin{cases} x & +y & +\frac{1}{2}z & =0 \\ & y & +\frac{3}{8}z & =-\frac{3}{4} \\ & 0 & +\frac{13}{4}z & =-\frac{13}{2} \end{cases} \iff \left[\begin{array}{ccc|c} 1 & 1 & \frac{1}{2} & 0 \\ 0 & 1 & \frac{3}{8} & -\frac{3}{4} \\ 0 & 0 & \frac{13}{4} & -\frac{13}{2} \end{array}\right]$$

$\xrightarrow{\text{(R1)}}$ ③ × $\frac{4}{13}$：

$$(*)\ \begin{cases} x & +y & +\frac{1}{2}z & =0 \\ & y & +\frac{3}{8}z & =-\frac{3}{4} \\ & & z & =-2 \end{cases} \iff \left[\begin{array}{ccc|c} 1 & 1 & \frac{1}{2} & 0 \\ 0 & 1 & \frac{3}{8} & -\frac{3}{4} \\ 0 & 0 & 1 & -2 \end{array}\right]$$

以上の操作を**前進消去**といいます．また，以下の操作は**後退代入**といいます．

$\xrightarrow{\text{(R2)}}$ ① + ③ × $\left(-\frac{1}{2}\right)$，　② + ③ × $\left(-\frac{3}{8}\right)$：

$$(*)\ \begin{cases} x & +y & +0 & =1 \\ & y & +0 & =0 \\ & & z & =-2 \end{cases} \iff \left[\begin{array}{ccc|c} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 \end{array}\right]$$

$\xrightarrow{\text{(R2)}}$ ① + ② × (−1)：

$$(*)\ \begin{cases} x & +0 & & =1 \\ & y & & =0 \\ & & z & =-2 \end{cases} \iff \left[\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 \end{array}\right]$$

よって，方程式の解として $x=1, y=0, z=-2$ が得られました．とくに，拡大係数行列のほうは $\left[\, E_3 \mid \boldsymbol{c} \,\right]$ の形になっていることに注意しましょう．プログラミングにおいては，拡大係数行列のみをデータとして入力し，変形していけばよいことも想像がつきます．

一般化　以上の操作をアルゴリズムとして一般化してみましょう．与えられた拡大係数行列

$$\left[\, A \mid \boldsymbol{b} \,\right] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right]$$

に対し，行基本変形で前進消去と後退代入を施していきます．ただし，先の例をよく見るとわかるように，前進消去においては (R1) と (R2) の作業はペア行われていて無駄が多

いように思われるので，以下では慣例にしたがって (R1) の部分を省略したアルゴリズムに書き換えることにします．具体的には，

$$\left[\, A \mid \boldsymbol{b} \,\right] =: \left[\, A^{(1)} \mid \boldsymbol{b}^{(1)} \,\right] \xrightarrow{\text{(R2)}} \left[\, A^{(2)} \mid \boldsymbol{b}^{(2)} \,\right] \xrightarrow{\text{(R2)}} \cdots \xrightarrow{\text{(R2)}} \left[\, A^{(n)} \mid \boldsymbol{b}^{(n)} \,\right]$$

と行基本変形 (R2) をひたすら繰り返し，最後の $\left[\, A^{(n)} \mid \boldsymbol{b}^{(n)} \,\right]$ が

$$\left[\, A^{(n)} \mid \boldsymbol{b}^{(n)} \,\right] =: \left[\, U \mid \boldsymbol{c} \,\right] = \left[\begin{array}{cccc|c} u_{11} & u_{12} & \cdots & u_{1n} & c_1 \\ 0 & u_{22} & \cdots & u_{2n} & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & u_{nn} & c_n \end{array}\right]$$

(青い影の部分はすべて 0) の形になるように変形しましょう．なお，U のように，対角成分より下にある成分がすべて 0 となる行列を**上三角行列**といいます．

ガウスの消去法のアルゴリズム：前進消去

$$\left[\, A \mid \boldsymbol{b} \,\right] =: \left[\, A^{(1)} \mid \boldsymbol{b}^{(1)} \,\right] = \left[\begin{array}{cccc|c} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1}^{(1)} & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} & b_n^{(1)} \end{array}\right] \quad \text{（これが初期値）}$$

$\xrightarrow{\text{(R2)}}$ ⓘ $+$ ① $\times \left(-a_{i1}^{(1)} / a_{11}^{(1)}\right)$ $(i = 2, 3, \cdots, n)$：

$$\left[\, A^{(2)} \mid \boldsymbol{b}^{(2)} \,\right] = \left[\begin{array}{cccc|c} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ & \vdots & \ddots & \vdots & \vdots \\ & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} & b_n^{(2)} \end{array}\right]$$

ただし，
$$\begin{cases} a_{ij}^{(2)} = a_{ij}^{(1)} - a_{1j}^{(1)} \times a_{i1}^{(1)} / a_{11}^{(1)} \\ b_i^{(2)} = b_i^{(1)} - b_1^{(1)} \times a_{i1}^{(1)} / a_{11}^{(1)} \\ (i, j = 2, 3, \cdots, n) \end{cases}$$

$\xrightarrow{\text{(R2)}}$ ① $+$ ② $\times \left(-a_{i2}^{(2)}/a_{22}^{(2)}\right)$　$(i = 3,\, 4,\, \cdots,\, n)$：

$$\left[\, A^{(3)} \mid \boldsymbol{b}^{(3)} \,\right] = \left[\begin{array}{ccccc|c} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ & & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} & b_3^{(3)} \\ & & \vdots & \ddots & \vdots & \vdots \\ & & a_{n3}^{(3)} & \cdots & a_{nn}^{(3)} & b_n^{(3)} \end{array}\right]$$

$$\text{ただし} \begin{cases} a_{ij}^{(3)} = a_{ij}^{(2)} - a_{1j}^{(2)} \times a_{i2}^{(2)}/a_{22}^{(2)} \\ b_i^{(3)} = b_i^{(2)} - b_2^{(2)} \times a_{i2}^{(2)}/a_{22}^{(2)} \\ (i,\, j = 3,\, 4,\, \cdots,\, n) \end{cases}$$

$\xrightarrow{\text{(R2)}}$ $\cdots$ $\xrightarrow{\text{(R2)}}$

$$\left[\, A^{(n)} \mid \boldsymbol{b}^{(n)} \,\right] = \left[\begin{array}{ccccc|c} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1,n-1}^{(1)} & a_{1n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2,n-1}^{(2)} & a_{2n}^{(2)} & b_2^{(2)} \\ & & a_{33}^{(3)} & \cdots & a_{3,n-1}^{(3)} & a_{3n}^{(3)} & b_3^{(3)} \\ & & & \ddots & \vdots & \vdots & \vdots \\ & & & & a_{n-1,n-1}^{(n-1)} & a_{n-1,n}^{(n-1)} & b_{n-1}^{(n-1)} \\ & & & & & a_{nn}^{(n)} & b_n^{(n)} \end{array}\right].$$

ガウスの消去法のアルゴリズム：後退代入　上の拡大係数行列を

$$\left[\, U \mid \boldsymbol{c} \,\right] = \left[\begin{array}{ccccc|c} u_{11} & u_{12} & \cdots & u_{1,n-1} & u_{1n} & c_1 \\ & u_{22} & \cdots & u_{2,n-1} & u_{2n} & c_2 \\ & & \ddots & \vdots & \vdots & \vdots \\ & & & u_{n-1,n-1} & u_{n-1,n} & c_{n-1} \\ & & & & u_{nn} & c_n \end{array}\right].$$

とおくとき，これは連立方程式

$$\begin{array}{ccccc|c} u_{11}\,x_1 + & u_{12}\,x_2 + & \cdots & u_{1,n-1}\,x_{n-1} & u_{1n}\,x_n & = c_1 \\ & u_{22}\,x_2 + & \cdots & u_{2,n-1}\,x_{n-1} & u_{2n}\,x_n & = c_2 \\ & & \ddots & \vdots & \vdots & \vdots \\ & & & u_{n-1,n-1}\,x_{n-1} + & u_{n-1,n}\,x_n & = c_{n-1} \\ & & & & u_{nn}\,x_n & = c_n \end{array}$$

を意味するから，あとは

$$
\begin{aligned}
x_n &= \frac{c_n}{u_{nn}}, \\
x_{n-1} &= \frac{c_{n-1} - u_{n-1,n}\, x_n}{u_{n-1,n-1}}, \\
x_{n-2} &= \frac{c_{n-2} - u_{n-2,n}\, x_n - u_{n-2,n-1}\, x_{n-1}}{u_{n-2,n-2}}, \\
&\vdots \qquad\qquad\qquad \vdots \\
x_1 &= \frac{c_1 - u_{1,n}\, x_n - u_{1,n-1}\, x_{n-1} \cdots - u_{12}\, x_2}{u_{11}}.
\end{aligned}
$$

とすればよい．

注意！　このアルゴリズムはある k で $a_{kk}^{(k)} = 0$ となった時点で破綻してしまいます．そうでなくても $a_{kk}^{(k)}$ （これを**ピボット** (pivot) という）が極端に 0 に近いと (R2) を施す過程で係数が急激に大きくなり，誤差の原因となるでしょう．こうした問題を解消するために必要なのが**ピボット選択** (pivotting) とよばれる操作です．これについては次回考えることにしましょう．

6.4　Julia におけるベクトル・行列

ガウスの消去法を Julia 実装するにあたって，ベクトルや行列を Julia で扱う方法を学びます．久々に REPL で実行していきましょう．

縦ベクトル　5 次元の縦ベクトル $\begin{bmatrix} 1 \\ 2 \\ \vdots \\ 5 \end{bmatrix}$ を作成してみます：

練習 6.1

```
julia> [1, 2, 3, 4, 5]
julia> [1; 2; 3; 4; 5]
julia> [i for i = 1:5]
julia> [i for i in 1:5]
```

いずれも同じものを定義しています．これらは 5×1 行列でもあります．

横ベクトル　5 次元の横ベクトル $\begin{bmatrix} 1 & 2 & \cdots & 5 \end{bmatrix}$ を作成してみます：

練習 6.2

```
julia> [1 2 3 4 5]
julia> [i for i = 1:5]'
```

1 行目は半角空白 ␣ で区切っています．2 行目は右肩にプライム ’（ダッシュ）がついています．これは転置行列を与える命令で，練習 6.1 の 3 行目のように作った縦ベクトル（＝ 5 × 1 行列）を転置して横ベクトル（＝ 1 × 5 行列）にしているのです[*2]．

行列 行列はプログラミング言語では **2 次元配列**とよばれることもあります．行列 $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ を作成してみます．

練習 6.3

```
julia> A = [1 2 3; 4 5 6]
julia> [1 2 3
          4 5 6]
julia> [3*i+j for i = 0:1, j = 1:3]
julia> [3*i+j for i in 0:1, j in 1:3]
```

いずれも同じ 2 × 3 行列 A と同じものを定義しています．1 行目はセミコロン ; が改行を意味しており，2 行目はコード上の改行そのものが行列の改行を意味しています．4 行目と 5 行目は for 文を用いた方法で，巨大な行列を作成するのに適した方法です．

次に，行列 A の要素を取り出します．行列 (i,j) 成分を取り出すには A[i,j] などとします．

練習 6.4

```
julia> A[1,2]          #(1,2) 成分の取り出し
julia> A[2,3]          #(2,3) 成分の取り出し
julia> A[2,3] = 777    #(2,3) 成分のみ値を 777 に変更
julia> A               # A の値の確認
```

[*2] 正確には，’ は転置行列にさらに各成分の複素共役を取った「随伴行列」（エルミート共役）とよばれるものを与えますが，この場合は実数成分なのでただの転置行列と一致します．複素共役をとらないただの転置は verb—transpose— 関数を使います．練習 6.5 も参照してください

```
julia> A[1, :]        # A の 1 行目からなる縦ベクトル
julia> A[:, 1]        # A の 2 列目からなる縦ベクトル
julia> size(A)        # A が m × n 行列なら (m, n) を返す
julia> size(A,1)      # A が m × n 行列なら m を返す
julia> size(A,2)      # A が m × n 行列なら n を返す
```

問題 6.1 以下の行列（2 次元配列）を作成せよ：

(1) $\begin{bmatrix} 1 & 1^2 & 1^3 & 1^4 & 1^5 \\ 3 & 3^2 & 3^3 & 3^4 & 3^5 \\ 5 & 5^2 & 5^3 & 5^4 & 5^5 \\ 7 & 7^2 & 7^3 & 7^4 & 7^5 \\ 9 & 9^2 & 9^3 & 9^4 & 9^5 \end{bmatrix}$ (2) $\begin{bmatrix} \sin 1^\circ & \sin 2^\circ & \cdots & \sin 90^\circ \\ \cos 1^\circ & \cos 2^\circ & \cdots & \cos 90^\circ \end{bmatrix}$

その上で，$\cos 37^\circ$ の値を (2) の行列の値として参照せよ．

行列の操作

練習 6.5

```
julia> A = [1 2; 3 4];  # 行列の定義（上書き）
julia> B = 2*A          # A の各成分を 2 倍し B とする
julia> A + B            # A と B の和
julia> A - B            # A と B の差
julia> A*B              # A と B の積
julia> A^2              # A の 2 乗，A*A と同じ
julia> A'               # A の転置（随伴行列）
julia> transpose(A)     # A の（ふつうの）転置
```

5 行目の行列の積は，A の列と B の行の数が一致しているときに限って定義できます．7 行目の「随伴行列」とは，A の (i,j) 成分が a_{ij} で与えられているとき，(i,j) 成分が $\overline{a_{ji}}$ （a_{ji} の複素共役 ）で与えられるような行列をいいます．8 行目の「（ふつうの）転置」とは，同じく A の (i,j) 成分が a_{ij} で与えられているとき，(i,j) 成分が a_{ji} で与えられるような行列をいいます．したがって，行列の成分が実数であれば随伴行列と転置

行列はおなじものです.

次回 次回はいよいよガウスの消去法を Julia で実装してみましょう.

第7講　連立 1 次方程式 2：ガウスの消去法 (2)

7.1　連立 1 次方程式（前回の設定の復習）

前回にひきつづき，n 個の未知数 $x_1, x_2, \cdots, x_n$ に対し，

$$A := \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \boldsymbol{x} := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \boldsymbol{b} := \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

とおき，

$$A\boldsymbol{x} \;=\; \boldsymbol{b}$$

と表される連立 1 次方程式を考えます．行列 A をこの連立 1 次方程式の**係数行列**といい，行列

$$\left[\, A \;\middle|\; \boldsymbol{b} \,\right] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right]$$

を**拡大係数行列**というのでした．以下，A は正則行列であり，したがって $\boldsymbol{x} = A^{-1}\boldsymbol{b}$ の形で表される解をもつものとします．このとき，$\boldsymbol{x}$ の値を数値計算するプログラムを書くのが今回の目標です．具体的には，前回定式化した**ガウスの消去法**を Julia で実装します．

7.2　ガウスの消去法への準備：行基本変形の関数

ガウスの消去法は，以下の**行基本変形**を繰り返して連立 1 次方程式の解を与えるアルゴリズムでした：

定義（行基本変形）. 行列 A に対する以下の 3 つの操作を**行基本変形**という：行列の第 i 行目を ⓘ と表すものと約束し，α を 0 でない定数とすると，

(R1) 第 i 行を α 倍する（$\textcircled{i} \longrightarrow \textcircled{i} \times \alpha$）.
(R2) 第 i 行を α 倍し，第 i' 行に加える（$\textcircled{i'} \longrightarrow \textcircled{i'} + \textcircled{i} \times \alpha$）.
(R3) 第 i 行と第 i' 行を入れ替える．（$\textcircled{i} \longleftrightarrow \textcircled{i'}$）.

行基本変形自体は（正方行列とは限らない）任意の行列に対して考えることができるので，与えられた行列 A に対し，行基本変形を施す関数を Julia で実現しましょう．なお，定数倍する α の代わりに文字 s を使うことにします．

まずは (R1) です：

練習 7.1

```
function R1(A, i, s)      # 関数の定義はじめ
    n = size(A,2)         # 列の数を n とする
    for j = 1:n           # [i, j] 成分への操作
        A[i, j] = A[i, j]*s
    end
    return A              # 値として変形後の A を返す
end                       # 関数の定義おわり
# 具体例
A = [1 2; 3 4; 5 6]
R1(A, 3, 100)
```

問題 7.1 行列 A の i 行目に k 行目の s 倍を加える関数 R2(A, i, k, s) を作成せよ.

問題 7.2 行列 A の i 行目と k 行目を入れ替える関数 R3(A, i, k) を作成せよ.

7.3　ガウスの消去法のアルゴリズム：前進消去

与えられた拡大係数行列

$$\left[\, A \mid \boldsymbol{b} \,\right] =: \left[\, A^{(1)} \mid \boldsymbol{b}^{(1)} \,\right] = \left[\begin{array}{cccc|c} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1}^{(1)} & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} & b_n^{(1)} \end{array}\right]$$

に対し，

$$\left[\, A \mid \boldsymbol{b} \,\right] =: \left[\, A^{(1)} \mid \boldsymbol{b}^{(1)} \,\right] \xrightarrow{\text{(R2)}} \left[\, A^{(2)} \mid \boldsymbol{b}^{(2)} \,\right] \xrightarrow{\text{(R2)}} \quad \cdots \quad \xrightarrow{\text{(R2)}} \left[\, A^{(n)} \mid \boldsymbol{b}^{(n)} \,\right]$$

と行基本変形 (R2) をひたすら繰り返し，最後の $\left[\, A^{(n)} \mid \boldsymbol{b}^{(n)} \,\right]$ が

$$\left[\, A^{(n)} \mid \boldsymbol{b}^{(n)} \,\right] =: \left[\, U \mid \boldsymbol{c} \,\right] = \left[\begin{array}{cccc|c} u_{11} & u_{12} & \cdots & u_{1n} & c_1 \\ 0 & u_{22} & \cdots & u_{2n} & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & u_{nn} & c_n \end{array}\right]$$

(青い影の部分はすべて 0) の形になるように変形します．U のように，対角成分より下にある成分がすべて 0 となる行列を**上三角行列**というのでした．

いま，$k = 1, 2, \cdots, n-1$ に対して，$\left[\, A^{(k)} \mid \boldsymbol{b}^{(k)} \,\right]$ から $\left[\, A^{(k+1)} \mid \boldsymbol{b}^{(k+1)} \,\right]$ への変形は次で与えられます（灰色の網掛け部分は 0 にしたい部分）：

$$\left[\, A^{(k)} \mid \boldsymbol{b}^{(k)} \,\right] = \left[\begin{array}{ccccccc|c} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1k}^{(1)} & a_{1,k+1}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2k}^{(2)} & a_{2,k+1}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ & & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ & & & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{kn}^{(k)} & b_k^{(k)} \\ & & & a_{k+1,k}^{(k)} & a_{k+1,k+1}^{(k)} & \cdots & a_{k+1,n}^{(k)} & b_{k+1}^{(k)} \\ & & & \vdots & \vdots & \ddots & \vdots & \vdots \\ & & & a_{nk}^{(k)} & a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k)} & b_n^{(k)} \end{array}\right]$$

$$\textcircled{i} \xrightarrow{\text{(R2)}} \textcircled{i} + \textcircled{k} \times \left(-a_{ik}^{(k)}/a_{kk}^{(k)}\right) \quad (i = k+1, k+2, \cdots, n)\text{：}$$

$$\left[\, A^{(k+1)} \mid \boldsymbol{b}^{(k+1)} \,\right] = \left[\begin{array}{ccccccc|c}
a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1k}^{(1)} & a_{1,k+1}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\
 & a_{22}^{(2)} & \cdots & a_{2k}^{(2)} & a_{2,k+1}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\
 & & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\
 & & & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{kn}^{(k)} & b_k^{(k)} \\
 & & & & a_{k+1,k+1}^{(k+1)} & \cdots & a_{k+1,n}^{(k+1)} & b_{k+1}^{(k+1)} \\
 & & & & \vdots & \ddots & \vdots & \vdots \\
 & & & & a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k+1)} & b_n^{(k+1)}
\end{array}\right]$$

これを Julia で実装したいわけですが，拡大係数行列をまとめて 1 つの行列として扱いたいので，プログラムとしては

$$a_{i,n+1}^{(k)} := b_i^{(k)} \quad (i, k = 1, 2, \cdots, n)$$

と考えて，最終的に

$$\left[\, A^{(n)} \mid \boldsymbol{b}^{(n)} \,\right] = \left[\begin{array}{ccccccc|c}
a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1,n-1}^{(1)} & a_{1n}^{(1)} & a_{1,n+1}^{(1)} \\
 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2,n-1}^{(2)} & a_{2n}^{(2)} & a_{2,n+1}^{(2)} \\
 & & a_{23}^{(3)} & \cdots & a_{3,n-1}^{(3)} & a_{3n}^{(3)} & a_{3,n+1}^{(3)} \\
 & & & \ddots & \vdots & \vdots & \vdots \\
 & & & & a_{n-1,n-1}^{(n-1)} & a_{n-1,n}^{(n-1)} & a_{n-1,n+1}^{(n-1)} \\
 & & & & & a_{nn}^{(n)} & a_{n,n+1}^{(n)}
\end{array}\right] \tag{7.1}$$

の形の上三角行列に変形することを目指します．

次は，与えられた正方行列 `A` と列ベクトル `b` に対し，前進消去をほどこした拡大係数行列を返す関数です：

練習 7.2

```
# 前進消去関数
function forwardElimination(A, b)
    n = size(A,1)          # 行のサイズ
    Ab = hcat(A, b)        # A の右手に b を結合し Ab とおく
    for k = 1:n-1
        for i = k+1:n
```

```
            s = -Ab[i, k]/Ab[k, k]
            Ab = R2(Ab, i, k, s)
        end
    end
    return Ab              # 拡大係数行列を返す
end
# 具体例
A = [2 2 1; 3 -1 0; -1 -3 2.0]
b = [0; 3; -5.0]
forwardElimination(A,b)
```

4 行目は行列 A の右手に b を結合し拡大係数行列 Ab を作成するために，hcat(A, b) という組み込み関数を用いています[*1]．また，14-15 行目で行列やベクトルの成分を Float64 型と認識させるために 2 をわざと 2.0 と書いたりしています[*2]．あとは，ほとんど上の説明そのままのコードですから解読可能かと思います．

7.4　ガウスの消去法のアルゴリズム：後退代入

前回学んだように，前進消去によって得られた拡大係数行列を

$$\left[U \mid \boldsymbol{c}\right] = \left[\begin{array}{ccccc|c} u_{11} & u_{12} & \cdots & u_{1,n-1} & u_{1n} & c_1 \\ & u_{22} & \cdots & u_{2,n-1} & u_{2n} & c_2 \\ & & \ddots & \vdots & \vdots & \vdots \\ & & & u_{n-1,n-1} & u_{n-1,n} & c_{n-1} \\ & & & & u_{nn} & c_n \end{array}\right]$$

[*1] [A b] と書いても同じものが得られます．これらは A と b の行のサイズが同じ場合にだけ適用できて，それ以外ではエラーとなります．

[*2] 本当は面倒くさがらずに A = [2.0 2.0 1.0; 3.0 -1.0 0.0; -1.0 -3.0 2.0] とするべきでしょう．

とおくとき，もとの連立方程式の解は x_n, x_{n-1}, $\cdots$, x_1 の順で

$$
\begin{aligned}
x_n &= \frac{c_n}{u_{nn}}, \\
x_{n-1} &= \frac{c_{n-1} - u_{n-1,n}\, x_n}{u_{n-1,n-1}}, \\
x_{n-2} &= \frac{c_{n-2} - u_{n-2,n}\, x_n - u_{n-2,n-1}\, x_{n-1}}{u_{n-2,n-2}}, \\
&\vdots \qquad\qquad \vdots \\
x_1 &= \frac{c_1 - u_{1,n}\, x_n - u_{1,n-1}\, x_{n-1} \cdots - u_{12}\, x_2}{u_{11}}.
\end{aligned}
$$

と与えられるのでした．これと 式 (7.1) を見比べつつ，前進消去の関数と合わせれば，次の「ガウスの消去法を実行する関数」が構成できます：

練習 7.3

```
# ガウスの消去法
function GaussianElimination(A, b)
    n = size(A,1)              # A のサイズ
    Uc = forwardElimination(A,b) # 前進消去した Ab
    x = zeros(n)         # 解の列ベクトル，値はダミー
    for k = n:-1:1             # k は n から 1 へ減らす
        y = Uc[k, n+1]         # c の第 k 成分のこと
        for i = k+1:n
            y = y - Uc[k, i] * x[i];
        end
        x[k] = y / Uc[k,k];
    end
    return x         # 値として x を返す
end
# 具体例
A = [2 2 1; 3 -1 0; -1 -3 2.0]
b = [0; 3; -5.0]
x = GaussianElimination(A,b)
```

```
19  A*x                    # 検算．ちゃんと b になるか？
```

5 行目の zeros(n) は n 次元のゼロ（列）ベクトルを作成する関数です[*3]．ここでは x = zeros(n) という命令で解となるベクトル x をとりあえずダミーの値で作成し，6 行目からの for 文で最終的な値を決定していきます．

問題 7.3 次の 4 元連立方程式を GaussianElimination で解け．

$$x+y=40,\quad y+z=45,\quad x+z=25,\quad z+w=35$$

答えは $(x,y,z,w)=(10,30,15,20)$.

7.5　ピボット選択

問題 7.4 (ピボット選択) A = [0 2 1; 0 -1 1; -1 0 0.0], b = [0; 3; -5] に対し forwardElimination(A,b) を実行すると成分に NaN (Not a Number) という文字列が出てきてしまう．理由を考えよ．

問題 7.5 (ピボット選択) 問題 7-3 の 4 元連立方程式を並べ替えて

$$x+y=40,\quad z+w=35,\quad x+z=25,\quad y+z=45$$

として GaussianElimination で解くと，成分に NaN が出てきてしまう．理由を考えよ．

前回の注意でも述べたように，前進消去のアルゴリズムは，ある k で $a_{kk}^{(k)}=0$ となった時点で破綻してしまいます．そうでなくても，$a_{kk}^{(k)}$ が非常に 0 に近い値だと係数が発散してしまい，計算誤差が蓄積してしまいます．そのため，通常は以下の**ピボット選択** (pivotting) とよばれる操作をほどこして，そのような不都合を回避します．

ピボット選択． $\left[A^{(k)}\mid \boldsymbol{b}^{(k)}\right]$ から $\left[A^{(k+1)}\mid \boldsymbol{b}^{(k+1)}\right]$ への (R2) による変形をする前に，

$$a_{k,k}^{(k)},\ a_{k+1,k}^{(k)},\ \cdots,\ a_{n,k}^{(k)}$$

[*3] 同様に，zeros(m, n) は m 行 n 列のゼロ行列を作成する関数です．

の中で絶対値が最大となる $a_{p,k}^{(k)}$ $(k \leq p \leq n)$ を見つけて，k 行目と p 行目を入れ替えておく．すなわち，(R3) ⓚ $\longleftrightarrow$ ⓟ を行う．

入れ替えたあとの $a_{k,k}^{(k)} \neq 0$ を**ピボット**といい，その後の (R2) による変形に用いる係数がこの値によって定まります[*4]．

問題 7.6 (1) 関数 `forwardElimination(A,b)` にピボット選択を組み込んだ `forwardEliminationPivotting(A,b)` を作成し，問題 7.4，問題 7.5 の例でもうまくいくことを確認せよ．

(2) 関数 `GaussianElimination(A,b)` に `forwardEliminationPivotting(A,b)` を用いた `GaussianEliminationPivotting(A,b)` を作成し，問題 7.4，問題 7.5 の例に対し解を与えよ．

[*4] もしピボットが見つからなければ，$a_{k,k}^{(k)} = a_{k+1,k}^{(k)} == a_{n,k}^{(k)} = 0$ ということであり，そもそもその行列は正則ではなかったことになります（理由を考えてみてください）．

第8講　連立1次方程式 3：LU 分解

8.1　LU 分解のアイディア

前々回・前回にひきつづき，A を正則な n 次正方行列，$\boldsymbol{b}$ を n 次元縦ベクトルとするとき，A を係数行列とする連立1次方程式

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

を解くことを考えましょう．たとえば

$$\boldsymbol{e}_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \boldsymbol{e}_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \cdots, \quad \boldsymbol{e}_n = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

を $\mathbb{R}^n$ の標準基底とするとき，各 j $(1 \leq j \leq n)$ に対する方程式

$$A\boldsymbol{x} = \boldsymbol{e}_j$$

の解をそれぞれ $\boldsymbol{c}_j$ とすれば，

$$\begin{bmatrix} A\boldsymbol{c}_1 & \cdots & A\boldsymbol{c}_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_1 & \cdots & \boldsymbol{e}_n \end{bmatrix} \iff A\begin{bmatrix} \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_1 & \cdots & \boldsymbol{e}_n \end{bmatrix}$$

が成り立ちます．すなわち，n 次正方行列 B を $B := \begin{bmatrix} \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_n \end{bmatrix}$ とすれば，

$$AB = E$$

ですから，A の逆行列 $A^{-1} = B$ が得られるわけです[*1]．

$\boldsymbol{c}_j$ を求めるときに $A\boldsymbol{x} = \boldsymbol{b}$ の形の方程式を n 回も解いていますが，毎回同じようなガウスの消去法をやるのは無駄が多い気がします．その点を解消するのが「**LU 分解**」のアイディアです．

[*1] 線形代数では，A^{-1} が必要になったら

$$\left[\begin{array}{c|c|c|c} A & \boldsymbol{e}_1 & \cdots & \boldsymbol{e}_n \end{array}\right] \longleftrightarrow \left[\begin{array}{c|c|c|c} E & \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_n \end{array}\right]$$

の形で行基本変形（これは $\left[\begin{array}{c|c} A & E \end{array}\right] \longleftrightarrow \left[\begin{array}{c|c} E & A^{-1} \end{array}\right]$ と同じこと）をすれば，A^{-1} がもとまるのでした．

正則行列 A に対し，ともに正則な下三角行列 L と上三角行列 U を見つけ，

$$A = LU$$

という積に分解できるとき，このような L と U のペアを A の **LU 分解** (LU decomposition) といいます．このとき，もとの連立方程式は

$$LU\boldsymbol{x} = \boldsymbol{b}$$

となりますが，$\boldsymbol{y} := U\boldsymbol{x}$ とおくと

$$\begin{cases} L\boldsymbol{y} &= \boldsymbol{b} \\ U\boldsymbol{x} &= \boldsymbol{y} \end{cases}$$

という 2 つの方程式に分解できるので，まず上の方程式を $\boldsymbol{y}$ について解き，次に下の方程式を $\boldsymbol{x}$ について解けばもとの方程式の解が求まるはずです．しかも，L と U はそれぞれ三角行列ですから，これら 2 つの方程式はガウスの消去法の後退代入と同じ原理で極めて簡単に解くことができるわけです．

8.2　LU 分解の求め方

LU 分解の存在は自明ではありませんが，以下では「ガウスの消去法がピボット選択なしでうまくいくこと」，すなわち

$$a_{11}^{(1)} \neq 0,\ \ a_{22}^{(2)} \neq 0,\ \ \cdots,\ \ a_{k-1,k-1}^{(k-1)} \neq 0$$

を仮定して話を進めます[*2]．

いまガウスの消去法における前進消去を思い出します．ここでは係数行列だけを考えれば十分なので，最初に与えられた

$$A =: A^{(1)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}^{(1)} & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} \end{bmatrix}$$

に対し，

$$A^{(1)} \xrightarrow{\text{(R2)}} A^{(2)} \ \cdots \ \xrightarrow{\text{(R2)}} A^{(n)} =: U$$

[*2] ピボット選択が必要な場合は行の入れ替えが入るので若干ややこしくなりますが，同様の計算を考えることができます．

と行基本変形 (R2) をひたすら繰り返し，最後に

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

(青い影の部分はすべて 0) の形になるように変形します．U のように，対角成分より下にある成分がすべて 0 となる行列を**上三角行列**というのでした．

いま，$k = 1, 2, \cdots, n-1$ に対して，$A^{(k)}$ から $A^{(k+1)}$ への変形は次で与えられます (灰色の網掛け部分は 0 にしたい部分)：

$$A^{(k)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1k}^{(1)} & a_{1,k+1}^{(1)} & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2k}^{(2)} & a_{2,k+1}^{(2)} & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots & \vdots & \ddots & \vdots \\ & & & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{kn}^{(k)} \\ & & & a_{k+1,k}^{(k)} & a_{k+1,k+1}^{(k)} & \cdots & a_{k+1,n}^{(k)} \\ & & & \vdots & \vdots & \ddots & \vdots \\ & & & a_{nk}^{(k)} & a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}$$

$\textcircled{i} \xrightarrow{\text{(R2)}} \textcircled{i} + \textcircled{k} \times \left(-a_{ik}^{(k)} / a_{kk}^{(k)}\right) \quad (i = k+1, k+2, \cdots, n)$：

$$A^{(k+1)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1k}^{(1)} & a_{1,k+1}^{(1)} & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2k}^{(2)} & a_{2,k+1}^{(2)} & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots & \vdots & \ddots & \vdots \\ & & & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{kn}^{(k)} \\ & & & & a_{k+1,k+1}^{(k+1)} & \cdots & a_{k+1,n}^{(k+1)} \\ & & & & \vdots & \ddots & \vdots \\ & & & & a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k+1)} \end{bmatrix}.$$

この操作は，行列 $A^{(k+1)}$ に

$$M_k = \begin{bmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ 0 & 0 & \ddots & & & & & \\ \vdots & \vdots & \ddots & 1 & & & & \\ \vdots & \vdots & \ddots & -m_{k+1,k} & 1 & & & \\ \vdots & \vdots & \ddots & -m_{k+2,k} & 0 & 1 & & \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & \cdots & -m_{nk} & 0 & \vdots & 0 & 1 \end{bmatrix}, \quad \begin{cases} m_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)} \\ i = k+1, k+2, \cdots, n \end{cases}$$

（赤の網掛け部分の成分すべて 0）の形の行列をかけて

$$A^{(k+1)} := M_k A^{(k)}$$

としていることと同じです．すなわち，

$$U = A^{(n)} = M_{n-1} M_{n-1} \cdots M_2 M_1 A$$

が成り立ちます．

さらに，$k = 1, \cdots, n-1$ に対し

$$M_k^{-1} = \begin{bmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ 0 & 0 & \ddots & & & & & \\ \vdots & \vdots & \ddots & 1 & & & & \\ \vdots & \vdots & \ddots & m_{k+1,k} & 1 & & & \\ \vdots & \vdots & \ddots & m_{k+2,k} & 0 & 1 & & \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & \cdots & m_{nk} & 0 & \vdots & 0 & 1 \end{bmatrix}$$

であることは実際に $M_{(k)}$ との積を計算すればわかりますし，

$$M_{(1)}^{-1} M_{(2)}^{-1} \cdots M_{(n-1)}^{-1} = \begin{bmatrix} 1 & & & & & & \\ m_{21} & 1 & & & & & \\ m_{31} & m_{32} & 1 & & & & \\ m_{41} & m_{42} & m_{43} & 1 & & & \\ \vdots & \vdots & \vdots & \ddots & \ddots & & \\ \vdots & \vdots & \vdots & \ddots & \ddots & 1 & \\ m_{n1} & m_{n2} & m_{n3} & \cdots & \cdots & m_{n,n-1} & 1 \end{bmatrix}$$

であることもわかるので，まさにこの下三角行列を

$$L := M_{(1)}^{-1} M_{(2)}^{-1} \cdots M_{(n-1)}^{-1}$$

とすれば

$$A = LU$$

が実現されることになります．以上を公式のような形でまとめておきましょう：

LU 分解による連立方程式の解法

(1) A を $A = LU = (m_{ij})(u_{ij})$ の形に分解する．
(2) $L\boldsymbol{y} = \boldsymbol{b} = (b_i)$ となる $\boldsymbol{y} = (y_i)$ を次のように計算する：

$$y_1 = b_1, \qquad y_k = b_k - \sum_{i=1}^{k-1} m_{ki} y_i \quad (k = 2, 3, \cdots, n)$$

(3) $U\boldsymbol{x} = \boldsymbol{y} = (y_i)$ となる $\boldsymbol{x} = (x_i)$ を次のように計算する：

$$x_n = \frac{y_n}{u_{nn}}, \qquad x_k = \frac{1}{u_{kk}} \left(y_k - \sum_{j=k+1}^{n} u_{kj} x_j \right) \quad (k = n-1, \cdots, 2, 1)$$

注意！　ガウスの消去法により（線形代数で学んだ方法で）逆行列 A^{-1} を求めるのに必要な加減算（足し算と引き算）および乗除算（掛け算と割り算）の回数は，それぞれ約 n^3 回ずつです．次に積 $A^{-1}\boldsymbol{b}$ を求めるのに，加減算と乗除算がそれぞれ n^2 回ずつ必要となります．

一方，LU 分解に必要な加減算と乗除算の回数は（`for` 文の無駄な部分を削れば）それぞれ約 $n^3/3$ 回ずつであり，さらに $L\boldsymbol{y} = \boldsymbol{b}, U\boldsymbol{x} = \boldsymbol{y}$ を満たす $\boldsymbol{y}$ と $\boldsymbol{x}$ を求めるために必要な加減算と乗除算の回数はそれぞれ約 n^2 回です．n が巨大な数である場合は，後者の方が効率的であることがわかります．

8.3　行列式

上のような前進消去（ピボット選択なし）の応用として，A の行列式を求めることを考えます．前進消去における $A^{(k)}$ から $A^{(k+1)}$ への変形は行基本変形の (R2) だけで実行されます．**(R2) の操作は行列式の値を変えない**ので，

$$|A| = |A^{(2)}| = \cdots = |A^{(n)}| = |U|$$

となります．上三角行列 U の行列式は（行列式の基本性質より）

$$|U| = u_{11}\, u_{22} \cdots u_{nn}$$

となるので，これが $|A|$ の値だとわかります．

ピボット選択が行われる場合でも，行基本変形の (R3) は 1 回ごとに -1 がかけられるだけなので，その回数さえ数えておけば $|A|$ の値が $|U|$ あるいは $-|U|$ として定まります．

8.4 Julia による実装：LU 分解

与えられた正方行列 A に対し，上のような手順で LU 分解（$A = LU$ となる L と U のペア）を求めるプログラムを書いてみましょう．

練習 8.1

```
function LUDecomposition(A) # LU 分解
    n = size(A, 1)          # 行のサイズ
    L = zeros(n, n)         # 初期化，ゼロ行列
    U = copy(A);            # 初期化，A の値をコピー
    for i = 1:n
        L[i,i] = 1.0        # 対角成分を 1 にしておく
    end
    for k = 1:n-1           # 2 行/列目以降の処理
        for i = k+1:n
            m = U[i, k]/U[k, k] # (R2) の係数
            L[i, k] = m          # m を L に格納
            for j = 1:k
                U[i, j] = 0     # 対角成分下を 0 に
            end
            for j= k+1:n        # ここは前進消去と同じ
                U[i, j] = U[i, j] - m * U[k, j]
            end
        end
    end
    return (L, U)        # L と U を返す
```

```
21 end
22 # 具体例
23 A = [2 2 1; 3 -1 0; -1 -3 2.0]
24 (L, U) = LUDecomposition(A)
25 display([L U])        # L と U を並べて行列表示
26 L*U                   # 検算．A になるか？
```

4 行目は U に A の値をコピー（copy）しています．25 行目は，このように書くことで L と U に関数の出力がそれぞれに代入されます．26 行目の display 関数は行列を行列っぽく表示してくれます．[L U] というのは hcat(L, U) と同じことです．

問題 8.1 LUDecomposition 関数をもちいて A の行列式を与える関数を作成せよ．

8.5 Julia による実装：LU 分解による方程式の解法

次に，与えられた A の LU 分解 $A = LU$ と列ベクトル $\boldsymbol{b}$ に対し，$A\boldsymbol{x} = \boldsymbol{b}$ の解を与える関数を作成してみます：

練習 8.2

```
1  function LUSolve((L, U), b)
2      n = size(L,1)    # L のサイズ
3      x = zeros(n)     # ゼロ列ベクトル，この値はダミー
4      y = zeros(n)     # ゼロ列ベクトル，この値はダミー
5      for k = 1:n      # L y = b の解を求めるループ
6          t = b[k]
7          for i= 1:k-1
8              t = t - L[k, i] * y[i];
9          end
10         y[k] = t;
11     end
12     for k = n:-1:1   # U x = y の解を求めるループ
```

```
        u = y[k]
        for i = k+1:n
            u = u - U[k, i] * x[i];
        end
        x[k] = u / U[k, k];
    end
    return x
end
# 具体例
A = [2 2 1; 3 -1 0; -1 -3 2.0];
(L, U)=LUDecomposition(A);
LUSolve((L, U),[1; 0; 0])
```

問題 8.2 LUSolve 関数をもちいて，A の逆行列を求めよ．

第9講　数値積分 1：リーマン積分と区分求積法

9.1　数値積分の必要性

積分 $\displaystyle\int_a^b f(x)\,dx$ を計算するときは，「微分積分学の基本定理」を用いるのが普通です．すなわち，$f(x)$ の原始関数 $F(x)$（微分したら $f(x)$ になる関数）が見つかれば

$$\int_a^b f(x)\,dx = F(b) - F(a)$$

が成り立つので，この右辺を計算すれば積分の値が求まるからです．しかし，

$$\int_a^b e^{-x^2}\,dx, \quad \int_a^b \frac{\sin x}{x}\,dx, \quad \int_a^b \frac{1}{\log x}\,dx$$

のように，積分される関数の原始関数が求まらない（初等的な関数の組み合わせで表現できない）ケースもあり，「微分積分学の基本定理」は事実上役に立たないことも多々あります[*1]．また，$x > 0$ に対し定義される**ガンマ関数**

$$\Gamma(x) := \int_0^\infty t^{x-1}\,e^{-t}\,dt$$

のように，広義積分を用いて定義される重要な関数もあるのでした．

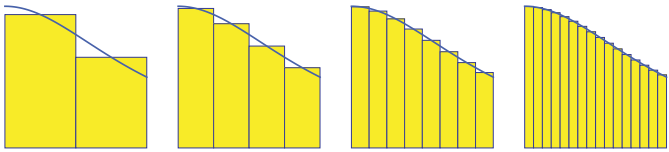
今回は，これらの積分を数値計算により，任意の精度で求める方法を学びます．以下，関数 $f(x)$ は定義域上で C^∞ 級であり，好きなだけ微分できるものとしましょう．

積分の定義のおさらい　まずは，積分の根底にある「区分求積法」のアイディアをおさらいします．

区間 $[a,b]$ 上の連続関数 $y = f(x)$ が与えられているとき，そのグラフと x 軸で囲まれた領域の（符号つき）面積を考えることができます．この領域をものすごく細い短冊（たんざく）で近似してみましょう．短冊ひとつひとつは長方形なので，その面積は「幅 × 高さ」となります．幅がものすごく小さいのでひとつの短冊の面積もまた小さいのですが，短冊たちが

[*1] これらの積分はそのまま特殊関数の定義として用いられます．たとえばガウスの誤差関数 $\mathrm{erf}(x) := \dfrac{2}{\sqrt{\pi}}\displaystyle\int_0^x e^{-t^2}\,dt$，正弦積分関数 $\mathrm{Si}(x) := \displaystyle\int_0^x \frac{\sin t}{t}\,dt$，対数積分関数 $\mathrm{Li}(x) := \displaystyle\int_2^x \frac{1}{\log t}\,dt$（これは x より小さい素数の数の近似値を与える）など．

細くなると同時に短冊の数も増加するので，面積の合計はある程度の大きさを保ちます．短冊がさらに細くなれば，短冊全体は関数のグラフと軸の間の領域を限りなく高い精度で近似して，面積の合計値はあるひとつの値に収束することが知られています．その値を，与えられた関数 $f(x)$ の区間 $[a,b]$ における**積分**もしくは**定積分**とよび，$\displaystyle\int_a^b f(x)\,dx$ と表すのでした．



区分求積法による定積分の定義　では，「定積分」のより正確な定義を与えていきましょう．原理的には，これが積分を数値計算するためのアルゴリズムにもなっているので，そのような気持ちで読んでみてください：

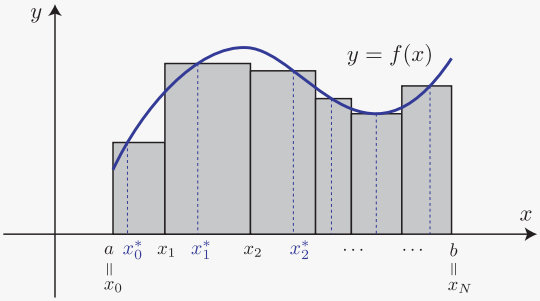
定義　(定積分)

Step 1. 区間 $[a,b]$ から**分割点** $\{x_k\}_{k=0}^{N}$ を

$$a = x_0 < x_1 < x_2 < \cdots < x_{N-1} < x_N = b$$

となるように選ぶ．これにより，区間 $[a,b]$ が N 個の小区間 $[x_k, x_{k+1}]$ $(0 \le k \le N-1)$ に細分される（シンプルに，区間を N 等分するように分割点を選んでもよい）．

Step 2. さらに，各小区間 $[x_k, x_{k+1}]$ から代表点 x_k^* を自由に選び，この区間において $y = f(x)$ のグラフと x 軸で囲まれる部分の面積を，幅 $x_{k+1} - x_k$，高さ $f(x_k^*)$ の長方形（これを**短冊**とよぶ）の符号つき面積 $f(x_k^*)(x_{k+1} - x_k)$ で近似する．



そのような N 個の短冊の面積和を考えると，

$$[\text{求めたい面積}] \approx \sum_{k=0}^{N-1} f(x_k^*)(x_{k+1} - x_k)$$

となる．この右辺の「近似値」を分割 $\{x_k\}_{k=0}^N$ に関する**リーマン和**とよぶ．

Step 3. 分割点 $\{x_k\}_{k=0}^N$ の選び方を，次のように変化させる：

- 点の個数を増やす．すなわち $N \to \infty$；かつ
- 分割の幅を一様に細かくする．すなわち $\max_{0\le k<N} |x_{k+1} - x_k| \to 0$

このとき，リーマン和はある一定の実数値 I に収束することが知られている．これを

$$\boldsymbol{I = \int_a^b f(x)\,dx}$$

と表し，連続関数 $f(x)$ の区間 $[a,b]$ における**定積分**という．また，$f(x)$ を定積分 I の**被積分関数**，a から b に向かう数直線上の経路を**積分区間**という．

注意！

- Step 1〜3 の方法で定積分を定める方法を**区分求積法**といいます．
- $a = b$，$a > b$ のときにはそれぞれ

$$\int_a^a f(x)\,dx := 0, \qquad \int_a^b f(x)\,dx := -\int_b^a f(x)\,dx$$

と定義します．

- 関数 $f(x)$ が連続でないときには，「区分求積法」では一定の積分値 I が定まらない可能性があります．一般に「区分求積法」で積分値が確定する（連続とは限らない）関数を**積分可能**な関数といいます．閉区間上の連続関数はすべて積分可能です．

9.2　リーマン和による近似

積分の定義 **Step 2** において，積分 $I = \int_a^b f(x)\,dx$ の近似値として

$$\Sigma := \sum_{k=0}^{N-1} f(x_k^*)(x_{k+1} - x_k) \tag{9.1}$$

を用いることが示唆されます．実際のところ，リーマン和 Σ は積分値 I の近似値としてどのくらいの精度（誤差）をもっているのでしょうか？それに答えるのが，次の定理です：

定理 9.1 (リーマン和の誤差) 式 (9.1) で定義される積分 I のリーマン和 Σ に対し，その分割の最大幅を $\Delta := \max_{0\le k<N}\{|x_{k+1} - x_k|\}$ とおく．また，積分区間 $[a,b]$ 上では $|f'(x)| \le K_1$ が成り立つと仮定する．このとき，積分値 I とリーマン和 Σ の絶対誤差は

次を満たす：

$$|\boldsymbol{I-\Sigma}| \le \boldsymbol{(b-a)K_1\Delta.}$$

とくに，分割点 $\{x_k\}$ として N 等分点をとったときのリーマン和を Σ_N とおくと，

$$|\boldsymbol{I-\Sigma_N}| \le \frac{\boldsymbol{K_1(b-a)^2}}{\boldsymbol{N}}.$$

ここで「N 等分点」とは，$\Delta x := \dfrac{b-a}{N}$ に対して $x_k := a + k\Delta x$ $(0 \le k \le N-1)$ とおいたものである．

すなわち，N 等分するリーマン和の場合，**代表点 $\{x_k^*\}$ の取り方に依存せず**，積分との誤差が少なくとも N に反比例して小さくなるのです．

【証明】 $f(x_k^*)(x_{k+1}-x_k) = \displaystyle\int_{x_k}^{x_{k+1}} f(x_k^*)\,dx$ であることに注意すると，三角不等式[*2]より

$$\begin{aligned}|I-\Sigma| &= \left|\int_a^b f(x)\,dx - \sum_{k=0}^{N-1}\int_{x_k}^{x_{k+1}} f(x_k^*)\,dx\right| \\ &= \left|\sum_{k=0}^{N-1}\int_{x_k}^{x_{k+1}} \{f(x)-f(x_k^*)\}\,dx\right| \le \sum_{k=0}^{N-1}\left|\int_{x_k}^{x_{k+1}} \{f(x)-f(x_k^*)\}\,dx\right|.\end{aligned}$$

$x_k \le x \le x_{k+1}$ のとき，平均値の定理より，$f(x)-f(x_k^*) = f'(c)(x-x_k^*)$ を満たす c が x ごとに存在するから，

$$|f(x)-f(x_k^*)| = |f'(c)(x-x_k^*)| \le K_1(x_{k+1}-x_k) \le K_1\Delta.$$

よって[*3]，

$$|I-\Sigma| \le \sum_{k=0}^{N-1} K_1\Delta(x_{k+1}-x_k) = K_1\Delta(b-a).$$

後半の主張は $\Delta = (b-a)/N$ を代入しただけである．■

例 1　積分

$$I = \int_a^b f(x)\,dx = \int_0^1 \frac{1}{x^2+1}\,dx$$

の値を適当なリーマン和によって近似してみます．この真の値は $\arctan 1 = \dfrac{\pi}{4} = 0.7853981634\cdots$ です．

[*2] 任意の複素数 z と w に対し，$|z+w| \le |z|+|w|$ が成り立ちます．これを**三角不等式** (the triangle inequality) というのでした．三角不等式を繰り返し用いることにより，実数 $A_1, A_2, \cdots, A_n$ に対し $|A_1+\cdots+A_n| \le |A_1|+\cdots+|A_n|$ がいえるので，それを用いています．

[*3] ここで，積分区間 $[a,b]$ 上で $|f(x)| \le M$ であるとき $\left|\displaystyle\int_a^b f(x)\,dx\right| \le M(b-a)$ であることと用いています．

区間 $[0,1]$ を N 等分する分割点

$$x_k = \frac{k}{N} \quad (0 \le k \le N)$$

を選び，$x_k \le x_k^* \le x_{k+1}$ $(0 \le k \le N-1)$ を満たす代表点として，簡単に

$$x_k^* = x_k = \frac{k}{N}$$

とします[*4]．これに対してリーマン和

$$\Sigma = \sum_{k=0}^{N-1} f(x_k^*)(x_{k+1} - x_k) = \sum_{k=0}^{N-1} \frac{1}{(k/N)^2+1} \frac{1}{N}$$

を計算してみたのが次の表です：

N	Σ_N	絶対誤差	相対誤差（%）
2	0.9	0.11460183660255174	14.59155902616465
4	0.8452941176470589	0.05989595424961058	7.626189752025231
8	0.8159971236227722	0.030598960225323935	3.89598061866608
16	0.8008604030103472	0.015462239612898943	1.9687134925313448
32	0.7931699732937437	0.007771809896295445	0.9895375694127445
64	0.7892942408714134	0.0038960774739651427	0.49606399092043013
128	0.7873487452659381	0.0019505818684898424	0.248355797020467
256	0.7863740901145704	0.0009759267171220998	0.12425884890034243
512	0.7858862857017304	0.0004881223042820792 6	0.062149662047919316
1024	0.785642264286018	0.00024410088856974266	0.031079890423198784

N を倍にすると，誤差が半分程度になっていますから，定理 9.1 と矛盾しません．

9.3　Julia による実装

上のリーマン和の計算をプログラムしてみましょう．まずは例 1 の計算を $N = 100$ で行うプログラムです：

練習 9.1

```
1 f(x) = 1/(x^2+1)    # 関数の定義
2 N = 100
```

[*4] 積分区間を N 等分して，その分割点での値から積分を求める一連の手法をニュートン・コーツ法といいます．

```
I = 0.0              # 積分の初期値
for k = 0:N-1
    I = I + f(k/N)*(1/N)     # リーマン和の計算
end
println("N = $N: I = $I")      # 値の出力
```

リーマン和は 0.7928939967307818 となるはずです．

問題 9.1 与えられた関数 f，区間の端点 a，b，等分割数 N に対し，リーマン和を与える関数 RiemannSum(f，a，b，N) を作成せよ．また，上の表のように $N = 2^i$ $(1 \leq i \leq 10)$ に対する値と誤差のデータを作成せよ．

問題 9.2 上で作成した関数 RiemannSum(f，a，b，N) を修正し，等分割点の間の代表点 x_k^* を k ごとにランダムに選びリーマン和を計算する関数 RiemannSumRandom(f，a，b，N) を作成せよ．ただし，0 と 1 の間の乱数（浮動小数点数）を発生させる関数は rand( ) である．

関数 rand( ) のカッコの中身に何も書かなくても，勝手に値を出力してくれます．各自実験してみてください．

9.4　収束速度の改良その 1：中点則

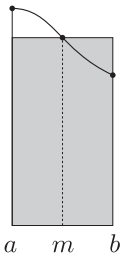
定理 9.1 より，原理的にはリーマン和 Σ_N の N を増加させることで，積分値 I が任意の精度で計算できます．しかし，誤差は N^{-1} にしか比例しないので，たとえば目標とする精度が誤差 10^{-5} 以内であるとき，10^5 程度の分割点に対して関数 $f(x)$ の値を計算することになります．

そこで，単純なリーマン和の計算よりも早く目標精度を実現してくれる 3 つの方法（中点則，台形則，シンプソン則）を紹介します．

中点則　まずは話を単純にするため，すでに区間 $[a,b]$ は十分に小さいと仮定してみます．このとき，積分値 I の大雑把な近似式として

$$I=\int_a^b f(x)\,dx \approx f(m)(b-a)$$

を採用してみます．ただし，$m=\dfrac{a+b}{2}$ で，a と b の中点のことです．これは図のように，グラフの面積を高さ $f(m)$ の長方形（短冊）によって近似することになります．



中点則 (midpoint rule) とは，与えられた積分区間 $[a,b]$ を N 等分して，その等分されたそれぞれの区間に対して上のような近似を適用し，積分の近似値を計算する方法です：

中点則.　積分区間 $I=[a,b]$ の N 等分点を $x_k:=a+k\Delta x$ $(0\le k\le N-1)$，x_{k+1} と x_k の中点を $m_k:=\dfrac{x_k+x_{k+1}}{2}$ とするとき，積分 I の近似値（リーマン和）として

$$\boldsymbol{M_N:=\Delta x\{f(m_0)+f(m_1)+\cdots+f(m_{N-1})\}}$$

を選ぶ．

簡単な変更に見えますが，次の定理のように，中点則はリーマン和の収束性をかなり改善してくれます：

定理 9.2 (中点則) 関数 $f(x)$ に対し $K_2:=\max\limits_{a\le x\le b}|f''(x)|$ とおくとき，上で与えた中点則による積分値 I の近似 M_N は

$$\boldsymbol{|I-M_N|\le\frac{K_2(b-a)^3}{24N^2}}$$

を満たす．すなわち，積分値との誤差は N^2 に反比例する．

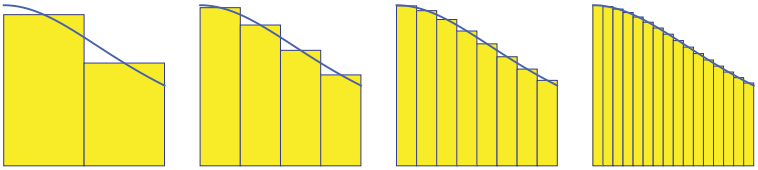
証明はそれほど難しくないのですが，省略します[*5]．

例 2　例 1 の積分に対し，$N=2,\,4,\,8,\,\cdots$ として中点則を用いた結果は次の表のようになります．

[*5] たとえば齊藤宣一「数値解析入門」(東京大学出版会)，p185 – を参照．

N	M_N	誤差	相対誤差（%）
2	0.7905882352941176	0.00519007189669313	0.6608204778857998
4	0.7867001295984857	0.0013019662010373967	0.16577148530694244
8	0.7857236823979222	0.00032551900047395055	0.0414463663966098
16	0.78547954357714	$8.138017969172573 \times 10^{-5}$	0.01036164629411586
32	0.7854185084490843	$2.0345051636061484 \times 10^{-5}$	0.0025904124282712304
64	0.7854032496604618	$5.086263013542869 \times 10^{-6}$	0.0006476031203766619
128	0.7853994349632036	$1.271565755356363 \times 10^{-6}$	0.00016190078034507592
256	0.7853984812888867	$3.1789143839500156 \times 10^{-7}$	$4.047519502972578 \times 10^{-5}$
512	0.7853982428703077	$7.947285940446136 \times 10^{-8}$	$1.0118798732693798 \times 10^{-5}$
1024	0.7853981832656631	$1.986821485111534 \times 10^{-8}$	$2.5296996831734494 \times 10^{-6}$

この例の場合，分割数が 2 倍になると誤差が約 1/4 になることがわかります．



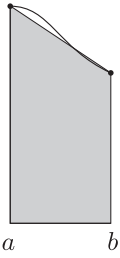
問題 9.3 関数 `RiemannSum(f, a, b, N)` を修正し，中点則を実現する関数 `midpontRule(f, a, b, N)` を作成せよ．また，上の表のような誤差の一覧を作成せよ．

9.5　収束速度の改良その 2：台形則

次に，十分小さな区間 $[a, b]$ に対する積分値 I の近似式として

$$I = \int_a^b f(x)\,dx \approx \frac{b-a}{2}\{f(a) + f(b)\}$$

を採用してみます．こちらはその名のとおり，グラフの形を台形によって近似しています．図の場合，若干足りない感じがしますが，近似精度は悪くなさそうです．



台形則 (trapezoidal rule) とは，与えられた積分区間を N 等分して，それぞれに上のような近似式を適用したものです．これはリーマン和とは異なるものですが，実際の積分値にちゃんと収束することが知られています．定理の形でまとめてみましょう：

定理 9.3 (台形則) 積分区間 $I=[a,b]$ の N 等分点を $x_k := a+k\Delta x$ $(0\le k\le N-1)$ とするとき，

$$\boldsymbol{T_N := \frac{\Delta x}{2}\{f(x_0)+2f(x_1)+2f(x_2)+\cdots+2f(x_{N-1})+f(x_N)\}}$$

は積分 I の近似値を与える．実際，$K_2 := \max_{a\le x\le b}|f''(x)|$ とするとき，

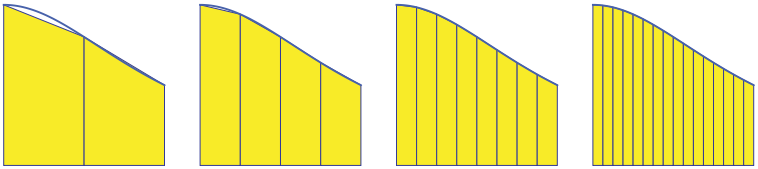
$$\boldsymbol{|I-T_N|\le \frac{K_2(b-a)^3}{12N^2}.}$$

意外な感じがしますが，中点則のほうが誤差は半分程度に抑えられます．

例 3（台形則の実行）　先ほどと同じ $I=\displaystyle\int_0^1 \frac{1}{x^2+1}\,dx = 0.7853981634\cdots$ の値を近似してみましょう．$N=2,4,8,\cdots$ に対し台形則を用いた結果は次の表のようになります：

N	T_N	誤差 T_N-I	相対誤差（%）
2	0.775	−0.010398163397448257	1.3239352830248852
4	0.7827941176470589	−0.0026040457503894165	0.3315574025695356
8	0.7847471236227722	−0.0006510397746760654	0.08289295863130365
16	0.7852354030103472	−0.00016276038710105745	0.020723296117346923
32	0.7853574732937437	$-4.069010370455483\times10^{-5}$	0.005180824911601396
64	0.7853879908714135	$-1.0172526034746276\times10^{-5}$	0.0012952062417286938
128	0.785395620265938	$-2.543131510268637\times10^{-6}$	0.00032380156063360863
256	0.7853975276145704	$-6.357828779002261\times10^{-7}$	$8.095039020080955\times10^{-5}$
512	0.7853980044517304	$-1.589457179207443\times10^{-7}$	$2.0237597352301205\times10^{-5}$
1024	0.785398123661018	$-3.973643025734219\times10^{-8}$	$5.059399437025892\times10^{-6}$

この場合も分割数が 2 倍になると誤差が約 1/4 になることがわかりますが，予想通り，全体的に見て中点則の 2 倍ほどの誤差が出ています．



問題 9.4 台形則を実現する関数 `trapezoidalRule(f, a, b, N)` を作成せよ．また，上の表のような誤差の一覧を作成せよ．

第10講　数値積分 2：シンプソン則・重積分・モンテカルロ法

10.1　収束速度の改良その３：シンプソン則

台形則はグラフの形を「線分」で近似していましたが，今度はグラフの形を「放物線」，すなわち 2 次関数で近似することを考えます．

まず準備として，次の公式を証明しましょう：

定理 10.1 (3 次以下の多項式の積分) $P(x)$ を 3 次以下の多項式とするとき，

$$\int_a^b \boldsymbol{P(x)\,dx} = \frac{\boldsymbol{b-a}}{\boldsymbol{6}}\{\boldsymbol{P(a)+4P(m)+P(b)}\}. \tag{10.1}$$

ただし，$m = \dfrac{a+b}{2}$（すなわち，m は a と b の中点）．

例 1　上の定理を用いて，高校数学でもおなじみの公式を導くことができる：

$$\int_a^b (x-a)(x-b)\,dx = \frac{b-a}{6}\left\{0+4\left(\frac{a+b}{2}-a\right)\left(\frac{a+b}{2}-b\right)+0\right\} = -\frac{(b-a)^3}{6},$$

$$\int_a^b (x-a)(x-b)^2\,dx = \frac{b-a}{6}\left\{0+4\left(\frac{a+b}{2}-a\right)\left(\frac{a+b}{2}-b\right)^2+0\right\} = \frac{(b-a)^4}{12}.$$

【証明（定理 10.1)】 $h := \dfrac{b-a}{2}$ とおき $t = x-m$ と変数変換すると，式 (10.1) は

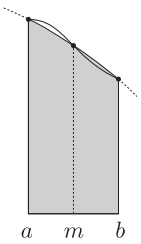
$$\int_{-h}^{h} P(t+m)\,dt = \frac{h}{3}\{P(-h+m)+4P(0+m)+P(h+m)\}$$

と変形される．したがって，最初から 3 次以下の多項式 $Q(t) := P(t+m)$ について

$$\int_{-h}^{h} Q(t)\,dt = \frac{h}{3}\{Q(-h)+4Q(0)+Q(h)\} \tag{10.2}$$

を示せば十分である．また，$Q(t) = pt^3+qt^2+rt+s$ の形に書けるので，$Q(t) = t^3,\ t^2,\ t,\ 1$ の場合について式 (10.2) を示せば十分である．たとえば $Q(t) = t^3$（奇関数）のとき，式 (10.2) は $0 = \dfrac{h}{3}((-h)^3+4\cdot 0+(h^3))$ となり成立している．残りも同様に確認できる．　■

積分の 2 次多項式近似　さて前回から引き続き，与えられた関数 $y = f(x)$ の区間 $[a, b]$ 上での積分 $I = \int_a^b f(x)\,dx$ を数値計算したいとします．そのために，関数 $f(x)$ そのものを多項式で近似することを考えてみましょう．もし区間 $[a, b]$ の幅がそれほど大きくなければ，関数 $f(x)$ の凹凸は少なく，たとえば 2 次関数であってもそれなりによい近似が得られると期待されます．



そこで $m = (a + b)/2$ として，区間 $[a, b]$ 上の関数 $y = f(x)$ を「$x = a, m, b$ のとき $P(x) = f(x)$」を満たす 2 次関数 $P(x)$ によって近似するのです[*1]．このとき，$P(x)$ の区間 $[a, b]$ 上での積分は，$P(x)$ を具体的に求めるまでもなく，公式 10.1 によって与えられます．さらに「$x = a, m, b$ のとき $P(x) = f(x)$」でしたから，積分値 I の近似式として

$$I = \int_a^b f(x)\,dx \approx \int_a^b P(x)\,dx = \frac{b-a}{6}\{f(a) + 4f(m) + f(b)\} \tag{10.3}$$

を採用することができます．

シンプソン則　積分の近似精度をあげるためには，まず区間 $[a, b]$ を十分細かく，**偶数個に**等分し，それらをふたつずつ束にして先ほど求めた近似式 (10.3) を適用するのがよいでしょう．それが次の「シンプソン則」とよばれる公式です：

定理 10.2 (シンプソン則) N を偶数とする．積分区間 $[a, b]$ の N 等分点を $x_k := a + k\Delta x$ $(\Delta x = (b-a)/N,\ 0 \le k \le N)$ とするとき，

$$\boldsymbol{S_N := \frac{\Delta x}{3}\{f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{N-1}) + f(x_N)\}} \tag{10.4}$$

は積分 I の近似を与える．実際，区間 $[a, b]$ 上で $|f^{(4)}(x)| \le K_4$ が成り立つとき，

$$\boldsymbol{|I - S_N| \le \frac{K_4(b-a)^5}{180N^4}}.$$

式 (10.4) の積分近似式を**シンプソン則** (Simpson's rule) という．

[*1] そのような $P(x)$ は $f(x)$ の「(2 次) ラグランジュ補間」とよばれ，次のように一意的に定まる：

$$P(x) = \frac{(x-m)(x-b)}{(a-m)(a-b)}f(a) + \frac{(x-b)(x-a)}{(m-b)(m-a)}f(m) + \frac{(x-a)(x-m)}{(b-a)(b-m)}f(b).$$

すなわち，誤差は少なくとも分割点の数 N の 4 乗に反比例します．

注意！　係数は

$$1,\ 4,\ 2,\ 4,\ 2,\ 4,\ \cdots,\ 4,\ 2,\ 4,\ 1$$

と並ぶことにも注意しましょう．式 (10.4) を数値計算するときは，係数毎にまとめて

$$\boldsymbol{S_N = \frac{\Delta x}{3}\Big\{f(x_0) + f(x_N) + 4\{f(x_1)+f(x_3) + \cdots + f(x_{N-1})\} + 2\{f(x_2) + f(x_4) + \cdots + f(x_{N-2})\}\Big\}}$$

とすることがあります[*2]．

例 1　前回にひきつづき，$[a,b] = [0,1]$，$f(x) = \dfrac{1}{x^2+1}$ とおき，シンプソン則を用いて積分

$$I = \int_a^b f(x)\,dx = \int_0^1 \frac{1}{1+x^2}\,dx$$

の値を近似してみましょう．真の値は $\mathrm{Tan}^{-1}1 = \dfrac{\pi}{4} = 0.785398163397\cdots$ でした．4 倍すれば円周率です．

たとえば $N = 4$ のとき，$\Delta x = 1/4$ であり，式 (10.4) は

$$\begin{aligned} S_4 &= \frac{1/4}{3}\left\{f(0) + 4f\left(\frac{1}{4}\right) + 2f\left(\frac{1}{2}\right) + 4f\left(\frac{3}{4}\right) + f(1)\right\} \\ &= \frac{1}{12}\left\{1 + \frac{4}{1+(1/4)^2} + \frac{2}{1+(1/2)^2} + \frac{4}{1+(3/4)^2} + \frac{1}{1+1^2}\right\} \\ &= \frac{8011}{10200} = 0.78539215\cdots \end{aligned}$$

と計算できます．真の値との誤差は約 6.0×10^{-6} で，相対誤差も約 0.00076%．また，$4S_4 = 3.141568\cdots$ は π と小数点以下 4 桁まで一致します．簡単な計算のわりには，驚くべき精度です[*3]．

$N = 2,\ 4,\ 8,\ 16,\ \cdots$ に対しシンプソン則を用いた結果は次の表のようになります：

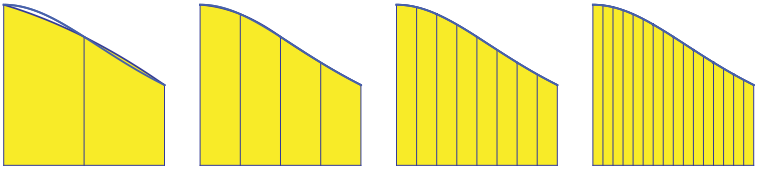
[*2] ただし，$N = 2$ のときは最後の $f(x_{N-2})$ は計算せず $f(x_0) + 4(x_1) + f(x_2)$ となります．

[*3] ちなみに円に内接する正多角形の面積で同程度の精度を実現するには，正 927 角形が必要です．

N	S_N	誤差	相対誤差（%）
2	0.7833333333333333	-0.002064830064114953	0.26290232907891997
4	0.785392156862745	$-6.006534703284494 \times 10^{-6}$	0.0007647757511045905
8	0.7853981256146767	$-3.778277157806542 \times 10^{-8}$	$4.8106518882888665 \times 10^{-6}$
16	0.7853981628062056	$-5.912427214482818 \times 10^{-10}$	$7.52793613484789 \times 10^{-8}$
32	0.7853981633882091	$-9.23916498862809 \times 10^{-12}$	$1.1763670223853884 \times 10^{-9}$
64	0.7853981633973041	$-1.4421797089880783 \times 10^{-13}$	$1.8362402360982704 \times 10^{-11}$
128	0.7853981633974457	$-2.55351295663786 \times 10^{-15}$	$3.2512336743849283 \times 10^{-13}$
256	0.7853981633974484	$1.1102230246251565 \times 10^{-16}$	$1.4135798584282296 \times 10^{-14}$
512	0.785398163397448	$-3.3306690738754696 \times 10^{-16}$	$4.240739575284689 \times 10^{-14}$
1024	0.7853981633974486	$3.3306690738754696 \times 10^{-16}$	$4.240739575284689 \times 10^{-14}$

分割数が 2 倍になると誤差が 1/100 近くに減少するので，中点則・台形則よりも早く高精度な値が得られます．$N = 256 = 2^8$ の段階で誤差は 64 ビット浮動小数点の限界に近づいており，その後は誤差が一旦増加しています．

次の図は，シンプソン則で計算している $f(x)$ の放物線近似を $N = 2, 4, 8, 16$ に対して描いたものです．



Julia による実装　早速プログラムを書いてみましょう．以下では，上の注意に書いたように，足し合わせる係数ごとにまとめて計算しています．

練習 10.1

```
# シンプソン則による数値積分
f(x) = 1/(x^2 + 1) # 関数の定義
N = 100
d = 1/N              # 分割の刻み幅
s1 = f(0) + f(1)     # 和の計算：係数が 1
s4 = 0.0             # 和の計算：係数が 4
for k = 1:2:N-1
    s4 = s4 + f(k/N)
```

```
end
s2 = 0.0              # 和の計算：係数が 2
for k = 2:2:N-2
    s2 = s2 + f(k/N)
end
I = (s1 + 4*s4 + 2*s2)*d/3 # シンプソン則による近似値
println("N = $N: I = $I")
```

問題 10.1 シンプソン則を実現する関数 `SimpsonRule(f, a, b, N)` を作成せよ．また，上の表のような誤差の一覧を作成せよ．

10.2　数値計算の限界？

次の問題を考えてみましょう：

問題 10.2 例 1 の $f(x)$ と同じく，$[0,1]$ 区間上での積分値が $\pi/4$ となる関数 $g(x) = \sqrt{1-x^2}$ に対してシンプソン則を適用し，下のような誤差の一覧を作成せよ．さらに，精度が思うように上がらない理由を考えよ．

N	S_N	誤差	相対誤差（%）
2	0.7440169358562924	−0.04138122754115592	5.268821531508355
4	0.7708987887367403	−0.014499374660707942	1.8461177191944336
8	0.7802972924438544	−0.005100870953593906	0.649463061070672
16	0.7835994172461492	−0.0017987461512990466	0.22902347307741241
32	0.7847630544733987	−0.00063510892404961	0.08086457973141645
64	0.7851737690201337	−0.00022439437731458511	0.028570779481315266
128	0.7853188547338981	$-7.930866355021493 \times 10^{-5}$	0.010097892667222986
256	0.7853701282860254	$-2.8035111422908265 \times 10^{-5}$	0.003569541250470328
512	0.7853882523267827	$-9.911070665613586 \times 10^{-6}$	0.0012619167102123868
1024	0.7853946594530347	$-3.503944413618676 \times 10^{-6}$	0.00044613605899731596

問題 10.3 積分

$$\int_0^1 g(x)\,dx = \int_0^1 \sqrt{1-x^2}\,dx$$

を $t^2 = 1 - x$ を満たす $t \geq 0$ を用いて変数変換せよ．それにシンプソン則を適用すれば，精度が改善されることを確認せよ．

10.3　重積分の定義

重積分の数値計算を考えてみましょう．重積分の定義は複雑ですから，xy 平面 $\mathbb{R}^2$ 内のある長方形上で定義された 2 変数関数の積分に限定して考えていきます．また，便宜的に（ア）から（エ）までの 4 ステップにわけて定義を述べていきます．

（ア）：区画　数直線 $\mathbb{R}$ 上の閉区間 $[a,b]$, $[c,d]$ に対し，

$$D = \big\{(x,y) \;\big|\; a \leq x \leq b,\ c \leq y \leq d\big\}$$

と表される集合を**区画**とよび，$\boldsymbol{D = [a,b] \times [c,d]}$ と表します．

以下，（少なくとも）区画 D 上で定義された関数 $z = f(x,y)$ の積分（重積分）を定義しましょう[*4]．

（イ）：区画の分割　区間 $[a,b]$ を m 分割する点

$$a = x_0 < x_1 < \cdots < x_{m-1} < x_m = b$$

と区間 $[c,d]$ を n 分割する点

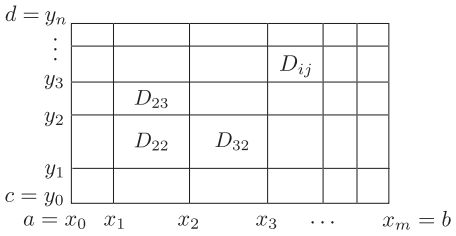
$$c = y_0 < y_1 < \cdots < y_{n-1} < y_n = d$$

をとります．これによって，区画 $D = [a,b] \times [c,d]$ は図のような mn 個の区画に分割されるので，それぞれを

$$D_{ij} := [x_{i-1}, x_i] \times [y_{j-1}, y_j]$$

（ただし $1 \leq i \leq m$ かつ $1 \leq j \leq n$）とおきます．

[*4] 現時点では，$f(x,y)$ は必ずしも連続関数とは限りません．



（ウ）代表点選びとリーマン和の計算　それぞれの D_{ij} から，**代表点** $(x_{ij}^*, y_{ij}^*) \in D_{ij}$ を自由に選んで，

$$\boldsymbol{\Sigma := \sum_{j=1}^{n} \sum_{i=1}^{m} f(x_{ij}^*, y_{ij}^*) \times (x_i - x_{i-1}) \times (y_j - y_{j-1})} \tag{10.5}$$

とおきます．この量も 1 次元のときと同様に，**リーマン和**といいます．

（エ）リーマン和の極限としての重積分　さらに，「リーマン和の極限」として重積分を定義しましょう：

定義　(区画上の重積分) 関数 $z = f(x, y)$ が区画 D 上で**積分可能**であるとは，区画 D の分割の最大幅

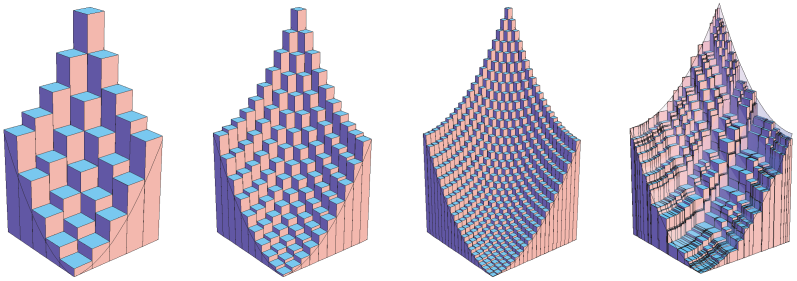
$$\max\{|x_1 - x_0|,\ \cdots,\ |x_m - x_{m-1}|, |y_1 - y_0|,\ \cdots,\ |y_n - y_{n-1}|\}$$

が 0 に近づくように区画の分割点の数を増やすとき，そのような分割点の選び方，代表点 $\left\{(x_{ij}^*, y_{ij}^*) \;\middle|\; 1 \le i \le m,\ 1 \le j \le n\right\}$ の選び方に依存せず，式 (10.5) のように計算したリーマン和 Σ が一定の実数値 I に近づくことをいう．この数

$$\boldsymbol{I = \iint_D f(x, y)\, dxdy}$$

を関数 $f(x, y)$ の区画 D 上での**重積分**（もしくは単に**積分**）という．また D を**積分領域**，$f(x, y)$ を**被積分関数**という．

例 2（数値計算の例）　区画 $D = [0, 1] \times [0, 1]$ に対し，上の関数 $f(x, y) = x^2 + y^2$ の積分を考えてみましょう．1 次元の積分が直観的には「短冊の面積和」であったように，2 次元の積分は「細い角材を寄せ集めた体積和」のように解釈できます．たとえば下の図は，区間 $[0, 1]$ の分割点として 5 等分，10 等分，20 等分と増やしたときの「角材」の様子です．ただし，代表点は等分割された各区画のなかでもっとも大きな値を取る右上隅の頂点を選んでいます．また，右端の図は分割点・代表点をランダムに選んだものです．



さて積分

$$\iint_D (x^2+y^2)\,dxdy$$

の真の値は $2/3 = 0.6666\cdots$ です．例のごとく，$N = 2, 4, 8, \cdots$ に対して数値計算した結果は次のようになります：

N	リーマン和	誤差 $\Sigma - I$	相対誤差（%）
2	1.25	0.5833333333333334	87.50000000000001
4	0.9375	0.27083333333333337	40.62500000000001
8	0.796875	0.13020833333333337	19.531250000000007
16	0.73046875	0.06380208333333337	9.570312500000007
32	0.6982421875	0.03157552083333337	4.736328125000006
64	0.682373046875	0.01570638020833337	2.3559570312500058
128	0.67449951171875	0.00783284505208337	1.1749267578125056
256	0.6705780029296875	0.00391133626302087	0.5867004394531306
512	0.6686210632324219	0.0019543965657552453	0.2931594848632868
1024	0.6676435470581055	0.000976880391438839	0.14653205871582586

N を 2 倍にすると，精度はだいたい半分ぐらいにはなります．しかし，リーマン和において $f(x,y)$ の値を計算すべき点の数は N^2 に比例するので，N が 2 倍になると 4 倍の点において関数値を計算しなくてはなりません．

数値積分の精度　2 変数の場合でも，定理 9.1 と同様に，式 (10.5) で与えられるリーマン和が，積分の近似としてどの程度の精度があるのかを示す定理が証明できます[*5]：

定理 10.3 (重積分とリーマン和の誤差) 区画 $D = [a,b] \times [c,d]$ 上の C^1 級関数 $z = f(x,y)$ に対し，重積分の値を $I := \iint_D f(x,y)\,dxdy$ とおき，式 (10.5) で与えられる

[*5] 証明は川平友規『微分積分—1 変数と 2 変数』（日本評論社）定理 25.3 参照.

リーマン和 Σ の分割の最大幅

$$\max\{|x_1 - x_0|,\ \cdots,\ |x_m - x_{m-1}|, |y_1 - y_0|,\ \cdots,\ |y_n - y_{n-1}|\}$$

を Δ とおく．また，定数 K_1 を区画 D 上において $\max\{|f_x(x,y)|, |f_y(x,y)|\} \leq K_1$ を満たすようにとる．このとき，

$$\boldsymbol{|I - \Sigma| \leq 2K_1 \mathrm{Area}\,(D)\Delta = 2K_1(b-a)(d-c)\Delta.}$$

とくに，分割点 $\{x_k\}$，$\{y_k\}$ としてそれぞれ N 等分点をとったときのリーマン和を Σ_N とおくと，

$$\boldsymbol{|I - \Sigma_N| \leq \frac{K_1 \mathrm{Area}\,(D)\{(b-a)+(d-c)\}}{N}.}$$

いずれにしても計算誤差は分割の最大幅に比例して小さくなっていますが，上で述べたとおり，区間を N 等分したら代表点での $f(x,y)$ の値を N^2 個計算しなくてはならないので，1 変数の積分に比べると精度を上げるために必要となる計算量が非常に大きくなります．たとえば，誤差を 1/10 にしたかったら，計算量は約 100 倍必要だと言うことです．

例 3　例 2 の数値計算では，$D = [0,1] \times [0,1]$，$f(x,y) = x^2 + y^2$，$f_x(x,y) = 2x$，$f_y(x,y) = 2y$ より $K_1 = 2$ である．よって $|I - \Sigma_N| \leq 4/N$ となる．

Julia での計算　以下では，区画 $[a,b] \times [c,d]$ 上で定義された $f(x,y)$ に対し，区間 $[a,b]$ と $[c,d]$ をそれぞれ N 等分する分割点をとり，各区画の代表点として「区画の左下隅の頂点」を選んだ場合にリーマン和を計算するプログラムです：

練習 10.2

```
# 2 次元リーマン和
function RiemannSum2(f, a, b, c, d, N)
    I = 0.0              # 積分の初期値
    dx = (b - a)/N       # x 方向の刻み幅
    dy = (d - c)/N       # y 方向の刻み幅
    dA = dx * dy         # 小区画の面積
    for i = 0:N-1
        for j = 0:N-1
            I = I + f(a + i*dx, c + j*dy) * dA
```

```
10          end
11      end
12      return I                # リーマン和の値
13  end
```

問題 10.4 例 1 に対し，表のような誤差の一覧を作成せよ．

問題 10.5 上の `RiemannSum2` 関数を修正し，「中点則」を適用した関数 `midpointRule2(f, a, b, c, d, N)` を作成し，精度が向上するか確認せよ．

10.4　モンテカルロ法

高次元の重積分は非常に計算コストがかかりますが，万策つきたときの「すがる藁」とされるのが，次の**モンテカルロ法** (Monte Carlo Method) です．数値計算というよりは「数値シミュレーション」という感もありますが，乱数に関するプログラムの練習も兼ねて少しだけ取り上げます．

原理　話をわかりやすくするために，2 次元の積分を考えます．ある区画 $D = [a,b]\times[c,d]$ が与えられており，その上の（連続とは限らない）関数 $f(x,y)$ を考えます．モンテカルロ法とは，与えられた集合 E $(\subset D)$ 上での積分

$$I = \iint_E f(x,y)\,dxdy := \iint_D f(x,y)\chi_E(x,y)\,dxdy$$

（ただし，$\chi_E(x,y)$ は $(x,y)\in E$ のとき 1，$(x,y)\notin E$ のとき 0 となる関数）の値を次のように近似する方法です：まず，$X_1, X_2, \cdots$ を区間 $[a,b]$ 上の一様乱数[*6]，$Y_1, Y_2, \cdots$ を区間 $[c,d]$ 上の一様乱数とします．このとき，十分大きな自然数 N に対し

$$I_N = \frac{(b-a)(d-c)}{N}\sum_{k=1}^{N} f(X_k,Y_k)\chi_E(X_k,Y_k)$$

を I の近似値として採用するものです[*7]．

[*6] 「一様乱数とは何か」を説明するのは難しいですが，任意の区間 $I\subset[a,b]$ に対し，X_k が I に値をとる確率が $[I\text{ の長さ}]/(b-a)$ になるような変数だといえます．

[*7] $|I-I_N|$ は $O(\sqrt{(\log\log N)/N})$ 程度であることが知られています．

例 4　例 1 の積分に $D = E = [0,1] \times [0,1]$ としてモンテカルロ法を適用してみましょう．$[0,1]$ 区間上の一様乱数を発生させる関数は `rand()` ですので，これを使います（括弧の中は何も入れなくて大丈夫です）．

練習 10.3

```
# 区画上のモンテカルロ法
function MonteCarlo(f, a, b, c, d, N)
    s = 0.0          # 和の初期値
    for k = 1:N      # 一様乱数による和
        s = s + f(a + (b-a)*rand(), c + (d-c)*rand())
    end
    return s*(b-a)*(d-c)/N    積分の近似値
end
# 具体例
h(x, y) = x^2 + y^2
MonteCarlo(h, 0, 1, 0, 1, 1000)    # N = 1000 でテスト
```

11 行目を何度か実行してみると，毎回値が変化することがわかると思います．$N = 10, 100, 1000, \cdots$ と増やして実行してみた結果が次のとおりです（最後は 10^{10} で，さすがに少し時間がかかるので 10^8 ぐらいにしておくとよいかもしれません）．

N	S_N	誤差	相対誤差（%）
10	0.7440072601211611	0.07734059345449451	11.601089018174177
100	0.6391674761654577	−0.02749919050120897	4.124878575181346
1000	0.676732841077962	0.010066174411295425	1.5099261616943138
10000	0.6698209544537919	0.003154287871252317	0.47314316806878476
100000	0.6672850495728007	0.0006183829061340296	0.09275743592010444
1000000	0.6669371560558914	0.0002704893892248039	0.04057340838372059
10000000	0.6665547771032121	−0.00011188956345453693	0.01678343451818054
100000000	0.6666944584396934	$2.7791773026808464 \times 10^{-5}$	0.00416876595402127
1000000000	0.66666177357423	$-4.893092436675239 \times 10^{-6}$	0.0007339638655012859
10000000000	0.666662412963001	$-4.253703665635378 \times 10^{-6}$	0.0006380555498453067

問題 10.6 上の表の最後の 2 行では，それまでの行に比べてあまり精度が向上していない．理由として考えられるものは何か？

例 5.　モンテカルロ法により円周率の近似を考えてみましょう．区画 $D =$

$[0,1]\times[0,1]$ 上の関数 $p(x,y)$ を，(x,y) が単位円の中にあれば 1，外にあれば 0 と定めます．すなわち，

$$p(x,y) := \begin{cases} 1 & : x^2+y^2 \leq 1 \text{ のとき} \\ 0 & : x^2+y^2 > 1 \text{ のとき} \end{cases}$$

と定めるわけです．これは，$E = \left\{(x,y) \in D \;\middle|\; x^2+y^2 \leq 1\right\}$ としたときの $\chi_E(x,y)$ と同じものですから，

$$\iint_D p(x,y)\,dxdy = \iint_D 1\cdot\chi_E(x,y)\,dxdy.$$

この値は E の面積そのものですから，$\pi/4$ となります．

問題 10.7 `MonteCarlo` 関数を用いて上の積分の近似値をもとめよ．また，π の近似値を計算せよ．

第11講　微分方程式 1：オイラー法

11.1　微分方程式とは

多くの自然法則は「微分方程式」とよばれる関数に関する方程式によってで表現されます．したがって，「未来を予測する」ことはしばしば「微分方程式を解く」ことに帰着されるのです．今回から 3 回にわたり，微分方程式の数値解法について学んでいきましょう．

注意！　以下では，関数を $y = f(x)$ の形ではなく，$y = y(x)$ の形で表すことにします．

11.2　微分方程式

指数関数 $y = e^x$ は $y' = e^x$ を満たすので，x の関数としての関係式 $y' = y$ $(y(x) = y'(x))$ を満たします．すなわち，「導関数が自分自身と一致する関数」というわけですが，ほかにもそのような関数は存在するのでしょうか？

同様に，$y = \sin x + \cos x$ のとき $y'(x) = \cos x - \sin x$, $y''(x) = -\sin x - \cos x$ なので，$y'' = -y$ を満たします．このような関係式を満たす関数をすべて求めることは可能でしょうか？

これらの問いは「微分方程式」の問題として定式化されます：

定義（微分方程式） 未知の関数 $y = y(x)$ とその導関数 $y', y'', \cdots$ の関係式を**微分方程式**とよぶ．とくに，関係式に含まれる最高階の導関数が $y^{(n)}$ であるとき，n **階微分方程式**という．

微分方程式を満たす関数をその**解**とよぶ．「微分方程式を解く」とは，そのような解をすべて求めることをいう．

現実的には，微分方程式に解が存在していても，x の式で具体的に表現できないことも多いため，実用上は数値的に微分方程式を解く必要があるわけですが，まずは解が具体的に求まる（どちらかというと，珍しい）例をみていきましょう．

ウサギの数を予測する　自然環境にあるウサギの個体数の増減について，微分方程式によるモデルを作ってみましょう．

モデルその１（指数関数）　餌（草）が無尽蔵にある環境では，ウサギは指数関数的に増加すると考えられます．そのような状況を数学的にモデル化してみます．

時間 x におけるウサギの個体数を $y = y(x)$ とおくと[*1]，期間 Δx の間に増加する個体数 Δy はその時点での個体数 y に比例すると考えられるので，

$$\Delta y = ky \cdot \Delta x \iff \frac{\Delta y}{\Delta x} = ky \quad (\text{ただし } k \text{ は正の定数})$$

と表現できます．したがって微分方程式

$$y' = ky$$

をそのモデルとして採用することができるわけです．

じつは次の例題 11.1 より，この微分方程式の解は具体的に求めることができて，個体数 y は指数関数 $y = Ae^{kx}$ $(A = y(0)$ によって推定されることがわかります：

例題 11.1 (指数関数の微分方程式) $k \neq 0$ とするとき，微分方程式

$$y' = ky \tag{11.1}$$

を解け．また，解のなかで $y(0) = 2$ を満たすものを求めよ．

解答　$y = y(x)$ が定数関数 $y = 0$ の場合，明らかに $y' = ky$ を満たす．そうでない場合，

$$\begin{aligned} y' = ky &\iff \frac{y'}{y} = k \\ &\iff \frac{d}{dx}\log|y| = k \\ &\iff \log|y| = kx + C_0 \quad (C_0\text{は任意の定数}) \\ &\iff |y| = e^{kx+C_0} \\ &\iff y = \pm e^{C_0} e^{kx} \end{aligned}$$

と変形できる．よって $A := \pm e^{C_0} \neq 0$ とおくと，$y = Ae^{kx}$ が定数関数でないすべての解となる．$A = 0$ とすれば定数関数 $y = 0$ も解として含むので，微分方程式 $y' = ky$ の解は $y = Ae^{kx}$ （A は任意の定数）となる．

また，$y = Ae^{kx}$ が $y(0) = 2$ を満たすとき，$2 = A$．よってそのような解は $y = 2e^{kx}$．　■

モデルその２（ロジスティック関数）　実際の自然環境には餌が無尽蔵にあるわけではなく，天敵（捕食者，たとえばキツネ）の存在も無視できません．そのような要素を考

[*1] 実際には月や年を単位として観測すべきでですが，モデルとしては連続的に変化する時間を考えるのがふつうです．

慮して，モデルその 1 の比例定数 k から y に比例する量を差し引いて，$k-ay$ （ただし，a は正の定数）としてみます．すなわち，微分方程式

$$y' = (k-ay)y$$

をウサギの個体数のモデルとして採用してみましょう．これは，「個体生物学」とよばれる分野で**ロジスティック方程式**として知られているものです[*2]．

例題 11.2 (ロジスティック方程式) a と k を正の定数とするとき，微分方程式

$$y' = (k-ay)y \tag{11.2}$$

を解け．また，$k=1, a=1$ のとき，解のなかで $y(0)=1/4$ を満たすもののグラフを描け．

解答　定数関数 $y=0$ および $y=k/a$ は微分方程式 (11.3) を満たす．
そうでないとき，

$$\frac{1}{(k-ay)y} = \frac{1}{k}\left(\frac{1}{y} + \frac{a}{k-ay}\right)$$

に注意すると，

$$\begin{aligned}
y' = (k-ay)y &\iff \frac{y'}{(k-ay)y} = 1 \\
&\iff \frac{y'}{y} + \frac{ay'}{k-ay} = k \\
&\iff (\log|y|)' - (\log|k-ay|)' = k \\
&\iff \log|y| - \log|k-ay| = kx + C \quad (C \text{ は任意の定数}) \\
&\iff \log\left|\frac{k-ay}{y}\right| = -kx - C \\
&\iff \frac{k}{y} - a = \pm e^{-C} e^{-kx}.
\end{aligned}$$

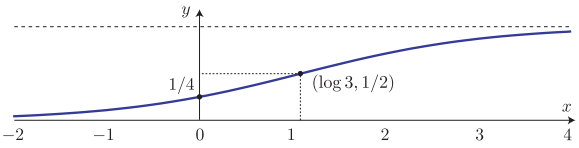
ここで $A := \pm e^{-C}$ とおくと，

$$\frac{y}{k} = \frac{1}{a + Ae^{-kx}} \iff y = \frac{k}{a + Ae^{-kx}}.$$

この式は $A=0$ のとき定数関数 $y=k/a$ を表すので，A を任意の実数として $y = k/(a+Ae^{-kx})$ および $y=0$ が求める解である．

また，$k=a=1$ のとき，$y(0)=1/4$ より $1/4 = 1/(1+A)$，よって $A=3$ である．増減表や凹凸を調べてグラフを描くと下のようになる．（変曲点は $x=\log 3$ のとき．）　■

[*2] 本来はウサギでなく，人口（ヒトの数）増加のモデルとしてベルギーの数学者フェルフルスト（Pierre-François Verhulst, 1804-1849）によって導入されました．



例題の結果の解釈　たとえばウサギ 1000 匹を 1 単位とみてグラフを解釈してみましょう．ある時点で 250 匹（$y(0) = 1/4$）であったウサギの数は時刻とともに順調に増加しますが，外敵等の環境的要因により増加が次第に抑制され，1000 匹を越えることはできないことがわかります．

11.3　参考：方向場

ロジスティック方程式 (11.3) の定数関数でない解は**ロジスティック関数**とよばれています．その方程式と解の関係を視覚的に理解する方法を紹介します．

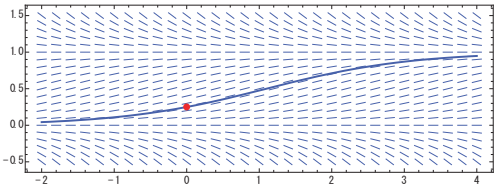
話を単純にするために，$k = a = 1$ の場合を考えると，ロジスティック方程式は

$$y' = (1 - y)y \tag{11.3}$$

となります．この関係式から，もし解 $y = y(x)$ が点 $(0, 1/4)$ を通るならば，そこでの微分係数は $y'(0) = (1 - 1/4) \cdot (1/4) = 3/16$ で与えられることがわかります．より一般に，

- $y < 0$ もしくは $y > 1$ のとき $y' < 0$
- $y = 0$ もしくは $y = 1$ のとき $y' = 0$
- $0 < y < 1$ のとき，$y' > 0$

だとわかります．そこで，xy 平面の各点 (x, y) に傾き $(1 - y)y$ の短い線分を描くことにすれば，図のようにロジスティック方程式 (11.4) の解の様子が推定できるわけです．



このような短い線分（微小線分）の分布を**方向場**といいます．上で扱った 2 つの微分方程式は式変形と積分によって「運良く」解けましたが，一般には（たとえ解が存在しても）そのような式変形が見つかるとは限りません．そこで，解の挙動を推定するひとつの方法として，方向場が用いられるわけです．

11.4　解の存在と一意性

関数 $y = y(x)$ が関係式

$$\boldsymbol{\frac{dy}{dx} = f(x,\, y)}$$

を満たすものとします．ただし，2 変数関数 $(s,t) \mapsto f(s,\,t)$ は連続関数であるものとしましょう．もう少し詳しく書くと，上の関係式は

$$\boldsymbol{\frac{d}{dx}y(x) = f(x,\, y(x))}$$

ということになります．この形の方程式を **1 階常微分方程式**といい，ふつうは，これにある $x = x_0$ における値を指定した

$$\boldsymbol{y(x_0) = A_0}$$

という形の条件（これを**初期条件**といいます）を加えて，解の存在や解の一意性[*3]について議論します．たとえば例題 11.1 では，$f(s,t) = kt$ (k は定数) の形の（s の出てこない特別な）2 変数関数に対して，$y(0) = 2$ という初期条件を考えました．また，例題 11.2 では，$f(s,t) = (1-t)t$ の形の 2 変数関数に対して $y(0) = 1/4$ という初期条件を考えました．

解の存在や一意性を保証するための一般論については微分方程式の教科書等を見ていただくことにして，ここでは次の定理が成り立つような状況だけを考えることにしましょう：

定理 11.3 (解の存在と一意性) 2 変数関数 $(s,t) \mapsto f(s,t)$ は以下を満たすものとする．ある $(a,b) \in \mathbb{R}^2$ と $p,q,L,M > 0$ が存在し，

- **連続・有界性：**$f(s,t)$ は (a,b) を含む区画 $D = [a-p, a+p] \times [b-q, b+q]$ において連続かつ $|f(s,t)| \leq M$.
- **リプシッツ連続性：**$(s,t_1), (s,t_2) \in D$ のとき，

$$\boldsymbol{|f(s,t_1) - f(s,t_2)| \leq L\,|t_1 - t_2|.}$$

[*3] 「解の一意性」とは，与えられた初期（値）条件に対し，微分方程式の解がただ 1 つだけに定まることをいいます．現実世界の物理法則を考えると，同一の条件下で複数の解が発生することは「不自然」なので，解の一意性が検討されるわけです．

このとき，$r = \min\{p, q/M\}$ に対し，微分方程式

$$\begin{cases} \dfrac{dy}{dx} = f(x, y) \\ y(a) = b \end{cases}$$

は区間 $[a - r, a + r]$ 上でただ 1 つ解 $y = y(x)$ をもつ．

注意！　区画（コンパクト集合）上の連続関数は最大値と最小値をもつ（値が正負に発散したりはしない）ことが知られているので，$|f(s,t)| \leq M$ を満たす $M > 0$ の存在は連続性だけから自動的に導かれます．

11.5　微分の近似方法

さて，以下では 1 階常微分方程式の数値解法を考えていきますが，その前に，与えられた関数の微分係数を近似する方法について考えましょう．以下，関数は必要な回数微分できるものとします．

$y = f(x)$ の $x = a$ における微分係数 $f'(a)$ の定義は

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

で定義されるのでした．一方で，高校で学んだように，

$$f'(a) = \lim_{h \to 0} \frac{f(a) - f(a-h)}{h} = \lim_{h \to 0} \frac{f(a+h) - f(a-h)}{2h}$$

といった等式も成立します．そこで，h を十分に小さな**正の**数とするとき，$f'(a)$ の近似式として

- **前進差分近似：** $\boldsymbol{f'(a) \approx \dfrac{f(a+h) - f(a)}{h}}$
- **後退差分近似：** $\boldsymbol{f'(a) \approx \dfrac{f(a) - f(a-h)}{h}}$
- **中心差分近似：** $\boldsymbol{f'(a) \approx \dfrac{f(a+h) - f(a-h)}{2h}}$

といった選択が考えられます．それぞれに利点があるのですが，以下では主に前進差分近似と中心差分近似を利用します．関数 $f(x)$ を $x = a$ を中心にテイラー展開して $f(a \pm h)$ を計算すれば，これらの近似式はそれぞれ

$$f'(a) = \frac{f(a+h) - f(a)}{h} + O(h), \qquad f'(a) = \frac{f(a+h) - f(a-h)}{2h} + O(h^2) \quad (11.4)$$

という性質を持つことがわかります[*4].

問題 11.1 式 (11.4) を示せ.

11.6　前進オイラー法

常微分方程式のもっとも基本的な解法として知られているのが次の**前進オイラー法** (forward Euler method) とよばれているものです[*5]. いま, $0 \le x \le L$ を満たす x に対し, 定理 11.3 の条件を満たす微分方程式

$$\frac{dy}{dx} = f(x, y); \quad y(0) = A \tag{11.5}$$

と十分に小さい正の数 h に対し, 前進差分近似を用いて

$$\frac{y(x+h) - y(x)}{h} \approx f(x, y(x)) \iff y(x+h) \approx y(x) + hf(x, y(x)).$$

と近似するのが基本的なアイディアです. 具体的には, 次のようなアルゴリズムとなります:

前進オイラー法

(1) 自然数 N に対し $h := L/N$ とおき, 区間 $[0, L]$ の N 等分点 $x_i = ih$ $(i = 0, 1, 2, \cdots, N)$ をとる.

(2) 各 $i = 0, 1, 2, \cdots, N-1$ に対し, $y(x_i)$ の値の近似値 y_i を漸化式

$$y_0 := A, \qquad y_{i+1} := y_i + h\, f(x_i, y_i)$$

によって定める.

(3) こうして定まる $\mathbb{R}^2$ 上の点 $(x_0, y_0) = (0, A), (x_1, y_2), \cdots, (x_N, y_N)$ を折れ線で結んだ関数を $Y_N(x)$ とおき, 関数 $y(x)$ の近似値として採用する.

このとき, 次が知られています:

[*4] 記号 $O(h)$ はランダウの記号. たとえば $A(h) = B(h) + O(h^p)$ と書いたら, 十分小さな h に対し $|A(h) - B(h)| \le M|h|^p$ が成り立つような h に依存しない定数 $M > 0$ が存在する, という意味です.

[*5] **コーシーの折れ線法**ともいいます. 原理的には, この方法で解の存在証明も与えることができます.

定理 11.4 (前進オイラー法の収束性) 微分方程式 (11.5) が滑らかな解 $y(x)$ をもつならば，関数 $Y_N(x)$ は $N \to \infty$ のとき解 $y(x)$ に誤差 $O(1/N)$ で一様収束する．すなわち，ある定数 C が存在し，

$$\max_{0 \le x \le L} |Y_N(x) - y(x)| \le \frac{C}{N}.$$

例 1（指数関数）　例題 11.1 の方程式 $y' = ky$ $(k \neq 0)$, $y(0) = A$ に対して $[0,1]$ 区間上で前進オイラー法を適用してみましょう．$h = 1/N$ より，

$$y_{i+1} = y_i + khy_i = (1 + kh)y_i \quad (0 \le i \le N-1)$$

となるので，$y_0 = y(0) = A$ より

$$y_i = A\,(1+kh)^i = A\left\{\left(1 + \frac{k}{N}\right)^{N/k}\right\}^{ki/N} = A\left\{\left(1 + \frac{k}{N}\right)^{N/k}\right\}^{k\,x_i}.$$

ここで $i = N$ とおくと，

$$y_N = A\,(1+kh)^N = A\left\{\left(1 + \frac{k}{N}\right)^{N/k}\right\}^{k} \to A\,e^k \quad (N \to \infty)$$

となるから，$Y_N(1) = y_N \to A\,e^k = y(1)$ $(N \to \infty)$．これは例題 11.1 の結果と合致しています[*6]．

Julia による実装　まずは例題 11.1 の微分方程式で $k = 1$ とした

$$\frac{dy}{dx} = -y, \qquad y(0) = 1.0$$

を区間 $[0,1]$ で数値的に解き，厳密解と比較してみましょう．結果はふつうの指数関数 $y = e^x$ となるはずです．基本的には，グラフ上の点を複数プロットして折れ線で結ぶことになります：

練習 11.1

```
using Plots     # グラフ描画用パッケージを読み込む
L = 1.0
N = 10
h = L/N         # x 方向の刻み幅
```

[*6] 一般に，任意の $x \in [0,1]$ と自然数 N に対し，$|x_{i_N} - x| \le 1/N$ を満たす x_{i_N} をひとつ選ぶことができます．このとき，$|Y_N(x_{i_N}) - A{\cdot}e^{kx}| = O(1/N)$ であることが証明できます．

```
5  x = [i*h for i = 0:N]   # x_i を配列として定義
6  y = zeros(N+1)          # y_i 用の配列を初期化
7  y[1] = 1.0              # 初期値 y_0 の値
8  for i = 1:N             # 前進オイラー法
9      y[i+1] = y[i] + h * y[i]
10 end
11 plot(x, y)               # 近似解のグラフを描画
12 plot!(x -> exp(x))       # 厳密解のグラフを追加
```

区間を N 分割するので，結ばれる点の数は $N+1$ 個（結ぶ線分の数は N 個）です．5 行目ではそのための x 座標 $N+1$ 個からなる配列を作っています．6 行目では近似解・厳密解用の y 座標をとりあえずすべて 0 を成分とする配列にして初期化しています．7 行目で近似解・厳密解それぞれの初期条件を設定しているのですが，**Julia では配列のインデックスが 1 から始まる**ため，理論上の $y_0, y_1, \cdots, y_N$ が配列の要素としてはそれぞれ添え字が 1 つ増えて `y[1]`，`y[2]`，`....` `y[N+1]` となります．x 座標についても同様で，理論上の $x_0, x_1, \cdots, x_N$ はそれぞれ `x[1]`，`x[2]`，`....` `x[N+1]` となりますから注意してください．11 行目の `plot(x,y)` によって $(x_0, y_0), (x_1, y_1), \cdots, (x_N, y_N)$ を折れ線で結んだグラフが描画されます．12 行目の `plot!(x -> exp(x))` では厳密解 $y = e^x$ のグラフを上で描画したグラフに付け加える形で描画するための命令です．

問題 11.2 $x = 1.0$ における近似値 $Y_N(1.0)$ と厳密解の値 $y(1.0) = e^{1.0} = e$ を比較せよ．さらに，N を増やして近似が良くなることを確認せよ．

一般化　汎用性を高めるために，区間 $[0, L]$ 上の常微分方程式

$$\frac{dy}{dx} = f(x, y), \quad y(0) = A$$

の解を求める関数を作成してみましょう．2 変数関数 `f`，区間の右端の値 `L`，初期値 `A`，さらに区間の分割数 `N` を入力すると，先のコードの例のように配列 `x` と `y` のペアを返すような関数にします：

練習 11.2

```
1  # 前進オイラー法
2  function forwardEuler(f, L, A, N)
```

```
        # f:関数，L: 区間終端，A: 初期値，N: 分割数
    h = L/N
    x = [i*h for i = 0:N]
    y = zeros(N+1)
    y[1] = A                # 初期条件 y_0 = A に相当
    for i = 1:N
        y[i+1] = y[i] + h * f(x[i], y[i])
    end
    return (x, y)           # 配列のペアを返す
end
# 具体例（ロジスティック関数）
f(s, t) = (1 - t) * t
(x, y) = forwardEuler(f, 5.0, 0.25, 10)
plot(x, y)
```

問題 11.3 上の具体例について，N = 5，10，20，40 のときのグラフをまとめて描画し比較せよ．

第12講　微分方程式 2：リープ・フロッグと連立微分方程式

今回も関数 $y = y(x)$ が関係式

$$\boldsymbol{\frac{dy}{dx} = f(x,\, y), \qquad y(0) = A} \tag{12.1}$$

を満たすものとし，2 変数関数 $(s,t) \mapsto f(s,\, t)$ が定義可能な範囲で C^1 級関数であるものとしましょう[*1]．このような関数は前回の定理 11.1（解の存在と一意性）の条件を満たすことがわかります．以下では，$0 \leq x \leq L$ において一意的な解 $y = y(x)$ の存在が理論的に保証されている状況で，その解を数値計算する方法を考えていきましょう．

12.1　リープ・フロッグ法

前回学んだ前進オイラー法は微分の前進差分近似

$$f'(a) = \frac{f(a+h) - f(a)}{h} + O(h)$$

を用いた方法でした．具体的には，十分大きな自然数 N に対し $h := L/N$ とおき，区間 $[0, L]$ の N 等分点 $x_i = ih$ $(i = 0,\, 1,\, 2,\, \cdots,\, N)$ における $y(x_i)$ の値の近似値 y_i を漸化式

$$\boldsymbol{y_0 := A, \qquad y_{i+1} := y_i + h\, f(x_i, y_i)}$$

によって定めるのでした．すなわち，$x = x_i$ において

$$\frac{dy}{dx}(x_i) \approx \frac{y_{i+1} - y_i}{h}, \qquad f(x_i, y(x_i)) \approx f(x_i, y_i)$$

と近似し，この右辺同士を等号とみなしたわけです．

では，これを中心差分近似

$$f'(a) = \frac{f(a+h) - f(a-h)}{2h} + O(h^2)$$

[*1] $(s,t) \mapsto f(s,\, t)$ が領域 D 上の C^1 **級関数**であるとは，変数 s と t に関して偏微分可能であり，s 偏導関数と t 偏導関数がともに連続関数となることをいうのでした．

に置き換えたらどうなるでしょうか．誤差が $O(h)$ から $O(h^2)$ に減少していることから，近似解の精度があがることも期待されます．そこで $x = x_i$ $(i = 1, 2, \cdots, N-1)$ において

$$\frac{dy}{dx}(x_i) \approx \frac{y_{i+1} - y_{i-1}}{2h}, \qquad f(x_i, y(x_i)) \approx f(x_i, y_i)$$

と近似すると，

$$y_0 := A, \qquad y_{i+1} := y_{i-1} + 2\,h\,f(x_i, y_i) \quad (i \geq 1)$$

という関係式が得られます．しかしよく見ると，これでは $i = 1$ のときの y_1 が定まらないので，そこは前進差分で置き換えた

$$\boldsymbol{y_0 := A, \quad y_1 := y_0 + hf(x_0, y_0), \quad y_{i+1} := y_{i-1} + 2\,h\,f(x_i, y_i) \quad (i \geq 1)}$$

という漸化式を採用することにします．これを**リープ・フロッグ法** (leapfrog method, 蛙跳び法) といいます．改めて，アルゴリズムとしてまとめておきましょう．

リープ・フロッグ法

(1) 自然数 N に対し $h := L/N$ とおき，区間 $[0, L]$ の N 等分点 $x_i = ih$ $(i = 0, 1, 2, \cdots, N)$ をとる．
(2) 各 $i = 0, 1, 2, \cdots, N-1$ に対し，$y(x_i)$ の値の近似値 y_i を漸化式

$$\boldsymbol{y_0 := A, \quad y_1 := y_0 + hf(x_0, y_0), \quad y_{i+1} := y_{i-1} + 2\,h\,f(x_i, y_i)}$$

によって定める．
(3) こうして定まる $\mathbb{R}^2$ 上の点 $(x_0, y_0) = (0, A), (x_1, y_1), \cdots, (x_N, y_N)$ を折れ線で結んだ関数を $Y_N(x)$ とおき，関数 $y(x)$ の近似値として採用する．

このとき，次が知られています：

定理 12.1 (リープ・フロッグ法の収束性) 微分方程式 (13.1) が滑らかな解 $y(x)$ をもつならば，関数 $Y_N(x)$ は $N \to \infty$ のとき解 $y(x)$ に誤差 $O(1/N^2)$ で一様収束する．すなわち，ある定数 C が存在し，

$$\max_{0 \leq x \leq L} |Y_N(x) - y(x)| \leq \frac{C}{N^2}.$$

Julia による実装

問題 12.1 区間 $[0, L]$ 上の常微分方程式

$$\frac{dy}{dx} = f(x, y), \quad y(0) = A$$

の解をリープ・フロッグ法によって計算する関数を作成したい．前回作成した関数 `forwardEuler(f, L, A, N)` を修正して，2 変数関数 `f`，区間の右端の値 `L`，初期値 `A`，さらに区間の分割数 `N` を入力すると，先のコードの例のように配列 `x` と `y` のペアを返すような関数とせよ．

問題 12.2 区間 $[0, 1]$ 上で常微分方程式

$$\frac{dy}{dx} = y, \quad y(0) = 1$$

の解を前進オイラー法およびリープ・フロッグ法によって計算し，$x = 1$ での値（これは $e^1 = e$ となる）への収束速度の違いを以下の方法で比較せよ．

(1) 厳密解 $y = e^x$ も含めたグラフを描いて比較する．
(2) 下のような表を作成し比較する．

N	前進オイラー	リープ・フロッグ	リープ・フロッグの誤差
2	2.25	2.5	-0.2182818284590451
4	2.44140625	2.65625	-0.06203182845904509
8	2.565784513950348	2.7022171020507812	-0.01606472640826384
16	2.6379284973665995	2.714229131668647	-0.004052696790397992
32	2.6769901293781833	2.717266343778124	-0.0010154846809209417
64	2.6973449525651	2.7180278124356687	$-0.0002540160233763799 6$
128	2.7077390196880193	2.718218315392857	$-6.351306618812558 \times 10^{-5}$
256	2.712991624253433	2.7182659496261175	$-1.5878832927640474 \times 10^{-5}$
512	2.71563200016899	2.7182778587154126	$-3.969743632481482 \times 10^{-6}$
1024	2.7169557294664357	2.7182808360209263	$-9.92438118796457 \times 10^{-7}$

12.2　連立常微分方程式

$y = y(x) = \sin x$ が満たす微分方程式

$$\frac{d^2y}{dx^2}(x) = -y(x), \qquad y(0) = 0, \qquad \frac{dy}{dx}(0) = 1$$

を区間 $[0, 2\pi]$ で数値的に解くことを考えましょう．これは今まで解いてきた 式 (13.1) の形ではありませんが，ちょっとしたトリックでこれまでの手法が適用できるようになります．

そのまえに，上の方程式は $y = y(x)$ の形で考えていましたが，変数を t と u に置き換えて $u = u(t)$ の形にした

$$\boldsymbol{\frac{d^2u}{dt^2}(t) = -u(t), \quad u(0) = 0, \quad \frac{du}{dt}(0) = 1} \tag{12.2}$$

を考えることにしましょう．さてその「トリック」とは，$u(t)$ の導関数 $v(t) = \dfrac{du}{dt}(t)$ を考えることなのです．このとき，上の方程式は **1 階連立常微分方程式**

$$\begin{cases} \boldsymbol{\dfrac{du}{dt}(t) = v(t), \qquad u(0) = 0} \\ \boldsymbol{\dfrac{dv}{dt}(t) = -u(t), \qquad v(0) = 1} \end{cases} \tag{12.3}$$

となりますから，たとえば前進オイラー法のアイディアで

(1) 自然数 N に対し $h := L/N$ とおき，区間 $[0, L]$ の N 等分点 $t_i = ih$ $(i = 0, 1, 2, \cdots, N)$ をとる．

(2) 各 $i = 0, 1, 2, \cdots, N-1$ に対し，$u(t_i)$ と $v(t_i)$ それぞれの近似値 u_i と v_i を漸化式

$$\begin{cases} \boldsymbol{u_0 := 0, \quad u_{i+1} := u_n + h\, v_i} \\ \boldsymbol{v_0 := 1, \quad v_{i+1} := v_i - h\, u_i} \end{cases}$$

によって定める．

(3) こうして定まる $\mathbb{R}^2$ 上の点 $(t_0, u_0) = (0, 0), (t_1, u_1), \cdots, (t_N, u_N)$ を折れ線で結んだ関数を $U_N(t)$ とおき，関数 $u(t)$ の近似値として採用する．

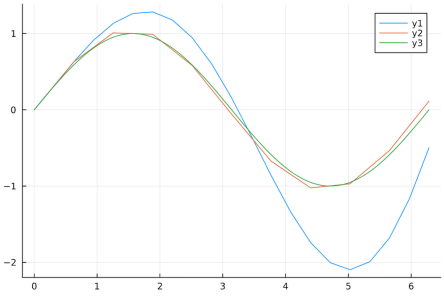
ということが考えられます．

練習 12.1

```
function mySine(L, N) # L: 区間右端, N: 分割数
    h = L/N
    t = zeros(N+1)     # 配列 t の初期化
    u = zeros(N+1)     # 配列 u の初期化
    v = zeros(N+1)     # 配列 v の初期化
    t[1] = 0.0         # t_0 に相当する値
    u[1] = 0.0         # u_0 の初期条件
    v[1] = 1.0         # v_0 の初期条件
    for i = 1:N           # 前進オイラー法を適用
        t[i+1] = t[i] + h
        u[i+1] = u[i] + h * v[i]
        v[i+1] = v[i] - h * u[i]
    end
    return (t, u, v)
end
# 具体例
(t1, u1, v1) = mySine(2*pi, 100)
plot(t1, u1)
plot!(x -> sin(x))
```

問題 12.3 mySine(L,N) 関数を改良し，リープ・フロッグ法を用いた mySineLeapFrog(L,N) 関数を作成せよ．さらに，同じ N に対して結果のグラフを描き比較せよ．

次の図は分割数 $N = 20$ に対して厳密解（緑），前進オイラー法（青），リープ・フロッグ法（赤）のグラフを描かせたものである．



問題 12.4
前進オイラー法を用いた (`t1, u1, v1) = mySine(2*pi, 20)`) と，リープ・フロッグ法を用いた (`t2, u2, v2) = mySineLeapFrog(2*pi, 20)`) の結果は，それぞれ $u(t) = \sin t$ と $v(t) = \cos t$ の近似を与えている．そこで，`plot(v1, u1)` と `plot(v2, u2)` の結果と単位円のパラメーター表示 $t \mapsto (\cos t, \sin t)$ を比較せよ．

12.3　研究：ロトカ-ヴォルテラ方程式

ある地域における時間 t でのウサギ（被食者）とキツネ（捕食者）の個体数をそれぞれ $u = u(t)$, $v = v(t)$ としたとき，これらの関数は次の**ロトカ-ヴォルテラ方程式** (Lotka-Volterra equations) でモデル化できます：

$$u' = u(\alpha - \beta v), \qquad v' = -v(\gamma - \delta u).$$

ただし，$\alpha, \beta, \gamma, \delta$ は正の定数です．たとえば餌（草）が無限にありかつ天敵のキツネがいなければ，ウサギは指数関数的に増えるのでウサギの個体数は微分方程式 $u' = \alpha u$ で表されるのでした（前回）．しかし実際には，捕食者であるキツネの個体数 v に比例してその増加係数 α は押さえられるので，$\alpha - \beta v$ と修正すべきです．この値が正であればウサギは増加し，負であればウサギは減少します．これが方程式 $u' = u(\alpha - \beta v)$ の解釈です．

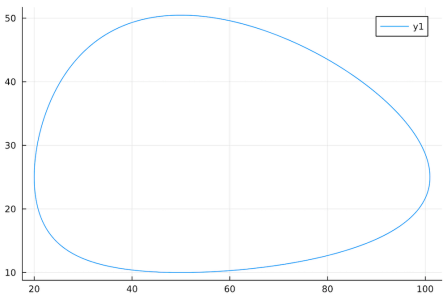
キツネのほうは $v' = \delta uv - \gamma v$ と見たほうがわかりやすくなります．キツネは主食であるウサギの個体数 u とキツネ自身の個体数 v に比例して増加しますが (δuv の項)，一方で自然死や移動にともなう自然減少も個体数に比例して存在します（$-\gamma v$ の項)．

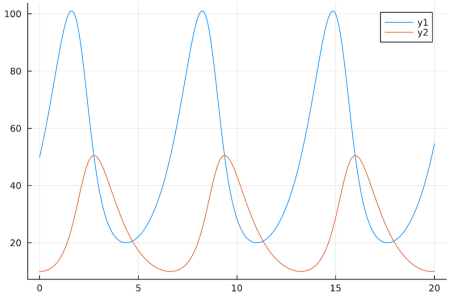
この方程式の解の挙動をグラフ化してみましょう．まずはロトカ-ヴォルテラ方程式を定数 $\alpha = \gamma = 1$, $\beta = 0.04$, $\delta = 0.02$, 初期値 $u(0) = 50$, $v(0) = 10$, 変域 $0 \le t \le 10$ として解き，解の軌道を uv 平面に図示してみます：

練習 12.2

```
# Lotka-Volterra 方程式（リープ・フロッグ法）
function LotkaVolterra(L, A, B, N)
    f(x, y) =   x * (1 - 0.04*y)
    g(x, y) = - y * (1 - 0.02*x)
    h = L/N
    t = zeros(N+1)
    u = zeros(N+1)
    v = zeros(N+1)
    t[1] = 0.0
    u[1] = A
    v[1] = B
    t[2] = h
    u[2] = u[1] + h * f(u[1], v[1])
    v[2] = v[1] + h * g(u[1], v[1])
    for i = 2:N
        t[i+1] = t[i-1] + 2*h
        u[i+1] = u[i-1] + 2*h * f(u[i], v[i])
        v[i+1] = v[i-1] + 2*h * g(u[i], v[i])
    end
    return (t, u, v)
end
# 具体例
end
(t, u, v) = LotkaVolterra(10.0, 50.0, 10.0, 500)
plot(u,v)
# gr = plot(t, u)  # u(t) と v(t) のグラフを
# plot!(gr, t, v)  # 比較したいとき
```

閉曲線（ループ）が得られました（左下図）．じつは初期値が適当な条件をみたすとき，ロトカ-ヴォルテラ方程式は周期解をもつことが知られています．すなわち，ウサギとキツネの個体数は一定の周期で増減を繰り返わけです．右下の図では $u(t)$（ウザギ，青）と $v(t)$（キツネ，赤）のグラフを比較したものです（上のコードで，最後の 2 行をコメントアウトして得られるグラフ）．





第13講　微分方程式 3：ルンゲ・クッタ法

前回に引き続き，関数 $y = y(x)$ は関係式

$$\frac{dy}{dx} = f(x, y), \qquad y(0) = A \tag{13.1}$$

を満たすものとし，2 変数関数 $(s,t) \mapsto f(s, t)$ が定義可能な範囲で C^1 級関数であるものとします．このような関数は前々回の定理 11.1（解の存在と一意性）の条件を満たすので，以下では $0 \leq x \leq L$ において一意的な解 $y = y(x)$ の存在が理論的に保証されている状況であるとします．

これまで「前進オイラー法」,「リープ・フロッグ法」と学んできましたが，今回はさらに精度の高い，「ルンゲ・クッタ法」とよばれる方法を学びましょう．

13.1　ホイン法（2 段のルンゲ・クッタ法）

「ルンゲ・クッタ法」には，近似の手間に応じた「段数」(stage) とよばれるランクがあり，「1 段のルンゲ・クッタ法」はちょうど「前進オイラー法」そのものになっていて，「2 段」「3 段」と上がるごとにより高い精度で厳密解の近似ができるように作られています．

ここでは「2 段のルンゲ・クッタ法」を実際に求めてみましょう．

いつものように，十分大きな自然数 N に対し $h := L/N$ とおき，区間 $[0, L]$ の N 等分点 $x_i = ih$ $(i = 0, 1, 2, \cdots, N)$ それぞれにおいて，$y(x_i)$ の値を近似した値 y_i を与えていきます．そのために，y_i から y_{i+1} を計算する漸化式を作りたいわけですが，それをまず

$$\begin{cases} k_1 = h\, f(x_i, y_i) \\ k_2 = h\, f(x_i + ph,\, y_i + qk_1) \\ y_{i+1} = y_i + ak_1 + bk_2 \end{cases} \tag{13.2}$$

とおき，適切に係数 p, q, a, b を選んでいくことにしましょう．

問題 13.1 式 (13.2) が前進オイラー法と同じとなるような p, q, a, b の値を求めよ．

テイラー展開　いま，十分に滑らかな解 $y = y(x)$ が存在したとき，$x_i = ih$ において

その関数をテイラー展開（漸近展開）すると，$x = x_i + \Delta x$ に対して

$$y(x) = y(x_i + \Delta x) = y(x_i) + y'(x_i)\Delta x + \frac{1}{2!}y''(x_i)\Delta x^2 + O(\Delta x^3)$$

となります．よって，$x = x_{i+1} = x_i + h$ を代入すれば

$$y(x_{i+1}) = y(x_i + h) = y(x_i) + y'(x_i)h + \frac{1}{2!}y''(x_i)h^2 + O(h^3).$$

ここで，式 (13.1) より

$$y'(x_i) = \left.\frac{dy}{dx}\right|_{x=x_i} = \left. f(x, y(x))\right|_{x=x_i} = f(x_i, y(x_i))$$

であり，同様にして

$$y''(x_i) = \left.\frac{d}{dx}\left(\frac{dy}{dx}\right)\right|_{x=x_i} = \left.\frac{d}{dx} f(x, y(x))\right|_{x=x_i}$$

となります．微分の計算をすると，

$$\frac{d}{dx} f(x, y(x)) = \frac{dx}{dx} f_x(x, y(x)) + \frac{dy}{dx} f_y(x, y(x)) = f_x(x, y(x)) + f(x, y(x)) f_y(x, y(x))$$

（ただし，f_x と f_y はそれぞれ第 1 成分，第 2 成分による偏導関数）となるので，

$$y''(x_i) = f_x(x_i, y(x_i)) + f(x_i, y(x_i)) f_y(x_i, y(x_i))$$

を得ます．以上をまとめると，

$$y(x_{i+1}) = y(x_i) + hf(x_i, y(x_i)) + \frac{h^2}{2!}\Big\{ f_x(x_i, y(x_i)) + f(x_i, y(x_i)) f_y(x_i, y(x_i)) \Big\} + O(h^3)$$

となりました．もし，$y(x_i)$ の近似値としてすでに y_i が計算できていると仮定すると，これから

$$y(x_{i+1}) \approx y_i + hf(x_i, y_i) + \frac{h^2}{2!}\Big\{ f_x(x_i, y_i) + f(x_i, y_i) f_y(x_i, y_i) \Big\} + O(h^3) \qquad (13.3)$$

という近似計算が示唆されます．

係数の決定　上の近似式を目標にして，式 (13.2) の係数を定めていきます．関数 $f(x, y)$ の $(x, y) = (x_i, y_i)$ における 1 次テイラー展開から

$$f(x_i + ph, y_i + qh) = f(x_i, y_i) + f_x(x_i, y_i) \cdot ph + f_y(x_i, y_i) \cdot qh + O(h^2)$$

となるので，

$$\begin{aligned} k_2 &= hf(x_i + ph, y_i + qh) \\ &= h\, f(x_i, y_i) + h^2\{p\, f_x(x_i, y_i) + q\, f_y(x_i, y_i)\} + O(h^3). \end{aligned}$$

よって式 (13.2) は

$$\begin{aligned} y_{i+1} &= y_i + ak_1 + bk_2 \\ &= y_i + (a+b)\,h\,f(x_i, y_i) + h^2\{b\,p\,f_x(x_i, y_i) + b\,q\,f_y(x_i, y_i)\} + O(h^3) \end{aligned}$$

となります．式 (13.3) と比較すると，

$$a + b = 1, \qquad b\,p = b\,q = \frac{1}{2}$$

とするのが良さそうです．そこで，$a = b = \dfrac{1}{2}$，$p = q = 1$ としたのが**ホイン法** (Heun's method) もしくは**改良オイラー法** (improved Euler's method) とよばれるものです：

ホイン法

$$\begin{cases} \boldsymbol{k_1 = h\,f(x_i, y_i)} \\ \boldsymbol{k_2 = h\,f(x_i + h,\, y_i + k_1)} \\ \boldsymbol{y_{i+1} = y_i + \dfrac{1}{2}(k_1 + k_2)} \end{cases} \tag{13.4}$$

13.2　一般のルンゲ・クッタ法

ホイン法と同じ考え方で，

$$\begin{aligned} k_1 &= h\,f(x_i, y_i) \\ k_2 &= h\,f(x_i + p_2 h,\, y_i + q_{21} k_1) \\ k_3 &= h\,f(x_i + p_3 h,\, y_i + q_{31} k_1 + q_{32} k_2) \\ k_4 &= h\,f(x_i + p_4 h,\, y_i + q_{41} k_1 + q_{42} k_2 + q_{43} k_3) \\ &\vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ k_s &= h\,f(x_i + p_s h,\, y_i + q_{s1} k_1 + q_{s2} k_2 + \cdots + q_{s,s-1} k_{s-1}) \\ y_{i+1} &= y_i + h\{a_1 k_1 + a_2 k_2 + \cdots + a_s k_s\} \end{aligned}$$

の形の漸化式を構成し，できるだけ $y(x_i + h)$ の $x = x_i$ におけるテイラー展開の形に近づけた一連の公式を（s 段の）**ルンゲ・クッタ法** (the Runge-Kutta method) といいます．たとえば，前進オイラー方は 1 段のルンゲ・クッタ法であり，ホイン法は 2 段のルンゲ・クッタ法になっています．

通常，「ルンゲ・クッタ法」としてもっとも有名かつ実用的なのが，次の 4 段の公式です[*1]：

4 段 4 次のルンゲ・クッタ法

$$\begin{cases} \boldsymbol{k_1 = h\, f(x_i, y_i)} \\ \boldsymbol{k_2 = h\, f(x_i + h/2,\, y_i + k_1/2)} \\ \boldsymbol{k_3 = h\, f(x_i + h/2,\, y_i + k_2/2)} \\ \boldsymbol{k_4 = h\, f(x_i + h,\, y_i + k_3)} \\ \boldsymbol{y_{i+1} = y_i + \dfrac{1}{6}(k_1 + 2\,k_2 + 2\,k_3 + k_4)} \end{cases} \tag{13.5}$$

ここで，y_{i+1} と $y(x_{i+1})$ の誤差は $O(h^5)$ となります．それが $h^{-1} = N$ に比例する数だけ集積するので，方程式全体で見た時の厳密解からの誤差は $O(h^4)$ 程度だと期待されます．式 (13.5) を実際に導出するのは大変な作業なので省略しますが，基本的なアイディアはホイン法のときと同じです．

Julia による実装　区間 $[0, L]$ 上の常微分方程式

$$\frac{dy}{dx} = f(x, y), \quad y(0) = A$$

の解を 4 段ルンゲ・クッタ法によって計算する関数を作成しましょう．

練習 13.1

```
# 4 段ルンゲ・クッタ法
function RungeKutta(f, L, A, N)
        # f: 関数, L: 区間終端, A: 初期値, N: 分割数
    h = L/N
    x = [i*h for i = 0:N]  # x_i の定義
    y = zeros(N+1)         # y_i の初期化
    y[1] = A               # 初期値 y_0 の設定
    for i = 1:N
        k1 = h * f(x[i], y[i])
        k2 = h * f(x[i] + h/2, y[i] + k1/2)
        k3 = h * f(x[i] + h/2, y[i] + k2/2)
```

[*1] 「古典的ルンゲ・クッタ法」とよばれることもあります．

```
        k4 = h * f(x[i] + h, y[i] + k3)
        y[i+1] = y[i] + (k1 + 2*k2 + 2*k3 + k4)/6
    end
    return (x,y)
end
# 具体例（指数関数）
f(s, t) = t
(x, y) = RungeKutta(f, 1.0, 1.0, 10)
plot(x, y)
plot!(x -> exp(x))  # 厳密解とグラフで比較
```

問題 13.2 区間 $[0,1]$ 上で常微分方程式

$$\frac{dy}{dx} = y, \quad y(0) = 1$$

の解を $N = 2, 4, 8, \cdots$ に対しリープ・フロッグ法およびルンゲ・クッタ法によって計算し，$x = 1$ での値（これは $e^1 = e$ となる）への収束速度を比較せよ．

N	リープ・フロッグ	ルンゲ・クッタ	ルンゲ・クッタの誤差
2	2.5	2.71734619140625	-0.0009356370527950908
4	2.65625	2.718209939201323	$-7.188925772227961 \times 10^{-5}$
8	2.7022171020507812	2.7182768444167347	$-4.9840423104186016 \times 10^{-6}$
16	2.714229131668647	2.7182815003405856	$-3.2811845951385976 \times 10^{-7}$
32	2.717266343778124	2.718281807411193	$-2.1047851905819925 \times 10^{-8}$
64	2.7180278124356687	2.718281827126323	$-1.3327219328118645 \times 10^{-9}$
128	2.718218315392857	2.718281828375204	$-8.384093419522287 \times 10^{-11}$
256	2.7182659496261175	2.718281828453785	$-5.260236690673992 \times 10^{-12}$
512	2.7182778587154126	2.7182818284587165	$-3.2862601528904634 \times 10^{-13}$
1024	2.7182808360209263	2.7182818284590256	$-1.9539925233402755 \times 10^{-14}$

13.3　研究：ローレンツ・アトラクター

次の連立微分方程式は**ローレンツ方程式** (Lorenz equation) とよばれ，いわゆる「カオス」を生成する例として，最初に発見されたもののひとつです：

$$\left\{\begin{array}{lcl} x' &=& \sigma(y-x) \\ y' &=& x(\rho-z)-y \\ z' &=& xy-\beta z \end{array}\right.$$

ただし，σ, ρ, β は定数です．解曲線が集積する集合は極めて複雑な図形となることが知られており，この方程式を発見した気象学者ローレンツ（Edward N. Lorenz）にちなんで**ローレンツ・アトラクター** (Lorenz attractor) とよばれています．ここでは，このローレンツの原論文[*2] にあるパラメーター $\sigma = 10, \rho = 28, \beta = 8/3$ を選んで，解曲線（アトラクターを近似していると考えられる）を描かせてみましょう：

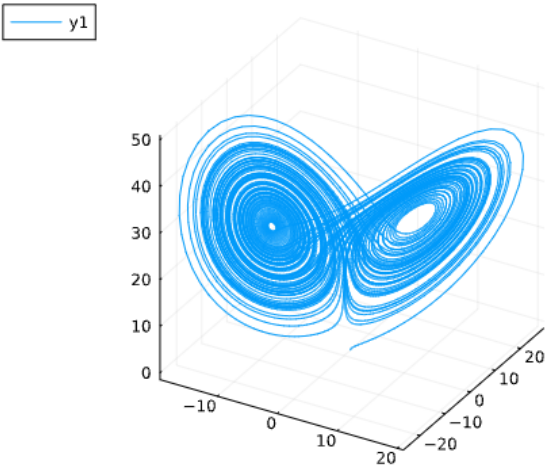
練習 13.2

```
function LorenzAttractor(L, A, B, C, N)
    f1(s, t, u) = 10.0 * (t - s)          # 関数の設定
    f2(s, t, u) = s * (28.0 - u) - t
    f3(s, t, u) = s * t - (8.0/3) * u
    h = L/N
    x = zeros(N+1); y = zeros(N+1); z = zeros(N+1)
    x[1] = A; y[1] = B; z[1] = C          # 初期値の設定
    for i = 1:N
        k1 = h * f1(x[i], y[i], z[i])     # ルンゲ・クッタ法の計算
        l1 = h * f2(x[i], y[i], z[i])
        m1 = h * f3(x[i], y[i], z[i])
        k2 = h * f1(x[i] + k1/2, y[i] + l1/2, z[i]+ m1/2)
        l2 = h * f2(x[i] + k1/2, y[i] + l1/2, z[i]+ m1/2)
        m2 = h * f3(x[i] + k1/2, y[i] + l1/2, z[i]+ m1/2)
        k3 = h * f1(x[i] + k2/2, y[i] + l2/2, z[i]+ m2/2)
```

[*2] E.N.Lorenz. Determinastic nonperiodic flow. *Journal of the Atomospheric Sciences*, **20**(1963), 130 – 141.

```
        l3 = h * f2(x[i] + k2/2, y[i] + l2/2, z[i]+ m2/2)
        m3 = h * f3(x[i] + k2/2, y[i] + l2/2, z[i]+ m2/2)
        k4 = h * f1(x[i] + k3, y[i] + l3, z[i] + m3)
        l4 = h * f2(x[i] + k3, y[i] + l3, z[i] + m3)
        m4 = h * f3(x[i] + k3, y[i] + l3, z[i] + m3)
        x[i+1] = x[i] + (k1 + 2*k2 + 2*k3 + k4)/6
        y[i+1] = y[i] + (l1 + 2*l2 + 2*l3 + l4)/6
        z[i+1] = z[i] + (m1 + 2*m2 + 2*m3 + m4)/6
    end
    return (x, y, z)
end
# 具体例
(x, y, z) = LorenzAttractor(100.0, 1.0, 0.0, 0.0, 10000)
plot(x, y, z)
```

「アトラクター」というのは「ひきつける」を意味する動詞 attract から来ています．初期値を変化させても解曲線は最終的に同じ集合に集積する（ひきつけられる）のです．イメージとしては，アトラクターに解曲線という「糸」が「巻きついていく」ような感じでしょうか．実際に初期値を変えてみて，解がどのような振る舞いをするか試してみると面白いと思います．


y1
50
40
30
20
10
0
−10
0
10
20
20
10
0
−10
−20