第6章　数値計算

コンピュータに数学的な問題を解かせることは多い。例として $f(x) = \dfrac{x}{(x+1)(x+2)}$ という関数が与えられたときに、$f(x)$ を $x=0$ から $x=1$ まで積分した $\displaystyle\int_0^1 f(x)dx$ の値を求める問題を考えてみよう。

人間がこれを解く場合、$f(x)$ の積分形を求めた上で、その式の計算をする。つまり、$\displaystyle\int f(x)dx = 2\log(x+2) - \log(x+1)$ $(= g(x)$ とおく$)$ と解いて、$\displaystyle\int_0^1 f(x)dx = g(1) - g(0) = \log(9/8)$ のように答を得る方法である。

コンピュータを使っても、同じように微分・積分や代数演算の規則を使って問題を解かせることができる。このような解き方は**数式処理**と呼ばれ、現在では学校で習う程度の数学の問題ならば自動的に解けるようになっている。下の画面は Mathematica という数式処理ソフトウェアで例の問題を解かせたものである。(In と書かれた行が入力した式で、Out と書かれた行がその答である。)

```
In[54]:= f[x_] = x / ((x + 1) (x + 2))

Out[54]= x / ((1 + x) (2 + x))

In[56]:= g[x_] = Integrate[f[x], x]

Out[56]= -Log[1 + x] + 2 Log[2 + x]

In[57]:= g[1] - g[0]

Out[57]= -3 Log[2] + 2 Log[3]
```

関数 f を定義し

その積分を求め g とする

求められた式

ただし、問題によっては数式処理をした結果が簡単な式では表わせないような場合があるので、この方法はいつも使えるとは限らない。例えば Mathematica では $\displaystyle\int \frac{\sin x}{\log x} dx$ の答を求めることはできない。

```
In[60]:= h[x_] = Integrate[Sin[x] / Log[x], x]

Out[60]= ∫ Sin[x] / Log[x] dx
```

(Out と書かれた行が積分記号を使った式のままになっているのは、積分形を見つけられなかったことを意味している。)

自然科学や工学における数学的問題は、数式処理では解けないことがめずらしくないため、反復計算によって近似的な解を求める**数値計算**と呼ばれる方法がとられることが多い。この章では数値計算によって問題を解く方法をいくつか紹介し、その際に必ず生じる誤差の問題についても触れる。

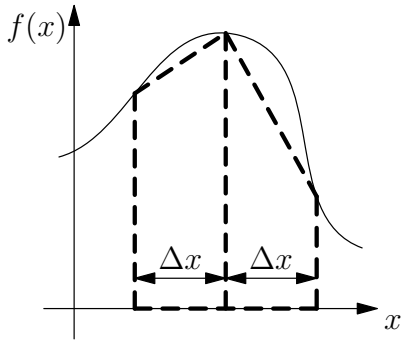
6.1　数値積分

まずは数値計算による積分から見てゆこう。関数 $f(x)$ が与えられたときにその定積分 $\int_{x_s}^{x_e} f(x)dx$ を求める問題である。

$f(x)$ のある区間の積分値を求めるということは、$f(x)$ のグラフを描き、グラフによって囲まれる領域の面積を求めることに相当する。$f(x)$ を実際に計算することによってグラフの高さを求めることはできるので、これを使って近似的な面積を求めるのが数値積分の方法である。この際、近似のやり方には**台形公式**や**シンプソン公式**などと呼ばれる方法が知られている。

6.1.1　台形公式

台形公式は、下図のように積分する範囲を幅 δx の区間に分け、各区間の領域を台形に近似して面積を求める方法である。



別の言い方をすると、関数を $f(x)$ を一定間隔の折れ線によって近似て、その関数を積分していることになる。

x_s から x_e までの積分範囲を n 等分した場合の近似式は次のようになる。まず、区間の幅を $\Delta x = \dfrac{x_e - x_s}{n}$ とおく。(左端を 0 番目とした)i 番目の区間の x 座標は $x_s + i\Delta x$ から $x_s + (i+1)\Delta x$ までであるので、この区間の台形の面積は

$$\frac{1}{2}\left\{f(x_s + i\Delta x) + f(x_s + (i+1)\Delta x)\right\}\Delta x$$

である。従って、積分値の近似値は x_s から x_e までの合計

$$\int_{x_s}^{x_e} f(x)dx \quad \simeq \quad \sum_{i=0}^{n-1} \frac{1}{2}\left\{f(x_s + i\Delta x) + f(x_s + (i+1)\Delta x)\right\}\Delta x \quad (6.1)$$

$$= \Delta x \left\{ \frac{f(x_s)+f(x_e)}{2} + \sum_{i=1}^{n-1} f(x_s + i\Delta x) \right\} \qquad (6.2)$$

となる。

式 6.2 を単純な繰り返しによって計算するのが以下の関数 `trapezoid(xs,xe,n)` である。`xs`, `xe` は積分範囲の両端の x の値を、`n` は分割数である。またこのプログラムでは、積分される関数として $f(x) = \dfrac{x}{(x+1)(x+2)}$ も同時に定義している。

```
1  def f(x)
2    x/((x+1.0)*(x+2.0))
3  end
4
5  def trapezoid(xs,xe,n)
6    deltax = (xe-xs)*1.0/n
7    sum = (f(xs)+f(xe))/2.0
8    for i in 1..(n-1)
9      sum = sum + f(xs+i*deltax)
10   end
11   deltax * sum
12 end
```

ファイル 6.1: `trapezoid.rb`

以下は、0 から 1 までの範囲を 100 分割して近似した場合の画面である。

```
1  irb(main):005:0> trapezoid(0,1,100)
2  => 0.11777910054096
3  irb(main):006:0> def g(x)
4  irb(main):007:1>   2*log(2+x)-log(1+x)
5  irb(main):008:1> end
6  => nil
7  irb(main):009:0> g(1)-g(0)
8  => 0.117783035656384
```

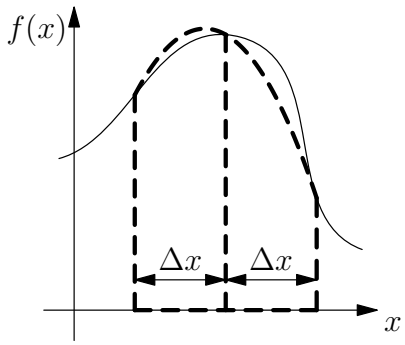
3 行目以下は、解析的に積分した $g(x) = \displaystyle\int f(x)dx = 2\log(x+2) - \log(x+1)$ を使って積分値を求めている。`trapezoid` の答が小数点以下 4 桁まで一致していることが分かる。

練習 6.1 (台形公式の正確さ) `trapezoid(0,1,n)` と $g(1)-g(0)$ の計算値は小数点以下何桁まで一致しているかを調べよ。8 桁目まで一致させるために

は n はどのくらい大きくなければいけないか、n = 100, 1000, 10000, . . . と変化させて調べよ。積分区間が $x = -0.9$ から $x = 0$ までの場合はどうか。

6.1.2　Simpson 公式

台形公式はグラフを折れ線、つまり δx の区間を直線で近似していた。Simpson 公式は直線のかわりに 2 次曲線で近似するものである。



x_s から x_e までの積分範囲を n 等分した場合の近似式は次のようになる。2 次式での近似のために 3 点の値を用いるので、近似する区間の**半分**の幅を $\Delta x = \dfrac{x_e - x_s}{2n}$ とおく。左端を 0 番目とした i 番目の区間の x 座標は $x_s + 2i\Delta x$ から $x_s + (2i+2)\Delta x$ までであり、さらに区間の中央の $x = (2i+1)\Delta x$ における値を用いて 2 次式を作り、それの積分を求めるとこの区間の面積は

$$\frac{1}{3}\left\{f(x_s + 2i\Delta x) + 4f(x + (2i+1)\Delta x) + f(x_s + (2i+2)\Delta x)\right\}\Delta x \quad (6.3)$$

となる。(この式の導出は後で述べる。) 従って積分値の近似値は x_s から x_e までの合計

$$\begin{aligned}
&\int_{x_s}^{x_e} f(x)dx \\
&\simeq \sum_{i=0}^{n-1} \frac{1}{3}\left\{ \begin{array}{r} f(x_s + 2i\Delta x) + 4f(x_s + (2i+1)\Delta x) \\ +f(x_s + (2i+2)\Delta x) \end{array} \right\}\Delta x \\
&= \frac{\Delta x}{3}\left\{ \begin{array}{l} f(x_s) + 4f(x_s + \Delta x) + f(x_e) \\ +\displaystyle\sum_{i=1}^{n-1}\left(2f(x_s + 2i\Delta x) + 4f(x_s + (2i+1)\Delta x)\right) \end{array} \right\} \quad (6.4)
\end{aligned}$$

となる。

練習 6.2 (Simpson 公式による積分)　a) 式 6.4 に従って積分を行う関数 `simpson(xs,xe,n)` を定義せよ。(注意: 台形公式の場合と違って Δx が積分範囲の $1/2n$ になっている。)

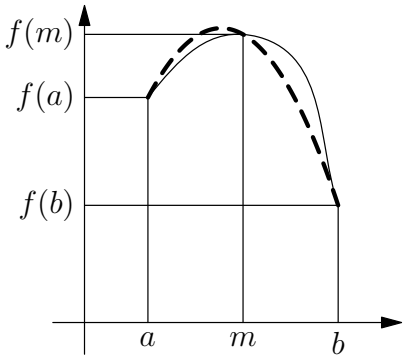
b) 練習 6.1 と同様に `simpson(xs,xe,n)` の解析的な積分値との違いを調べよ。

```
irb(main):011:0> simpson(0,1,100)
=> 0.117783035638943
```

Lagrange 補間と Simpson 公式 *

式 6.3 の導出方法を説明する。いま、区間の両端を a, b, 区間中の 1 点を m だとする。$f(a), f(m), f(b)$ の 3 点を通るような Lagrange 補間式 $P(x)$ は次のような 2 次関数になる。

$$\begin{aligned} P(x) &= \frac{(x-b)(x-m)}{(a-b)(a-m)}f(a) + \frac{(x-a)(x-b)}{(m-a)(m-b)}f(m) \\ &\quad + \frac{(x-a)(x-m)}{(b-a)(b-m)}f(b) \end{aligned} \tag{6.5}$$



ここで m を a, b のちょうど中間、つまり $m = \dfrac{b+a}{2}$ とした上で $P(x)$ を $x = a$ から b まで積分すると

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

となり、$a = x, \quad b = x + 2\Delta x, \quad m = x + \Delta x$ とおくと

$$\int_a^b P(x)dx = \frac{1}{3}\{f(x) + 4f(x+\Delta x) + f(x+2\Delta x)\}\Delta x$$

を得る。

6.1.3　Monte Carlo 法による積分

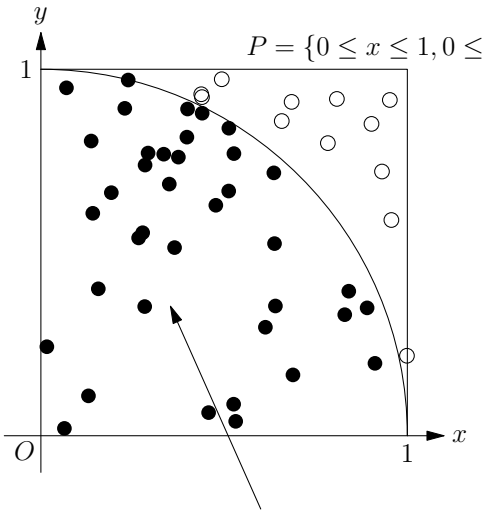
積分される関数が高次元の場合には Monte Carlo 法を用いた数値積分を行うことがある。

ここでは簡単のために 1 次元の関数を積分して四分円の面積を求める問題で説明する。つまり、

$$f(x) = \sqrt{1 - x^2}$$

を $x = 0$ から 1 まで積分することを考える。

積分範囲における $f(x)$ のグラフは下図のようになっている。



(黒い小円が $y < f(x)$ を満たしているものである)

まず、積分範囲の $f(x)$ を含むような長方形の領域

$$P = \{0 \le x \le 1, 0 \le y \le 1\}$$

に n 個の点をデタラメに配置する (図の黒い小円と白い小円)。次にこれらの点の中で、$y < f(x)$ となっているもの、つまり (図の黒い小円) の数 m を数える。この場合は領域

$$Q = \{x^2 + y^2 < 1, 0 \le x \le 1, 0 \le y \le 1\}$$

に含まれているものを数えればよいので $x^2 + y^2 < 1$ かどうかで判断すればよい。

デタラメに配置した点が均等に分布している場合、P, Q にそれぞれ含まれている点の数 n, m は P, Q の面積に比例する。P の面積は 1 であるので、Q の面積、つまり積分値は

$$\int_0^1 f(x)dx = Q \simeq \frac{m}{n}P = \frac{m}{n}$$

によって求められる。

この方法による積分を行う関数 `montecarlo(n)` をファイル 6.2 に示す。4 行目とその次の行で用いられている関数 `rand()` は、0 以上 1 未満の乱数を 1 つ発生させるものである。

```
def montecarlo(n)
  m=0
  for i in 1..n
    x=rand() # random number in [0,1)
    y=rand()
    if x*x + y*y < 1.0
      m = m + 1
    end
  end
  m*1.0/n
end
```

ファイル 6.2: montecarlo.rb

練習 6.3 (Monte Carlo 法の正確さ) Monte Carlo 法で高い精度の結果を得るためには、非常に多くの点を用いる必要がある。Q の面積は半径 1 の円の面積の 1/4 になることから montecarlo(n) の結果と解析的に求めた積分値を比較し、n を $10, 100, 1000, \ldots$ と増やした場合に何桁目まで一致するかを調べよ。(注意: 乱数を用いているので n の大きさに関らず偶然良い結果や悪い結果になることがあるので、同じ n の値で何回か実行してみよ。)

6.2　乱数と Monte Carlo 法

関数 montecarlo で使われていた rand() は**乱数**を発生させる関数であった。人間であればサイコロを振ることで乱数を作ることができるが、コンピュータは、基本的には与えられたデータを用いた計算しか行わないために、乱数を作ることは意外に難しい。

6.2.1　乱数とは何か

そこで、まず乱数の定義を与えた上で、コンピュータが作る乱数についての説明をしておこう。

まず定義であるが、ある 1 つの数だけを見てそれがデタラメな数かどうかを言うことはできない。乱数とは、数を何個も作ったときに、それらがデタラメになっているかどうかによって定義される。つまり数列、**乱数列**として定義される。より正確には次のように定義される。

次のような数列

$$\{x_1, x_2, \ldots, x_n, x_{n+1}, \ldots\}$$

の x_{n+1} の値が、$x_1, x_2, \ldots, x_n$ からは予測できず、x_i 全体の値はある確率分布に従うとき、この数列は**乱数列**であるという

練習 6.4 (乱数列となる条件) 次のような数列は乱数列と言えるかどうか考えよ。

a) ある地点の毎日の最高気温を日付順に並べたもの

b) 毎年の年末ジャンボ宝くじの当籤番号を年ごとに並べたもの (抽籤方法は数字が書かれた回転する円盤を矢で射て決めており、数字の範囲と当籤本数は毎年同じだとする)

c) 52 枚のカードを充分にシャッフルした後で一列に並べ、各カードの数字を取り出したもの

6.2.2　乱数と確率分布

代表的な確率分布の一つは**一様分布**で、この分布に従う乱数は**一様乱数**と呼ばれる。一様乱数は、出現確率が値によらず一定になっている。例えば、サイコロを振って出た数を並べたものは、前後の数と関係がないため乱数列になる。そして 1 から 6 までの数字が 1/6 の確率で出現するため一様乱数である。

次に放射性原子の集まりがあったときに、一定時間あたりに崩壊する原子の個数を数え、それを並べた数列を考えてみよう。1 つ 1 つの原子核が、他の原子と無関係に崩壊するのであれば、これは乱数列になる。しかし、その個数は平均的には一定の値になるが、ある個数になる確率は個数によって異なる。つまり、これは一様ではない乱数列になっている。

値の出現確率が正規分布に従うような乱数は**正規乱数**という。正規分布に従うとは、確率変数 x が ξ よりも小さい値をとる確率 $P(x)$ が以下の正規分布に従う場合である。

$$P(x < \xi) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\xi} \exp\left(-\frac{x^2}{2}\right) dx$$

自然現象における確率は上の放射性原子の例のように正規分布に従うものも多いため、自然現象を扱うシミュレーションでは、一様乱数と共に正規乱数をよく用いる。

6.2.3　疑似乱数列

さて、Monte Carlo 法のところで用いた関数 `rand()` は「乱数を発生させる」と説明されていたが、実際には何をどのようにして作っているのだろうか。

この関数は、乱数列**のように見える**数列を計算によって求めている。n 個目までの乱数列が作られているときに、$n+1$ 個目の数を求める計算は、直前の m 個の値 $x_n, x_{n+1}, \ldots, x_{n-m+1}$ を用いた、以下のような式で表わされる。

$$x_{n+1} = f(x_n, x_{n+1}, \ldots, x_{n-m+1})$$

つまりこの数列の値は、その値より前の値と無関係だとは言えない (それどころか前の値を使って作られている!) ので、本当の意味での乱数列とは言えず、**疑似乱数列**と呼ばれる。

また、それぞれの値は有限の桁数で表わされる数であるので、沢山乱数を作っていると、必ず同じ値の繰り返しに戻ってしまう。従って、疑似乱数列を作る場合は、同じ値の繰り返しに戻るまでの個数、つまり疑似乱数の**周期**ができるだけ大きくなるように設計することが望ましい。疑似乱数を作るアルゴリズムを設計する場合は、一様分布に関する適合度、統計的独立性などを様々な乱数検定法で検定し疑似乱数としての資格を与えている。

6.2.4　Monte Carlo 法

第 6.1.3 節では、乱数を使って積分を求める方法を Monte Carlo 法として紹介したが、Monte Carlo 法は乱数を用いた数学的問題の解法の総称であり、積分以外の問題にも用いられる。

Monte Carlo 法がよく用いられる問題には**決定論的問題**と**非決定論的問題**の 2 種類がある。

決定論的問題は、厳密な解が存在する問題であり、乱数によって近似的に求めるものであり、(多次元の) 積分が代表例である。

非決定論的問題は、自然科学や社会科学におけるシミュレーションのように、問題の中に確率的な変動を含むような問題を指す。このような問題の多くは実際に実験が困難であったり多大な費用がかかったりするので、乱数を用いて疑似的な実験を行うことで問題を「解く」わけである。このような問題の例としては、確率的に移動する粒子が時間経過とともにどこに移るかを調べるようなランダムウォークがある。

6.3　実数データ表現と誤差

数値計算では実数値を用いる場合が多い。コンピュータ上では、実数といっても有限の精度でしか表現できないため、数値計算の結果は真の値の近似値になってしまう。計算結果がどのくらい正しいかを判断するためには、数値がどのように表現されているかを知ることと、どのような場合に誤差が生じるかを知っておく必要がある。

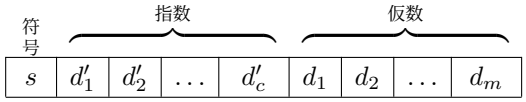
6.3.1　実数データ表現

コンピュータ上での数値データは 0 と 1 を並べた**ビット列**によって表わされる。数値計算で用いられる数値データは、桁数が非常に大きい数から 0 に近い小さな数までが同時に用いられる一方で、高速に計算することも求められているため、**浮動小数点表現**が用いられる。この浮動小数点表現は、数値を**符号部**、**指数部**、**仮数部**の 3 つに分けて表わす方法である。

浮動小数点表現にも色々な表現方法があるが、現在のコンピュータではIEEE 754 という規格が多く用いられている。

> 注意:　以下では、2 進数で表わされた数を使うことがあるので、そのような場合は区別のために数の後ろに $_{(2)}$ を付ける。10 進数であることを明記する場合は $_{(10)}$ を付ける。例えば $1101_{(2)}$ は 2 進数であり、10 進数で書くと $13_{(10)}$ という数を表わしている。また、$x_1x_2\ldots x_{n(2)}$ のように書いた場合は、上から i 桁目が x_i であるような n 桁の 2 進数を意味する。

IEEE 754 規格の中にもいくつかの表現方法があるのだが、その中でも現在よく用いられているものは、32 ビットのビット列によって表わす**単精度**表現と、64 ビットのビット列によって表わす**倍精度**表現である。これらは、32 または 64 ビットのビット列を



という 3 つの部分に分け、

$$(-1)^{s(2)}1.d_1d_2\ldots d_{m(2)} \times 2^{d'_1d'_2\ldots d'_{c(2)}-b}$$

という数を表わす方法である。

指数部のビット数 (c), 仮数部のビット数 (m) および (後述する) バイアス値 (b) は単精度表現か倍精度表現かによって異なり、以下のように決められている。

	仮数部 m	指数部 c	バイアス値 b
単精度	23	8	127
倍精度	52	11	1023

例えば次のような単精度 (32 ビット) の浮動小数点数があった場合、

0	1	0	0	1	1	0	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	1	1	1	1	0	0	0	0	1	0

符号 は 0 ビット目を $s = 0$ として (-1) の s 乗、つまり 0 ならば正、1 ならば負、

指数 は 1 から 8 ビット目を正の整数として、そこからバイアス値 b を引いた値、つまり $d'_1 \dots d'_{8(2)} - b = 10011011_{(2)} - b = 155 - 127_{(10)} = 28_{(10)}$

仮数 は 9 ビット目から 31 ビット目までを 2^{-1} から 2^{-23} に対応させ、さらに 1 を足した値、つまり $1.d_1 \dots d_{23(2)} = 1.00011101111001111000010_{(2)} = 2^0 + 2^{-4} + 2^{-5} + 2^{-6} + \cdots +_{(10)} = 1.116814_{(10)}$

であるので、

$$(-1)^0 \times 1.00011101111001111000010_{(2)} \times 2^{10011011_{(2)}-127}_{(10)} = 1.116814 \times 2^{28} = 2.997925 \times 10^8$$

という数を表わしている。

ただし、この表現だけでは 0 を表わすことができない。そこで、$d'_1 \dots d'_c$ と $d_1 \dots d_m$ が全て 0 の場合は 0 を表わす、などの特別な決まりも定められている。0 の他にも、無限大や計算結果が実数でなくなったことを表わす特別な値もある。

練習 6.5 (浮動小数点数の表現)　a) 次のビット列は IEEE 754 規格の単精度浮動小数点数である。どのような値を表わしているかを求めて 10 進数で書け。

$$\overset{s}{1}\overbrace{11111110}^{d'_1 \dots d'_8}\overbrace{11000000000000000000000}^{d_1 \dots d_{23}}$$

b) 次の数を IEEE 754 規格の単精度浮動小数点数で表わすときの $s, d'_1 \dots d'_8, d_1 \dots d_{23}$ をビット列で書け。

(a) 1.0

(b) $3145728_{(10)}$

(c) $5/65536_{(10)}$ ($\simeq 7.62939453125 \times 10^{-5}$)

練習 6.6 (浮動小数点の大小 *) 2 つの数 x, y が IEEE 754 規格によって 32 あるいは 64 ビットの浮動小数点数として表わされていたとする。このビット列を 32 あるいは 64 ビットの整数だと解釈すると、全く別の数を表わしていることになるが、x と y の間の大小関係だけは保たれる。このことを証明せよ。ただし、整数と解釈する場合は 2 の補数表現が用いられているものとする。

6.3.2　丸め誤差

浮動小数点数を用いた数値計算は、一定の桁数を使った近似値を使って計算をするため、正確な値からの誤差が生じる。多くの場合、この誤差は無視

できるほど小さいのだが、場合によっては計算の正確さを損なうほど大きくなる。そのため、どのような場合にどのような誤差が生じるかを知っておくことは数値計算によって意味のある結果を得るためにも重要である。

最初に紹介するのは**丸め誤差**というものである。これは 10 進数だと正確に表わせる数でも有限桁の 2 進数では近似値になってしまうことで生じる誤差である。

例えば有限桁の 10 進数で 1/3 を表わすことを考えてみよう。1/3 は $0.33333\cdots$ という無限に続く小数なので、小数点以下 5 桁目で切り捨てた近似値 0.33333 は正しい値 1/3 よりも $0.0000033333\cdots$ だけ小さい値になってしまう。

2 進数を使っている場合に気を付けなければいけないのは、10 進数では有限の桁数で正確に表わされているのに、2 進数で表わすと無限に続く小数になってしまうような数である。例えば 1/10 は 10 進数では 0.1 という有限の桁数で正確に表わせる。しかし 2 進数にすると、

$$
\begin{aligned}
0.1_{(10)} &= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \cdots_{(10)} && (6.6)\\
&= 0.0001100110011\cdots_{(2)} && (6.7)\\
&= 1.100110011\cdots_{(2)} \times 2^{-4} && (6.8)
\end{aligned}
$$

という循環小数になってしまう。

これを IEEE 754 規格の倍精度浮動小数点数として表わす場合、仮数部は (先頭の 1 を除いた)52 桁で表現するので、以下の下線部分が使われる。

$$1.\underline{1001100110011001100110011001100110011001100110011001}1001\cdots$$

さらに、切り捨てられる桁は 0 捨 1 入 (0 ならばそのまま、1 ならば切り上げ) がされるので、

$$1.\underline{1001100110011001100110011001100110011001100110011010}$$

という数に近似される。最後の 2 桁が切り上げによって変化していることに注意してほしい。そのため、この数は 0.1 よりもわずかに大きな値を表わしている。実際、Ruby でも以下の計算によって確認できる。

```
1 irb(main):003:0> 0.1 * 3 == 0.3
2 => false
```

$0.3_{(10)}$ は 2 進数で表わすと $1.001100110011\cdots \times 2^{-2}$ という循環小数である。どちらも循環小数であるのに値が一致しないことは、$0.1_{(10)} \times 3_{(10)}$ が 2 進数でどのように計算されるかを追うことで確認できる。この計算は以下のように進む。

$$
\begin{array}{rl}
1.10011001100\ldots110011010 & {}_{(2)} \times 2^{-4}_{(10)} \\
\times) \qquad\qquad 1.1 & {}_{(2)} \times 2^{1}_{(10)} \\
\hline
110011001100\ldots110011010 & {}_{(2)} \times 2^{-3}_{(10)} \\
+)\ 110011001100\ldots110011010 & {}_{(2)} \times 2^{-3}_{(10)} \\
\hline
10.011001100110\ldots011001110 & {}_{(2)} \times 2^{-3}_{(10)} \quad (*) \\
= 1.\underbrace{0011001100110\ldots0110100}_{52\text{ 桁}} \hookleftarrow & {}_{(2)} \times 2^{-2}_{(10)}
\end{array}
$$

(*) の行までの計算は正確に行われるが、この値は先頭の 1 を除いて 54 桁あるので、末尾の 2 桁が 0 捨 1 入される。結局、繰り上がりが起きて最後の行のような値になる。

一方、$0.3_{(10)}$ の 2 進数表現は $1.001100110011\cdots \times 2^{-2}$ だったので、52 桁の仮数部は ...0110011 で終わる。よって、$0.1_{(10)} \times 3_{(10)}$ の方が

$$0.\underbrace{00\ldots001}_{52\text{ 桁}}{}_{(2)} \times 2^{-2} = 2^{-54}$$

だけ大きくなってしまっている。

このことも、次のような計算式で実際に確かめられる。

```
1 irb(main):005:0> 0.1 * 3 - 0.3
2 => 5.55111512312578e-17
3 irb(main):006:0> 2 ** -54
4 => 5.55111512312578e-17
```

練習 6.7 (丸め誤差) irb で `0.1 * 5 - 0.5` を計算し、その結果を説明せよ。

6.3.3　浮動小数点数の精度

ここで実数データ表現が扱える数値の範囲を整理しておこう。浮動小数点数表現の場合、仮数部の桁数は有効数字の桁数に相当する。また、指数部の桁数は表わされる数の大きさを決定する。

IEEE 754 規格の単精度表現の場合は仮数部が 23 ビット、指数部が 8 ビットであった。仮数部の先頭には 1 が付くので合計 24 ビットの情報を表わせるが、これは $\log_{10}(2^{24}) \simeq 7.2$ と約 7 桁の精度しかないことを意味している。

また、仮数部の表わせる値の範囲は $1.00\ldots0_{(2)} = 1$ から $1.11\ldots1_{(2)} \simeq 2_{(10)}$ までである。

指数部は $2^{-126} \simeq 1.2 \times 10^{-38}$ から $2^{127} \simeq 1.7 \times 10^{38}$ までの範囲の値をとれる。(指数部のビットがすべて 0 の場合とすべて 1 の場合は、0 や無限大などの特別な値を表わすために用いられるので除外してある。)

結局、単精度表現で表わせる最小の数は約 1.2×10^{-38}, 最大の数は $2\times1.7\times10^{38} \simeq 3.4 \times 10^{38}$ となる。

練習 6.8 (倍精度表現の精度) IEEE 754 規格の単精度表現の有効桁数と表現できる値の範囲は下のようにまとめられる。これにならって倍精度の場合の有効数と表現できる値の範囲を求めよ。

	単精度	倍精度
仮数部のビット数	23	52
10 進有効桁数 (約)	$\log_{10}(2^{23+1}) \simeq 7$	
指数部のビット数	8	11
最小の指数	$2^{-126} \simeq 1.2 \times 10^{-38}$	
最大の指数	$2^{127} \simeq 1.7 \times 10^{38}$	
最大値	$2 \times 1.7 \times 10^{38} \simeq 3.4 \times 10^{38}$	

6.3.4　桁落ち誤差

第 6.1 節で見た数値積分の方法は、微小区間の面積を合計して全体の面積を求めるというものであった。このときの区間の幅を短くしてゆけば、それだけ正確な値が求められるはずである。

このように微小な値を扱う場合は違った種類の誤差に注意しなければいけない。

例として $f(x) = \log(x)$ の微分を数値計算によって求める問題で説明しよう。$f(x)$ の導関数は $f'(x) = \lim_{h \to 0} \dfrac{f(x+h) - f(x)}{h}$ である。極限を実際に求めることはできないので、数値計算では適当に小さな h を使った近似値 $\dfrac{f(x+h) - f(x)}{h}$ を使う。理論的には h を小さくすれば小さくするほど正確な値になるはずであるが、実際にそうなるか確かめてみよう。

次に示すのは h を $1,\ 0.1,\ 0.01,\ 0.001,\ \ldots,\ (0.1)^{\texttt{h\_digits}}$ と 1 桁ずつ小さくしてゆきながら $\dfrac{f(x+h) - f(x)}{h}$ を求め、解析的に求めた $f'(x) = \dfrac{1}{x}$ との差を求めてゆく関数 `cancellation(x,h_digits)` の定義である。

h に対する誤差の変化を見るため、この関数では (先頭を 0 番目として) i 番目が $h = (0.1)^i$ のときの誤差であるような配列を作って答えている。

関数 `cancellation` を使って $x = 2$, $h = 1.0$ から 10^{-15} までについて誤差を求めた結果は次のようになる。

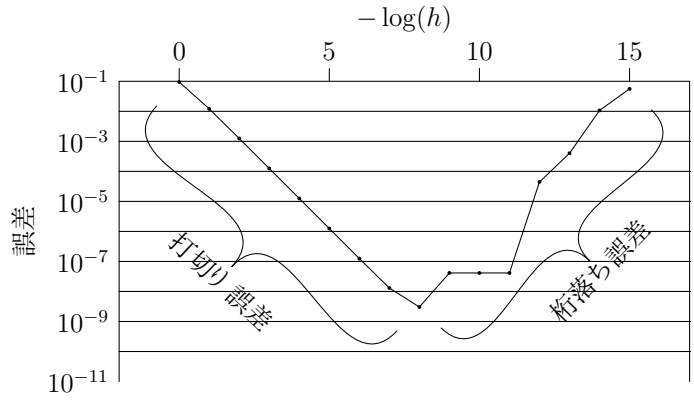
```
irb(main):004:0> cancellations(2.0, 15)
=> [0.0945348918918355, 0.0120983583056795,
    0.0012458488961028, 0.000124958348945658,
    1.24995825353524e-05, 1.24998572570423e-06,
    1.24941223755837e-07, 1.30307428736209e-08,
    3.03873576301683e-09, 4.13701852775006e-08,
    4.13701852775006e-08, 4.13701851664783e-08,
    4.44502911701727e-05, 0.00039963891867989,
    0.0107025913275716, 0.0559107901499378]
```

```
include(Math)
load("./abs.rb")

def f(x)
  log(x)
end

def cancellations(x,h_digits)
  errors=Array.new(h_digits+1)
  for i in 0..h_digits
    h = 0.1**i
    df=(f(x+h)-f(x))/h
    errors[i] = abs(df-1.0/x)
  end
  errors
end
```

ファイル 6.3: cancellation.rb (練習 3.1 で定義した abs を使用している)

最初は予想通り h を小さくするにつれて誤差が小さくなっている。この部分は粗い近似によって引き起こされる**打ち切り誤差**(後述) である。

しかし誤差は $h = 10^{-8}$ のときに約 3.0×10^{-9} となったのが最小で、その後は増加に転じている。横軸に h の桁数、縦軸に誤差の対数をとったグラフにすると下のようになる。



つまり h を小さくすればするほど、逆に誤差が大きくなっている。この後半の誤差は**桁落ち誤差**と呼ばれるものである。桁落ち誤差は、非常に近い 2 つの値の差を求めた場合に有効桁数が減ることで引き起こされる誤差を指す。

この例で言えば、h を小さくするにつれて $f(x+h)$ と $f(x)$ の上位桁はよく多く一致する。従って、その差の有効桁数はそれだけ減るため、誤差が生じている。

6.3.5　情報落ち誤差

数値積分では、微小領域を台形や 2 次関数で近似した面積を求め、それを足し合わせることで全体の面積を求めていた。微小領域の幅を小さくすればそれだけ近似が精確になるので、求められる解もより精確になるはずであるが、果たして本当だろうか。

そこで $1/n$ を n 回加えるという関数を定義し、n によってその答がどう変わるかを調べてみよう。これはつまり $f(x)=1$ という関数を 0 から 1 までの範囲を n 個に分けて数値積分することに相当する。ここでは $n=10^{\texttt{digits}}$ の場合に限って sum(digits) という関数を定義する。

```
1  # simulate 32 bit floating point representaiton
2  def coerce32(f)
3    [f].pack("f").unpack("f")[0]
4  end
5
6  def sum(digits)
7    n=10**digits
8    sum = 0.0
9    d = coerce32(1.0 / n)
10   for i in 1..n
11     sum = coerce32(sum + d)
12   end
13   sum
14 end
```

ファイル 6.4: lossofinformation.rb

この関数では、$1/n$ を計算するときと、合計する変数 sum に $1/n$ を加える際に coerce32 という関数を使っている。この関数は値を 32 ビット浮動小数点数に変換して、再び 64 ビット浮動小数点数に戻している。Ruby は (使用している環境にもよるが) 浮動小数点数には 64 ビット表現を用いているため、そのままだと情報落ち誤差の影響を観測することが難しいため、あえてビット数の少ない表現に変換している。

関数 sum(digits) を使って、$n=10^6, 10^7, 10^8$ についての値を求めた画面は以下のようになる。

注意:　$n = 10^8$ のときは 1 億回の繰り返しになるので、以下の計算には非常に時間がかかる。

```
irb(main):004:0> sum(6)
=> 1.0090389251709
irb(main):005:0> sum(7)
=> 1.06476747989655
irb(main):006:0> sum(8)
=> 0.25
```

$n = 10^6, 10^7$ のときの誤差は、丸め誤差が蓄積したものである。$n = 10^8$ のときに突然 0.25 という異常な値になっているが、これが情報落ち誤差である。

情報落ち誤差とは、大きな数に小さな数を足した場合に、小さな数が大きな数の有効桁の範囲外になることで無視されてしまうために起きる。この例では、$n = 10^8$ のときの d は 10^{-8} であり、sum が 1 に近い値の場合には 8 桁以上の有効桁がなければ d の値を足した分は無視されてしまう。32 ビット浮動小数点数の仮数部の有効桁数は約 7 桁なので途中から何回 d を加えても値が増えなくなってしまい、0.25 という結果になってしまっている。

6.3.6　打切り誤差

Taylor 展開のように、ある値が無限級数によって表わされているとき、その値を近似的に計算する場合には適当な数の項までを計算して以降の計算を打ち切ることになる。そのような場合、仮に計算を無限に高い精度で行っていたとしても打ち切られた以降の項が誤差として残ることになる。このような誤差を**打ち切り誤差**という。

例えば指数関数の原点の周りの Taylor 展開は、

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \cdots$$

という式であるが、この計算を無限回の計算を行うことはできないので、有限回 (n 回) で打切った近似値

$$e^x \simeq \sum_{i=0}^{n} \frac{x^i}{i!}$$

を計算することになる。このとき、打ち切り誤差の主要成分は

$$\frac{x^{n+1}}{(n+1)!}$$

となる。

6.4　連立 1 次方程式

$2x - y = 2$ と $-x + 2y = 5$ という方程式から x, y の値を求めるような連立 1 次方程式を n 個の変数に一般化した問題は、CT スキャナや検索エンジンにおける web ページの順位付けなどのように色々な場面で用いられる。

連立 1 次方程式の解法は、コンピュータの発明以前から数多くのものが考えられてきた。特に小規模な (つまり変数の数が少ない) 連立 1 次方程式は **Gauss 消去法**や **Gauss-Jordan 法**などの**消去法**によって解くことができる。これは人間が筆算で解く方法に相当する。

規模の大きな連立 1 次方程式は、**Jacobi 法**などの**反復法**という解法が用いられることが多いが、本書では扱わない。

6.4.1　Gauss 消去法

Gauss 消去法は、連立 1 次方程式の変数 1 つずつに注目し、その変数の係数を 0 にする (つまりその変数を消去する) ような変形を繰り返してゆく方法である。x, y, z からなる連立 1 次方程式であれば、1 つ目の方程式を使って他の 2 つの方程式から x を消去し、2 つの目の方程式を使って 3 つの目の方程式から y を消去して、z の値を求めるという手順になる。その後は、求めた z の値と 2 つの方程式から y の値を、そして y, z の値と 1 つ目の方程式から x の値を求めることができる。

実際の連立 1 次方程式でこの手順を説明しよう。連立 1 次方程式として次のようなものが与えられたとする。

$$\begin{array}{rcrcrcl} x & + & y & - & z & = & 2 \\ 3x & + & 5y & - & 7z & = & 0 \\ 2x & - & 3y & + & z & = & 5 \end{array}$$

行列演算を使うと、この方程式は次のような係数行列と変数ベクトル、定数ベクトルの間の方程式として書ける。

$$\underbrace{\begin{pmatrix} 1 & 1 & -1 \\ 3 & 5 & -7 \\ 2 & -3 & 1 \end{pmatrix}}_{\text{係数行列}} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \underbrace{\begin{pmatrix} 2 \\ 0 \\ 5 \end{pmatrix}}_{\text{定数ベクトル}}$$

消去法では、1 つの方程式を定数倍したり、ある方程式に別の方程式 (の定数倍したものを) 加えたりする操作を行う。これは行列と定数ベクトルの特定の行の定数倍や、行の間の加減算に相当するので、以降では係数行列と定数ベクトルをつなげた行列を使って考えることにする。変数ベクトルは変化し

ないので、説明のために下のような書き方を使う:

1	1	−1	2	r_1
3	5	−7	0	r_2
2	−3	1	5	r_3

これは左から係数行列、定数ベクトル、方程式の名前や操作を表わす。

さて、最初の行列・ベクトルが下のような状態だったとする。

1	1	−1	2	r_1
③	5	−7	0	r_2
2	−3	1	5	r_3

まずは r_1 を使って r_2, r_3 の (先頭を 1 として)1 列目の係数を 0 にする。③で示された係数を 0 にするためには r_2 から r_1 の 3 倍を引けばよい。この操作を $r_2 - 3r_1$ と表わすことにして、2 行目の右端に書くことにする。

同様に 2 で示された係数を消去するために $r_3 - 2r_1$ を行う。変形後の係数・定数は次のようになる。右端の列には、それぞれの行にどのような操作を行ったかが書かれている。

1	1	−1	2	r_1
0	2	−4	−6	$r_2 - 3r_1$
0	−5	3	1	$r_3 - 2r_1$

各方程式の名前は再び r_1, r_2, r_3 だとして、今度は〇で示された係数を調整する。つまり変形前が

1	1	−1	2	r_1
0	②	−4	−6	r_2
0	㊀5	3	1	r_3

であり、これの r_2 の 1 列目を 1 に、r_3 の 1 列目を 0 にするような変形を行う操作と結果は次のようになる。

1	1	−1	2	r_1
0	1	−2	−3	$r_2/2$
0	0	−7	−14	$r_3 + (5/2)r_2$

さらに 3 行 3 列を 1 にするような変形を行うと、次のようになる。

1	1	−1	2	r_1
0	1	−2	−3	r_2
0	0	1	2	$-r_3/7$

これで行列の対角成分が全て 1 であるような方程式が得られた。3 行目から逆順に z, y, x を求める**後退置換**を行うと、

$$\begin{aligned} z &= 2 \\ y &= 2z - 3 = 1 \\ x &= -y + z + 2 = 3 \end{aligned}$$

が最終的な解として求まる。

6.4.2　Gauss-Jordan 法

Gauss 消去法の最後は、消去によって得られた上三角行列に、後退置換を行っている。消去の手順を少し変更すると、係数行列を単位行列に変形することができ、後退置換を行う必要がなくなる。これが **Gauss-Jordan 法**である。

前出の連立 1 次方程式を用いて Gauss-Jordan 法の手順を見てゆこう。

1. まずは Gauss 消去法と同様に係数行列と定数ベクトルを並べたものから始める。

1	1	−1	2	r_1
3	5	−7	0	r_2
2	−3	1	5	r_3

2. r_1 を使って r_2, r_3 の第 1 列目の係数を消去する。これは Gauss 消去法と同一である。

1	1	−1	2	r_1
0	2	−4	−6	$r_2 - 3r_1$
0	−5	3	1	$r_3 - 2r_1$

3. 次に r_2 を使って他の行の係数を消去する。ここで Gauss-Jordan 法では r_1 の係数も消去する。

1	0	1	5	$r_1 - r_2/2$
0	1	−2	−3	$r_2/2$
0	0	−7	−14	$r_3 + (5/2)r_2$

4. さらに r_3 を使って r_1, r_2 の係数を消去する。

1	0	0	3	$r_1 + r_3/7$
0	1	0	1	$r_2 - 2r_3/7$
0	0	1	2	$-r_3/7$

このとき係数行列は単位行列になっているので、定数ベクトルの部分が与えられた連立 1 次方程式の解になっている。

Gauss-Jordan 法のプログラム

```
1  def gj(a) # Gauss-Jordan method without pivoting
2    row = a.length()
3    col = a[0].length()
4    for k in 0..(col-2)
5      akk = a[k][k]
6      for i in 0..(col-1)        # normalize row k
7        a[k][i]=a[k][i]*1.0/akk
8      end
9      for i in 0..(row-1)        # eliminate column k
10       if i != k                # of all rows but k
11         aik = a[i][k]
12         for j in k..(col-1)
13           a[i][j] = a[i][j] - aik * a[k][j]
14         end
15       end
16     end
17   end
18   a
19 end
```

ファイル 6.5: gj.rb

ファイル 6.5 に Gauss-Jordan 法によって n 変数の連立 1 次方程式を解く関数 gj(a) を示す。この関数は、係数行列と定数ベクトルを並べた n 行 $n+1$ 列の配列 a を引数として受け取り、行数と列数を変数 row, col に覚えておく。

関数全体は k 行目を使った消去を、0 行目から順に行ってゆく (4 行目からの繰り返し)。なお、Ruby では配列の番号は 0 から始まるので、以降では行や列は 0 行目から始まるとして説明する。

消去はまず、$\mathtt{a}_{kk}$ が 1 になるように k 行の係数・定数を正規化する。具体的には 6 行目からの繰り返しですべての列 i について $\mathtt{a}_{ki}$ を $1/\mathtt{a}_{kk}$ 倍している。このとき整数どうしの割り算によって切り捨てが起きること防ぐため、割り算に先立って 1.0 を先に掛けている。

次に 9 行目からの繰り返しで、k 以外のすべての行 i について係数 a_{ik} を消去する。そのためには、k 行を a_{ik} 倍した行ベクトルを i 行から引けばよいので、すべての列 j について a_{ij} から a_{kj} の a_{ik} 倍を引いている (13 行目)。

この繰り返しが終わると各行の最後の要素が、それぞれの変数の解になっている。関数 gj は、(単位行列へと変形された) 係数行列と定数ベクトルを

あわせた配列のまま答を返している。

```
irb(main):005:0> a=[[1, 1,-1,2],
irb(main):006:1*     [3, 5,-7,0],
irb(main):007:1*     [2,-3, 1,5]]
=> [[1, 1, -1, 2], [3, 5, -7, 0], [2, -3, 1, 5]]
irb(main):008:0> gj(a)
=> [[1.0, 0.0, 0.0, 3.0], [0.0, 1.0, 0.0, 1.0],
    [-0.0, -0.0, 1.0, 2.0]]
```

6.4.3　Pivoting 付き Gauss-Jordan 法

消去法では係数行列の要素で割り算を行う場所がある。関数 gj(ファイル 6.5) の 7 行目がそれである。このとき、

- a_{kk} が 0 の場合は、0 による割り算が行われてしまう、また、
- a_{kk} が 0 に近い値の場合は、その結果が非常に大きな値になり、その後で k 行目を a_{ik} 倍して i 行目から引く際に、大きな数どうしの引き算によって**情報落ち誤差**が生じてしまう

という問題がある。例えば

$$\begin{array}{rrrcr} x & -50y & -3z & = & -90 \\ -85x & +2y & -25z & = & -6 \\ 79x & +5y & +30z & = & -1 \end{array}$$

を解析的に解くと

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ -3 \end{pmatrix}$$

となるはずであるが、前出の gj(a) を使って解くと次のように誤差を含んだ結果になってしまう。

```
irb(main):010:0> b=[[  1,-50, -3,-90],
irb(main):011:1*     [-85,  2,-25, -6],
irb(main):012:1*     [ 79,  5, 30, -1]]
=> [[1, -50, -3, -90], [-85, 2, -25, -6], [79, 5,
    30, -1]]
irb(main):013:0> gj(b)
=> [[1.0, 0.0, 0.0, 0.999999999999977], [-0.0, 1.0,
     0.0, 2.0], [0.0, 0.0, 1.0, -2.99999999999996]]
```

このようなことを防ぐ方法として **pivoting** と呼ばれる行を入れ替えるやり方がある。具体的には、k 行目を使って消去する前に、$|a_{ik}|$ が最も大きい行 $i = p$ を選び、k 行と p 行を入れ替える。これは連立方程式の式の順序を入れ替えることに相当するため、自由に行うことができる。

$$
\begin{array}{rccccc}
 & \vdots & \vdots & \vdots & \\
\cdots & 1 & a_{k-1,k} & a_{k-1,k+1} & \cdots \\
k\text{ 行} \quad \cdots & 0 & \textcircled{$a_{k,k}$} & a_{k,k+1} & \cdots \\
\cdots & 0 & a_{k+1,k} & a_{k+1,k+1} & \cdots \\
 & \vdots & \vdots & \vdots & \\
i\text{ 行} \quad \cdots & 0 & a_{i,k} & a_{i,k+1} & \cdots \\
 & \vdots & \vdots & \vdots &
\end{array}
$$

Pivoting を行いながら Gauss-Jordan 法によって連立一次方程式を解く関数 `gjp(a)` をファイル 6.6 に示す。この関数は、ファイル 6.5 の `gj(a)` に 5 行目と 6 行目を追加しただけで、他はまったく同じものである。(ただし、そこで使われている `maxrow` の定義は省略してある。)

Pivoting は、k 行を使って消去を行う前に、`maxrow(a,k)` という関数を使って k 列目の係数の絶対値が最大となる行を k 行目以降から探し、その行番号を `max` にしまう (5 行目)。そして、`swap(a,k,max)` に配列 `a` の `k` 番目と `max` 番目を入れ替える、つまり行列の k 行を `maxrow` で見つけた行と交換する。

練習 6.9 (Pivoting の完成) 関数 `maxrow(a,k)` の定義を追加し、pivoting 付き Gauss-Jordan 法を完成させよ。さらに前出の連立 1 次方程式が、より小さな誤差で解けることを確認せよ。

```
irb(main):015:0> b=[[  1,-50, -3,-90],
irb(main):016:1*     [-85,  2,-25, -6],
irb(main):017:1*     [ 79,  5, 30, -1]]
=> [[1, -50, -3, -90], [-85, 2, -25, -6], [79, 5,
   30, -1]]
irb(main):018:0> maxrow(b,0)
=> 1
irb(main):019:0> maxrow(b,1)
=> 2
irb(main):020:0> gjp(b)
=> [[1.0, 0.0, 0.0, 1.0], [-0.0, 1.0, 0.0, 2.0],
   [0.0, 0.0, 1.0, -3.0]]
```

```
def gjp(a) # Gauss-Jordan method WITH pivoting
  row = a.length()
  col = a[0].length()
  for k in 0..(col-2)
    max=maxrow(a,k) # find absolute-maximal coeff.
    swap(a,k,max)   # swap rows
    akk = a[k][k]
    for i in 0..(col-1)       # normalize row k
      a[k][i]=a[k][i]*1.0/akk
    end
    for i in 0..(row-1)       # eliminate column k
      if i != k       # of all rows but k
        aik = a[i][k]
        for j in k..(col-1)
          a[i][j] = a[i][j] - aik * a[k][j]
        end
      end
    end
  end
  a
end
```

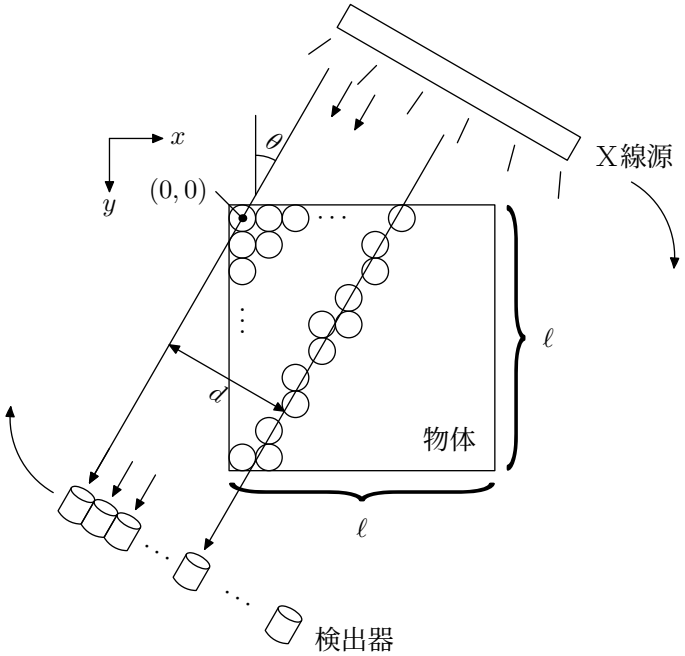
ファイル 6.6: gjp.rb (関数 gjp の定義のみ)

6.5　応用問題: CT スキャン*

6.5.1　原理

コンピュータ断層撮影 (computed tomography) とは、X 線装置を用いて人体の各方向から透過線量を測り、そのデータをもとに人体の横断面画像を逆算する技術である。

実際の CT スキャンでは X 線の拡散や計測誤差などの問題を扱わなければいけないが、ここではより簡単化した問題を考えることにしよう。ここでの CT スキャナは、次の断面図のように X 線源と検出器を並べたものが物体をはさんで回転しながら計測をする機器である。



物体は直径 1 の小さな円が $\ell \times \ell$ 個並べられて構成されているものとする。線源から照射された X 線は、物体の内部を通過して検出器に到達する。このとき X 線は物体内部の物質に吸収されるので、検出器は通過した線上の物質の密度の合計値を測ることになる。一列に並んだ検出器によって、異なる線上の物質についての情報が分かることになる。

これだけでは内部の構造は分からないので、CT スキャナ全体の角度 θ を何回か変えて同様の計測をする。すると各小円の密度を未知変数とした連立 1 次方程式を作ることができる。角度 θ のときに 1 つの検出器が得た値は、その検出器に到達する線上の小円の密度に関する 1 次方程式を 1 つ作るので、ℓ 個の検出器を使えば ℓ 個の連立 1 次方程式になる。角度を変えた測定を何回か行えば、解が存在するのに充分な数の連立 1 次方程式が得られる。

6.5.2　計算式

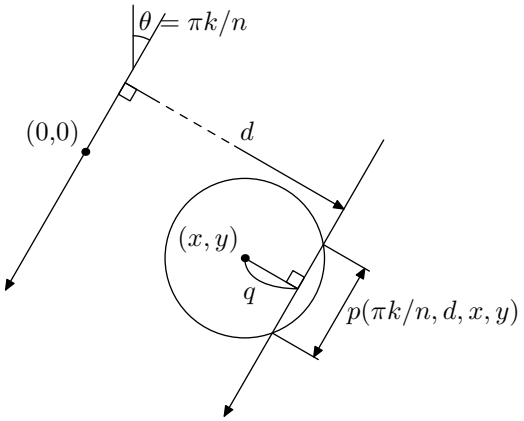
それでは計測値からどのような連立 1 次方程式を作ればよいのかを考えてゆこう。いま、検出器は間隔 1 で並び ℓ 個あり、端から $0, 1, 2, \ldots, \ell - 1$ という番号が付いているものとする。測定した n 回行い、そのときの角度 θ は $0, \pi/n, 2\pi/n, \ldots, \pi(n-1)/n$ だったとする。角度 $\pi k/n$ のときに d 番の検出器が測定した値を s_{kd} と書くことにする。また、左端の検出器に届く X 線は、常に左上の座標 (0,0) にある小円上を通過する位置にあるものとする。

座標 (x, y) にある小円の密度を変数 w_{xy} で表わすことにする。以下は、w_{xy} を未知数とした連立 1 次方程式を作ることになる。ここで s_{kd} として測定さ

れた X 線量は、その検出器に至る直線と、線上にあった各小円との交わりの長さと密度の積の和だとする。つまり、角度 $\pi k/n$ で原点との距離が d である直線と、中心が (x, y) にある小円との交わりの長さを $p(\pi k/n, d, x, y)$ で表わすことにすると、

$$s_{kd} = \sum_{\substack{x=0\ldots\ell-1 \\ y=0\ldots\ell-1}} p(\pi k/n, d, x, y) w_{xy} \tag{6.9}$$

ということである。ここで $p(\theta, d, u, v)$(ただし $\theta = \pi k/n$) は、下の図のような関係になっており、



$$\begin{aligned} p(\theta, d, u, v) &= \begin{cases} \sqrt{1-4q^2} & (q < \frac{1}{2}) \\ 0 & (q \geq \frac{1}{2}) \end{cases} \\ \text{ただし } q &= |x\cos\theta + y\sin\theta - d| \end{aligned} \tag{6.10}$$

となっている。

6.5.3　関数の定義

配布プログラム `scans.rb` には、関数 `scan1()`, `scan2()`, `scan3()`, `scan4()` が定義されている。これらの関数は、物体を CT スキャナによって測定された X 線量を n 行 ℓ 列の 2 次元配列 `s` を返す。行数 n(= `s.length()`) はスキャンの回数で、列数 ℓ(= `s[0].length()`) は画像の 1 辺の長さである。この配列の値 `s[`k`][`d`]` は、角度 $\pi k/n$ でスキャンしたときの d 番目の検出器の測定値である。

練習 6.10 (部分の通過距離) 式 6.10 に従って p(`theta,d,x,y`) を求める関数 `penetration(theta,d,x,y)` を定義せよ。

```
1 irb(main):004:0> penetration(0,0,0,0)
2 => 1.0
```

```
irb(main):005:0> penetration(3.141592/4,1,1,1)
=> 0.560096865715943
irb(main):006:0> penetration(3.141592/4,1,2,1)
=> 0.0
```

式 6.9 を $k = 0, \ldots, n-1,\ d = 0, \ldots, \ell - 1$ について連立させた 1 次方程式を係数行列 M、変数ベクトル $\mathbf{w}$、定数ベクトル $\mathbf{s}$ を用いて

$$M\mathbf{w} = \mathbf{s}$$

と書くことにする。ただし、$\mathbf{w}_{\ell y+x} = w_{xy}, \mathbf{s}_{\ell k+d} = s_{kd}$ である。このとき行列 M は

$$M_{\ell k+d,\ \ell y+x} = p(\pi k/n, d, x, y)$$

のように定められる。

練習 6.11 (係数行列の作成) 画像の 1 辺の長さが l, スキャンの回数が n のときの係数行列 M を作成する関数 coefficients(l,n) を定義せよ。

```
irb(main):008:0> coefficients(2,3)
=> [[1.0, 0.0, 1.0, 0.0], [0.0, 1.0, 0.0, 1.0],
    [1.0, 0.0, 0.0, 0.0], [0.0, 2.1073424255447e-08,
     0.963433044002285, 0.681250038633213], [1.0,
    2.98023223876953e-08, 0.0, 0.681250038633213],
    [0.0, 0.0, 0.963433044002285, 0.0]]
```

練習 6.12 (係数行列と定数ベクトルの合成) 係数行列 M と、測定値ベクトルを表わす配列 s から連立 1 次方程式を表わす配列を作る make_equations(l,n,m,s) を定義せよ。ただし l は 1 辺の長さ、n はスキャンの回数、m は $\mathtt{l} \times \mathtt{n}$ 行 $\mathtt{l}^2$ 列の配列、s は n 行 l 列の配列である。作られる配列は $\mathtt{l} \times \mathtt{n}$ 行 $\mathtt{l}^2 + 1$ 列の配列で、右端の列以外は m と同じ内容で、右端の列に s の内容を縦に並べた値が入る。

```
irb(main):010:0> make_equations(2,3,coefficients
    (2,3),[[1,1],[1.4,1.9],[1.4,1]])
=> [[1.0, 0.0, 1.0, 0.0, 1], [0.0, 1.0, 0.0, 1.0,
    1], [1.0, 0.0, 0.0, 0.0, 1.4], [0.0,
    2.1073424255447e-08, 0.963433044002285,
    0.681250038633213, 1.9], [1.0, 2.98023223876953e
    -08, 0.0, 0.681250038633213, 1.4], [0.0, 0.0,
    0.963433044002285, 0.0, 1]]
```

練習 6.13 (画像の復元) 測定値を表わす配列 s から 2 次元の画像を作成する関数 reconstruct(s) は次のように定義できる。この中で使われている equations_to_image(l,solved) は、配列 solved の (左端を 0 列目として)l^2 列目の上から l^2 個を $l \times l$ に並べ直した 2 次元配列を作るような関数である。この関数を定義して、reconstruct を完成させよ。

配布プログラム scans.rb に用意された関数 scan1(), scan2(), scan3(), scan4() が与える測定値から画像を作成せよ。(show(reconstruct(scan1())) のように実行すればよい。)

```
1 def reconstruct(s)
2   n=s.length()
3   l=s[0].length()
4   m=coefficients(l,n)
5   equations=make_equations(l,n,m,s)
6   solved=gjp(equations)
7   equations_to_image(l,solved)
8 end
```

ファイル 6.7: ctscan.rb 一部

6.6　章末問題

練習 6.14 (解析的な積分が難しい関数の数値積分) 関数 $g(x) = \dfrac{\sin x}{\log x}$ についての積分 $\displaystyle\int_{x_s}^{x_e} g(x)dx$ (ただし $x_e < 1$) について

a) 台形公式によって数値積分を求める関数 trapezoid_sinlog(xs,xe,n) を定義せよ。ただし n は分割の数とする。

b) Simpson 公式によって数値積分を求める関数 simpson_sinlog(xs,xe,n) を定義せよ。ただし n は分割の数とする。

```
1 irb(main):027:0> trapezoid_sinlog(0.1,0.9,100)
2 => -1.07129320182014
3 irb(main):028:0> simpson_sinlog(0.1,0.9,100)
4 => -1.07050038503067
```

練習 6.15 (Monte Carlo 法による 3 次元の積分) 半径 1 の球の体積を Monte Carlo 法によって求める関数 montecarlo3d(n) を定義せよ。ただし n は点

の個数だとする。半径 r の球の体積は $\frac{4}{3}\pi r^3$ であることが知られているので、これとの誤差を調べよ。

練習 6.16 (Monte Carlo 法による積分) 関数 $f(x) = \dfrac{x}{(x+1)(x+2)}$ を Monte Carlo 法によって数値積分する関数 `montecarlo_f(xs,xe,n)` を定義せよ。ただし、(`xs`, `xe`) は 0 < `xs` であるような積分区間、`n` は点の個数だとする。

ヒント: 区間 (`xs`, `xe`) において $f(x)$ が内側に収まるような長方形の領域を求め、その内側に点を打てばよい。$x > 0$ の範囲で $0 < f(x) < 3 - 2\sqrt{2}$ である。