情報科学 2007　久野クラス　＃7

久野　靖*

2007.11.30

はじめに

さて皆様、駒場祭はいかがでしたか。もう年末が近いので、年内に頑張って来年になってからは楽をするということで、あとちょっと頑張ってください。今回は数値解析で最後に残った話題である「常微分方程式の数値解法」を扱い、その後ようやく、Ruby の特徴である「オブジェクト指向」の話題に入ります。

1　演習問題解説

1.1　演習 1a: Gauss-Jordan の消去法

演習 1a は、Gauss-Jordan の消去法のプログラムを作ることだった。これは Gauss 法を削ってちょっと直せばできる。

```
def gaussjordan(a)
  n = a.length
  n.times do |i|
    n.times do |j|
      if i != j then
        r =  a[j][i] / a[i][i].to_f
        i.step(n) do |k| a[j][k] = a[j][k] - a[i][k]*r end
      end
    end
    # p(a)
  end
  n.times do |i| a[i][n] = a[i][n] / a[i][i] end
  return a
end
```

要するに、内側ループを i 以降ではなく 0 番から全部やり、ただし i 番は飛ばすために if で囲んだ。あと、消去しただけだと係数が 1 になっていないので、最後に係数で割る処理を 1 行入れてある。

1.2　演習 1b: ピボット選択

ピボット選択とは何だったかというと、x_i を消去しようとしたところで i 列の x_i の係数が 0(かそれに近い) だとうまく行かないので別の列と入れ替えて進めるという話だった。で、いちいち判断するのでなく、x_i を消去するときに常に係数の絶対値が最大の列を持って来てそれを使うようにした。ところで、係数の絶対値が最大の列を持って来たのに、その係数が 0(かそれに近い) だったら? それは方程式が不定/ 不能だということで、ついでに演習 1c も組み込んである。

*筑波大学大学院経営システム科学専攻

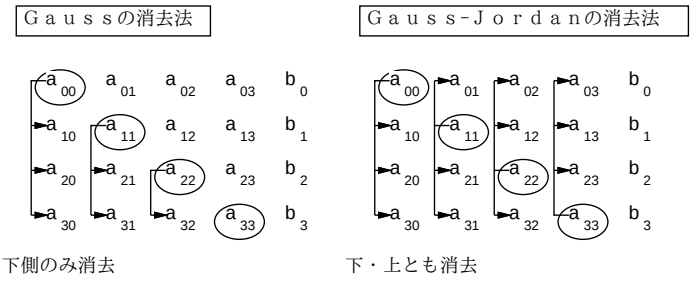

図 1: Gauss-Jordan の消去法

```
def gaussjordanpivot(a)
  n = a.length; idx = Array.new(n) do |i| i end
  n.times do |i|
    k = selectpivot(a, i, n)
    if a[k][i].abs < 0.00000001 then return nil end
    a[i],a[k] = a[k],a[i]; idx[i],idx[k] = idx[k],idx[i]
    n.times do |j|
      if i != j then
        r =  a[j][i] / a[i][i].to_f
        i.step(n) do |k| a[j][k] = a[j][k] - a[i][k]*r end
      end
    end
    p(a, idx)
  end
  b = Array.new(n)
  n.times do |i| k = idx[i]; b[k] = a[k][n] / a[k][k] end
  return b
end
```

つまり、下請け関数 `selectpivot` で i 列目以降 (この手前はもう消去に使ってしまったので除外) で x_i 項の係数が最大の列番号 k を得て、もしその係数が 0 なら不定/不能の印として `nil` を返す。そうでない場合は i 列と k 列を交換してこれまで通り進める。

ただしこれをやってしまうと、列がバラバラに入れ替わってしまうので、最後に解を出力するときどの変数が何番目に入っているか分からなくなって困る。このため、最初に配列 `idx` を用意してここに 0〜$n-1$ の番号を順に入れておき (このヘンな `Array.new` は初期化時に添字 i を渡されて任意の式により初期値を計算させてくれるというもの)、ここでは要するに `[0,1,2,...,n-1]` で初期化)、ピボット選択の交換時にこの配列も同じに交換することで、元の番号が分かるようにし、最後に結果用の配列 `b` を用意してここに元の番号で書き込むようにしている。

1.3 演習 2: 反復法

2a と 2b については色々入れて観察するだけですが、要は対角成分が大きいと収束しそうでないと収束しないとか、正しい解の回りを振動しつつ収束したり発散したりするとか、分かればいいのではないでしょうか。いちおう、Jacobi 法のコードを再掲:

```
def jacobi(a)
  n = a.length; x = Array.new(n, 0); count = 0
  while true do
    x1 = Array.new(n, 0); d = 0.0
```

```
    n.times do |i|
      v = a[i][n].to_f
      n.times do |j| if j != i then v = v - a[i][j]*x[j] end end
      x1[i] = v / a[i][i]; d = d + (x1[i]-x[i]).abs
    end
    x = x1; count = count + 1
    # p(x)
    if d < 0.000000001 then return x end
    if count > 100 then return nil end
  end
  return x
end
```

これを Gauss-Seidel 法にするには、要は配列 x1 を使わずに x だけで済ませるように直せばよい。そのとき、収束判定のための変数 d に x_i の変化の絶対値を累計するので、作業変数を 1 つ用意してここに計算し、差の絶対値を累計してから x に入れる程度。

```
def gaussseidel(a)
  n = a.length; x = Array.new(n, 0); count = 0
  while true do
    d = 0.0
    n.times do |i|
      v = a[i][n].to_f
      n.times do |j| if j != i then v = v - a[i][j]*x[j] end end
      z = v / a[i][i]; d = d + (z-x[i]).abs; x[i] = z;
    end
    count = count + 1
    p(x)
    if d < 0.00000001 then return x end
    if count > 10 then return nil end
  end
  return x
end
```

さて、収束はどうだろうか。100 だと多くて大変なので、10 回でやめるようにして経過を出力させ、前回のデータで動かしてみる。まず Jacobi 法。

```
irb(main):008:0> jacobi([[5,2,-1,19],[2,-3,2,-1],[1,1,-2,8]])
[3.8, 0.333333333333333, -4.0]
[2.86666666666667, 0.2, -1.93333333333333]
[3.33333333333333, 0.955555555555555, -2.46666666666667]
[2.92444444444444, 0.911111111111111, -1.85555555555556]
[3.06444444444444, 1.04592592592593, -2.08222222222222]
[2.96518518518518, 0.988148148148148, -1.94481481481482]
[3.01577777777778, 1.01358024691358, -2.02333333333333]
[2.9899012345679, 0.994962962962962, -1.98532098765432]
[3.00495061728395, 1.00305349794239, -2.00756790123457]
[2.99726502057613, 0.998255144032921, -1.99599794238683]
[3.00149835390947, 1.00084471879287, -2.00223991769547]
=> nil
```

続いて Gauss-Seidel 法。

```
irb(main):009:0> gaussseidel([[5,2,-1,19],[2,-3,2,-1],[1,1,-2,8]])
[3.8, 2.86666666666667, -0.666666666666667]
[2.52, 1.56888888888889, -1.95555555555556]
[2.78133333333333, 0.883851851851852, -2.16740740740741]
[3.01297777777778, 0.897046913580247, -2.04498765432099]
[3.0321837037037, 0.991464032921811, -1.98817613168724]
[3.00577916049383, 1.01173535253772, -1.99124274348423]
[2.99705731028807, 1.00387637786923, -1.99953315592135]
[2.99854281766804, 0.999339774497789, -2.00105870391709]
[3.00005234941747, 0.999329097000254, -2.00030927679114]
[3.00020650584167, 0.999931486033688, -1.99993100406232]
[3.00004120477406, 1.00007346714116, -1.99994266404239]
=> nil
```

こうしてみると、解 (3,1,-2) への収束は後者の方がかなり速いことが分かる。各周回において、おおむね後者は前者の誤差を ϵ とすると、後者の誤差は ϵ^2 くらいな感じである。

乱数の演習の解説は省略させてください。ゲーム作るのがメインだろうし。

2 常微分方程式の数値解法

2.1 常微分方程式

常微分方程式とは、未知関数とその導関数から成る等式で定義される方程式 (微分方程式) のうち、未知関数が (本質的に)1 つの変数を持つ場合を言う。導関数が 2 次以上の時は高階常微分方程式、方程式が複数ある場合は連立常微分方程式と呼ぶ。ここでは一番簡単な場合として $\frac{dy}{dx} = f(x,y)$ の形を持つものを扱う。

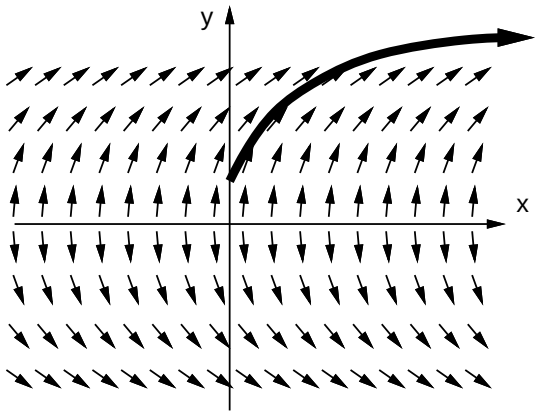

図 2: 常微分方程式と初期値問題

直観的に説明するなら、XY 平面の至るところにおいて傾きを表す矢印が定義されていて、その方向に沿った曲線を求める問題、という風に考えることができる (図 2)。もちろん、そのような曲線は無限にあるが (一般解)、任意の点 (x_0, y_0) を与えてそこを通る曲線を求めるとすれば、その曲線は 1 つだけになる (特殊解)。この、初期値を与えて特殊解を求める問題を**初期値問題**と呼ぶ。

簡単な例として、次の常微分方程式を考える。

$$\frac{dy}{dx} = \frac{1}{2y}$$

これを次のように変形する。

$$2y\ dy\ =\ 1\ dx$$

両辺を不定積分する。

$$\int 2y\ dy\ =\ \int 1\ dx$$

両辺とも積分の結果次のようになる。

$$y^2\ =\ x + C$$

よって次の式が求まり、これが一般解となる (もちろん $y = 0$ は除く)。

$$y\ =\ \pm\sqrt{x + C}$$

そして、たとえば点 $(1, 0)$ を通る特殊解を求めるなら、上式の y に 1、x に 0 を代入して $C = 1$ が求まるから $y = \sqrt{x + 1}$ が解となる。

2.2 Euler 法

前節のような簡単な常微分方程式の初期値問題は解析的に (数式の形で) 解が求まったが、現実の問題では解析的には解が求められないことが多い。そのような場合に、(数値積分と同様に) 数値的に解を求める方法 (数値解法) が使われる。

一番素朴な方法として、初期値 (x_0, y_0) における解曲線の接線の方向は $\frac{dy}{dx} = f(x, y)$ として分かっているわけなので、この方向に微小な値 h ぶんだけ動いた点を (x_1, y_1) とし、以下同様にして次々に曲線上の値を (近似的に) 求めて行くという方法が考えられる。これを **Euler 法**と呼ぶ。整理すると、次の式で x_i, y_i を計算していくのが Euler 法である。

$$x_{i+1} = x_i + h$$
$$y_{i+1} = y_i + h\ f(x_i, y_i)$$

これを使って x が指定値になるまで指定した分割数で計算して最後の y の値 ($(0, 1)$ から始めた場合 $\sqrt{x + 1}$ になるはず) を返す Ruby プログラムを示す。

```
def euler(xmin, y, xmax, count)
  h = (xmax-xmin).to_f / count
  count.times do |i|
    # x = xmin + h * (i+1)
    y = y + h * 0.5/y  # f(x,y) = 1/2y
  end
  return y
end
```

なお、この方程式の場合は x を使わない ($f(x, y)$ が x に依存しない) ので、x の計算はコメントアウトしてある。

実際に動かしてみよう。x として 1 を指定することで、2 の平方根を計算するようにし、刻み数を変えることで精度の変化を見ている。

```
irb(main):003:0> euler(0,1,1,100)
=> 1.41482796736591
irb(main):004:0> euler(0,1,1,1000)
=> 1.41427484589246
irb(main):005:0> euler(0,1,1,10000)
=> 1.41421968916029
```

まあ、あんまりよくはない。覚えていない人のため、1 桁の平方数でない数の平方根を掲げておく。

$\sqrt{2} \approx 1.41421356237309504880168872420969807856 96$
$\sqrt{3} \approx 1.73205080756887729352744634150587236694 28$
$\sqrt{5} \approx 2.23606797749978969640917366873127623544 06$
$\sqrt{6} \approx 2.44948974278317809819728407470589139196 59$
$\sqrt{7} \approx 2.64575131106459059050161575363926042571 02$
$\sqrt{8} \approx 2.82842712474619009760337744841939615713 93$

double 型の精度は 16 桁くらいなので、もうちょっと精度良く求まって欲しいわけだが… Euler 法の場合、「個々の点における」接線を延長して次の点を求めているため、微分の向き (矢線の向き) が揃っている (まっすぐな) 所ではいいけれど、カーブしているところではどんどん「外側」にずれて行ってしまう。これを補う方法を次に述べる。

2.3 Runge-Kutta 法

Euler 法では各ステップの始点における接線を延長して次の点を求めていたが、それでは解曲線がカーブしているときに誤差が大きい。そこで、始点と終点の両方で接線の向きを求め、それを平均すればより正確になると考える。しかし実際には、各ステップの終点は「これから求める」ので分からない。そこで Euler 法で終点の近似値を求め、そこの傾きを使い、これと始点での傾きとの平均を取る (図 3)。つまり次のようにするわけである。

$$x_{i+1} = x_i + h$$
$$k1 = h\ f(x_i, y_i)$$
$$k2 = h\ f(x_i + h, y_i + k1)$$
$$y_{i+1} = y_i + \frac{1}{2}(k1 + k2)$$

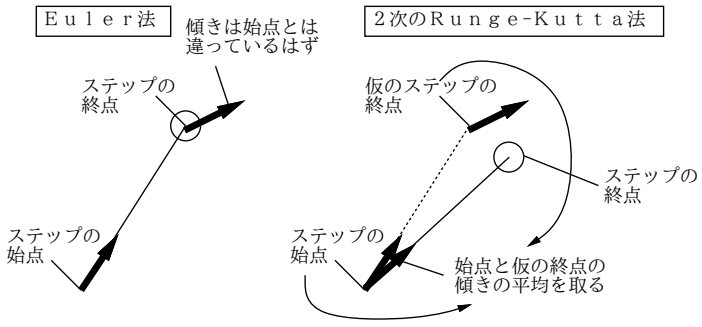

図 3: Euler 法と 2 次の Runge-Kutta 法

これを **2 次の Runge-Kutta 法**と呼ぶ。これを Ruby プログラムにしたものを示しておく。

```
def rungekutta2(xmin, y, xmax, count)
  h = (xmax-xmin).to_f / count
  count.times do |i|
    # x = xmin + h * (i+1)
    k1 = h * 0.5/y
    k2 = h * 0.5/(y+k1)
    y = y + 0.5*(k1+k2)
  end
  return y
end
```

これによる計算を示しておく。ずっと精度がよくなっていることが分かる。

```
irb(main):013:0> rungekutta2(0,1,1,100)
=> 1.41421356445272
irb(main):014:0> rungekutta2(0,1,1,1000)
=> 1.41421356237517
irb(main):015:0> rungekutta2(0,1,1,10000)
=> 1.41421356237309
```

さらに正確さを増すため、h の半分だけ上記の方法で進み、そこから終点までを 2 番目の近似値で進み、始点、最初の近似値、2 番目の近似値、終点の近似値を 1:2:2:1 の比率で混ぜる方法があり、**4 次の Runge-Kutta 法**として知られている。単に Runge-Kutta 法と言った場合はこちらを指すことが多い。

$$
\begin{aligned}
&x_{i+1} = x_i + h \\
&k1 = h\ f(x_i,\ y_i) \\
&k2 = h\ f(x_i + \tfrac{h}{2},\ y_i + \tfrac{k1}{2}) \\
&k3 = h\ f(x_i + \tfrac{h}{2},\ y_i + \tfrac{k2}{2}) \\
&k4 = h\ f(x_i + h,\ y_i + k3) \\
&y_{i+1} = y_i + \tfrac{1}{6}(k1 + 2k2 + 2k3 + k4)
\end{aligned}
$$

なぜ 1:2:2:1 の比率がいいかとかはここでは説明しないが、Taylor 展開による 1 次の項のあてはめが Euler 法、2 次の項まであてはめるのが 2 次の Runge-Kutta 法、4 次の項まであてはめるのが 4 次の Runge-Kutta 法である。

ついでに誤差についてもおおまかに説明すると、Euler 法では各ステップの誤差が (1 次まであてはめた残りなので)$O(h^2)$、それを $\frac{1}{h}$ ステップ累計するため全体の誤差は $O(h)$ となる。同様に 2 次の Runge-Kutta 法では各ステップ $O(h^3)$、全体で $O(h^2)$、4 次の Runge-Kutta 法では各ステップ $O(h^5)$、全体で $O(h^4)$ となる。これを言い替えると、h を $\frac{1}{2}$ にしたとき Eular 法では全体の誤差がほぼ $\frac{1}{2}$ になるのに対し、2 次の Runge-Kutta 法では $\frac{1}{2^2}$、4 次の Runge-Kutta 法では $\frac{1}{2^4}$ になる。

なお、刻みをどんどん細かくする程いいかというと、細かくする程計算時間が掛かるだけでなく、丸め誤差や情報落ち誤差が累積するので、結局精度もある程度以上は上がらなくなる。このため、2 次や 4 次の Runge-Kutta 法のように最初から性能のよい公式を使うことは大変有利だと言える。

演習 1 Eular 法または 2 次の Runge-Kutta 法のプログラムを打ち込み、これらでさまざまな値の平方根を刻み幅を変えて計算し、上で説明した誤差に関する性質が成り立っているかどうか確認してみよ。

演習 2 さらに 4 次の Runge-Kutta 法のプログラムも用意し、上と同様に検討せよ。

3 オブジェクト指向

3.1 オブジェクト指向とは

皆様はここまでに、配列、レコードなどのデータ構造を用意し、それを操作するメソッドを複数組み合わせてアルゴリズムを実現する、という形でプログラムを作成してきた。最初の方で説明したように、このようなスタイルを手続き型のプログラミングスタイルという。

手続き型のプログラミングスタイルは長い間主流として使われて来たが、近年のようにプログラムが大きくなり複雑化してくると、次のような弱点が問題になってきた。

- 「データ構造」と「手続き」が分離しているので、どの手続きがどのようなデータ構造を扱う、というのがごちゃごちゃになりやすい。
- 手続きが多数になると、どの手続きが何だったのかの構造を憶えて管理するのが大変になる。
- データ構造の内容がどこからでも分かりアクセス可能なので、本来なら構造を直接いじらない (専用の手続きに頼む) ところでまちがっていじってしまいトラブルをまねくことがある。

オブジェクト指向 (Object-orientation, O-O) とは、これらの弱点を克服すべく手続き型言語を拡張した概念であり、今日では多くのプログラミング言語に取り込まれている。その要点は何かというと、プログラムが扱うデータをさまざまな「もの」として扱うことにある。

我々が日常扱っている「もの」にはそれぞれ固有の機能や特性があり、我々は「内部構造」には関わらなくてもこれらの「もの」の機能や特性を活用することができる。たとえばイスであれば「座る」「高さを調節」「移動させる」などの操作ができるし、ペンであれば「キャップをつける/外す」「描く」などの操作ができるし、それぞれ固有の色などもある。しかしこれらを利用したり参照するのに、イスやペンの内部構造を理解している必要はない。

3.2 クラスとインスタンス

では次に、前節で述べたような「もの」を言語上でどのように表すかを考えてみまよう。1 つの素直な考え方として、ものには「種類」(**クラス**) があるものと考え、その種類ごとに「どんな性質を持つか」「どのような操作ができるか」などを定義していく、というものがある。このような考え方に従うオブジェクト指向言語を**クラス方式**のオブジェクト指向言語、と呼ぶ。Ruby も Java も C++もクラス方式のオブジェクト指向言語である。

では次に、ものの種類の定義、つまり**クラス定義**には何が含まれるべきかを考えてみよう。先に述べたように、それぞれの「もの」には固有のデータと固有の操作があるので、それを「変数」と「メソッド」で表すのが自然だといえる。

ただし、ここでいう変数とメソッドは、これまで使って来た変数やメソッドとは少し違っている。つまり、あるクラスを定義して、そのクラスに所属する「もの」(プログラミング言語の言葉ではこれを**インスタンス**ないし「実体」と呼びます) を定義したとすると、クラスで定義した変数やメソッドは「そのクラスのインスタンスに対して適用される」ものとなります。これを**インスタンス変数**、**インスタンスメソッド**と呼ぶ。

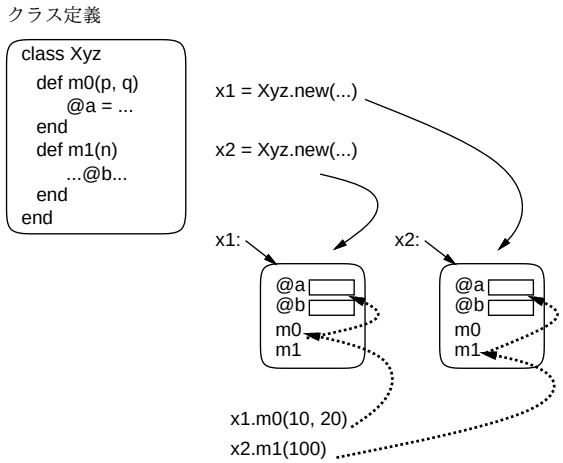

図 4: クラスとインスタンス

たとえば図 4 では、クラス定義 `Xyz` を「ひな型」として 2 つのインスタンスを生成し、変数 `x1` と `x2` に入れている。このとき、この 2 つのインスタンスは内部に持っているインスタンス変数群も使用できるメソッド群も同じ (つまりクラス定義に書かれている通り) であるが、インスタンスとしては別個、つまりインスタンス変数群はそれぞれ別個になっている。

ここで、メソッド呼び出し記法により「`x1.m0(...)`」「`x2.m1(...)`」のようにそれぞれのインスタンスメソッドを呼び出したとすると、それらのインスタンスメソッドの中でインスタンス変数を参照したときは、それぞれ自オブジェクトのインスタンス変数が使われることになる。

よく使われる「動物」の例で考えると、「犬」というクラスを作って、そこで「名前」「走っている速さ」というインスタンス変数を持たせたとすると、どの犬もこれら 2 つのインスタンス変数を持っているという点は同じだが、そこに格納されている値、つまりそれぞれの犬の名前やそれぞれの犬の走っている速さはどの犬かによって、つまりインスタンスによって違う、というわけである。

3.3 Ruby による簡単なクラスの定義

では Ruby の場合について説明しよう。まずクラスは次の構文により定義する。

```
class クラス名
  ...
end
```

クラス名は必ず英大文字で始めることになっている。そして、この中にメソッドの def を書くと、自動的にインスタンスメソッドになり、「オブジェクト . メソッド名 (...)」で呼び出せる (この書き方は既に Array の push だとか Number の times だとか山のように使っていたが、それが自分で定義できるということ)。最後に、インスタンス変数はこれまでの変数と異なり、名前の最初が「@」で始まる。以上です。

おっとあと 1 つだけ。クラスからインスタンスを作るには

```
クラス名.new(...)
```

という特別なメソッド呼び出しを使うが、このときもしインスタンスメソッドの中に initialize という名前のものがあればそれが呼び出され、その時 new に渡したパラメタがそっくりそのまま渡されて来る。つまりまさに、初期化のためにこのようになっているわけだ。

では、非常に簡単なクラスの定義を見てみよう。これは上で挙げた「犬」のクラスの例である。

```
class Dog
  def initialize(name)
    @name = name; @speed = 0.0
  end
  def talk
    puts('my name is ' + @name)
  end
  def addspeed(d)
    @speed = @speed + d
    puts('speed = ' + @speed)
  end
end
```

initialize では name(名前) を受け取り、その値でインスタンス変数@name を初期化する。インスタンス変数@speed は 0 に設定します。メソッド talk では自分の名前を喋る (喋る犬?)。addspeed では渡された値だけスピードを増してそれを表示する。動かしてみよう。

```
irb(main):021:0> a = Dog.new('pochi')
=> #<Dog:0x81b5b2c @name="pochi", @speed=0.0>
irb(main):022:0> b = Dog.new('tama')
=> #<Dog:0x81b049c @name="tama", @speed=0.0>
irb(main):023:0> a.talk
my name is pochi
=> nil
irb(main):024:0> b.talk
my name is tama
=> nil
irb(main):025:0> a.addspeed(5.0)
speed = 5.0
=> nil
irb(main):026:0> b.addspeed(8.0)
```

```
speed = 8.0
=> nil
irb(main):027:0> a.addspeed(10.0)
speed = 15.0
=> nil
```

ポチのインスタンスとタマのインスタンスは別のもので、名前や速度を別個に持っていることが分かると思う。このように「もの」単位で扱えるところがオブジェクト指向の特徴だということ。

演習 3 この例題を打ち込んでそのまま動かしなさい。動いたら、「ほえる」メソッド bark(引数は無し) と、「ほえる回数」を設定するメソッド setcount(引数は回数) を追加しなさい。とくに設定しない場合は 3 回ほえるものとします (もちろん、ほえる回数を憶えるインスタンス変数を追加する必要もあるでしょう)。

3.4 例: 有理数クラス

ワンと吠える犬ではあんま役に立ちそうな気がしないので、もう少し有用なものを作って見ることにしよう。これまで、整数でない数値計算には誤差がつきものだという話をさんざんして来ましたよね。では、浮動小数点計算をするのではなく、割り算をしたときに「分子, 分母」の分数の形で保持すれば誤差なしで済む。もちろん、加減算のときは通分して計算する。そのようなクラスを作ってみた。

```
class Rational
  def initialize(a = 1, b = 1)
    @a = a; @b = b
    if b == 0 then @a = 1; return end
    if b < 0 then @a = -a; @b = -b end
    if a == 0 then @b = 1
    elsif a > 0 then @a = a / gcd(a,b); @b = b / gcd(a,b)
    else             @a = a / gcd(-a,b); @b = b / gcd(-a,b)
    end
  end
  def getDivisor
    return @b
   end
  def getDividend
     return @a
  end
  def to_s
    return "#{@a}/#{@b}"
  end
  def +(r)
    return Rational.new(@a*r.getDivisor+r.getDividend*@b, @b*r.getDivisor)
  end
  def gcd(x, y)
    while true do
      if x > y then x = x % y; if x == 0 then return y end
      else          y = y % x; if y == 0 then return x end
      end
    end
  end
```

```
end
```

つまりインスタンス変数@aと@bに分子と分母をそれぞれ保持しておくわけだ。initializeのパラメタに代入が書いてあるが、これは「デフォルト値」つまりそのパラメタを省略した場合はこの値を使ってねという意味になる。だから「Rational.new(3)」で $\frac{3}{1}$ になるわけだ。その後initializeの中身がごちゃごちゃしているが、これは (1) 分母が 0 のとき (不定) は分子を 1 とする、(2) 分母は 0 でないなら常に正とする (負の数は分子が負)、(3) 値ゼロは $\frac{0}{1}$ で表す、(4) 必ず既約分数にする、という正規化 (なるべく形を揃えること) を行っているから。メソッド gcd はご存じの通りのものが末尾に入れてある。

次に、分母だけ、分子だけを取り出したい場合のために、メソッド getDivisor、getDivdend を用意した。このような、インスタンス変数をアクセスするだけのメソッドのことを**アクセサ**と呼ぶ。逆に言えば、アクセサがなければインスタンス変数の内容は外部からは参照できない。このようにしておくと、後で表現方法を変えた時にも外部に影響が及ばないなどの利点がある。

また、文字列への変換メソッド to_s も用意した。これは puts などによる打ち出しなどの時に自動的に「a/b」という形の文字列を生成できるので用意しておくと便利である。

そして、演算としてはとりあえず加算だけ用意した。加算はやはり「+」で表したいので、メソッド名を「+」にしてある。このようにして、演算子を定義できるのは Ruby の特徴の 1 つである (もっとも C++などでもこれができるが)。では動かしてみよう。

```
irb(main):004:0> a = Rational.new(3,5)
=> #<Rational:0x81f978c @b=5, @a=3>
irb(main):005:0> puts a
3/5
=> nil
irb(main):006:0> b = Rational.new(8,7)
=> #<Rational:0x81f00d8 @b=7, @a=8>
irb(main):007:0> puts b
8/7
=> nil
irb(main):008:0> puts a+b
61/35
=> nil
```

確かに、通分して計算してくれている。

演習 4 有理数クラスをそのまま打ち込んで動かせ。動いたら、四則の他の演算も追加し、動作を確認せよ。できれば、これを用いて浮動小数点では正確に行えない「実用的な」計算が正確にできることを確認してみよ。

演習 5 複素数を表すクラス Complex を定義し、動作を確認せよ。これを用いて何らかの役に立つ計算をしてみられるとなおよい。

演習 6 クラス定義を活用した Ruby「面白い」プログラムを作って動かせ。面白さの定義は各自に任されるものとする。

A　本日の課題 7A

「演習 1」～「演習 3」から 1 つ選び、今日中に久野までメールで送ってください。

1. Subject: は「Report 7A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 「演習 1」～「演習 3」で動かしたプログラムどれか 1 つのソース。
4. 検討結果のまとめ (とできれば簡単な考察)
5. 以下のアンケートの回答。

Q1. 常微分方程式の数値解法について納得しましたか。

Q2. クラスの概念やその機能について納得しましたか。

Q3. 本日の全体的な感想と今後の要望をお書きください。

B　次回までの課題 7B

次回までの課題は「演習 1」〜「演習 6」(の、7A で提出していないもの) から 2 つ以上選んで報告することです。

各課題のために作成したプログラムはすべてレポートに掲載してください。レポートは授業開始時刻の **10 分前** までに久野までメールで送付してください。

1. Subject: は「Report 7B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 1 つ目のプログラムのソース。
4. その説明と分析/考察。
5. 2 つ目のプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

Q1. クラス定義が書けるようになりましたか。

Q2. この先、このクラス定義の機能がどのように使われると予想しますか。

Q3. 課題に対する感想と今後の要望をお書きください。