情報科学 2011　久野クラス　＃7

久野　靖*

2011.11.18

はじめに

この科目も早くも折り返し点の第7回となりました。そして、「次回までの課題 (B 課題)」があるのは今回と次回で終わりで、あとは A 課題のみとなります。さらに、来週は駒場祭なので B 課題の期限は 2 週間後となります。そういうわけで、あとちょっと、頑張ってください。

今回は、今日のプログラミングにおいて重要な概念となっているオブジェクト指向について取り上げます。また、乱数と乱数を使ったアルゴリズムについても紹介します。

1　前回の演習問題解説

1.1　演習 1a — ガウス–ジョルダンの消去法

演習 1a は、ガウス–ジョルダンの消去法のプログラムを作ることでしたが、これはガウス法を削ってちょっと直せばできます:

```
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
```

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 で囲みました。さらに、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 行入れています。

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(かそれに近い値) だとうまく行かないので、別の列と入れ換えて進めるという話でした。そこで、いちいち判断するのでなく、x_i を消去する時に

*筑波大学大学院経営システム科学専攻

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ました。そのために、selectpivot という下請けメソッドを追加しています。[1]

```
def selectpivot(a, i, n)
  max = a[i][i].abs; k = i;
  (i+1).step(n-1) do |j|
    if a[j][i].abs > max then max = a[j][i].abs; k = j end
  end
  return k
end
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
    swap(a, i, k); swap(idx, i, k)
    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) do |i| k = idx[i]; a[k][n] / a[k][k] end
  return b
end
def swap(a, i, j)
  x = a[i]; a[i] = a[j]; a[j] = x
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 を返します。そうでない場合は i 列と k 列を交換してからこれまでどおり進めます。

ただし、交換を行うと列の順序が入れ替わってしまうので、最後に解を出力する時、どの変数が何番目に入っているか分からなくなってしまいます。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、[2]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並べるようにしています。[3]

1.3 演習 2 — 反復法

演習 2a と 2b については色々入れて観察するだけですが、要は非対角成分が大き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、分かればよいのではないでしょうか。

演習 2c ですが、ヤコビ法からガウス–ザイデル法に直すには、配列 x1 を使わずに x だけで済ませるように直せばよいわけです: [4]

[1] ところで、係数の絶対値が最大の列を持ってきたのに、その係数が 0(ないしそれに近い値) だったら? それは方程式が不定/不能だということなので、ついでに演習 1c も組み込みました。

[2] その作成は、Array.new にブロックを渡して初期値を計算させる方法で、渡された添字番号をそのまま返すようにして実現しました。

[3] 結果配列の作成時も Array.new にブロックを渡し、その中で i 番目の解として交換語の idx[i] 行目の結果を入れるようにしています。

[4] その時、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 に入れるようにしました。

```
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、収束はどうでしょうか。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、先のデータで動かしてみます。まずヤコビ法:

```
irb> 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
```

続いてガウス–ザイデル法:

```
irb> 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
```

比較すると、解 (3,1,-2) への収束は後者の方が速いことが分かります。各周回において、おおむね前者の誤差を ϵ とすると、後者の誤差は ϵ^2 という感じですね。

1.4 演習 3-4 — オイラー法とルンゲ–クッタ法の誤差

まとめてやるために、まず 4 次のルンゲ–クッタ法のコードを作成しました (公式どおりにあてはめるので難しくはないですね):

```
def rungekutta4(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+0.5*k1)
    k3 = h * 0.5/(y+0.5*k2)
    k4 = h * 0.5/(y+k3)
    y = y + (k1+2*k2+2*k3+k4)/6.0
  end
  return y
end
```

では、$\sqrt{2}$ を刻み幅を変えて計算してみましょう。まずオイラー法から:

```
irb> euler(0,1,1,10)
=> 1.42051979929104    ←誤差 0.006306
irb> euler(0,1,1,20)
=> 1.41732103648967    ←誤差 0.003107
irb> euler(0,1,1,40)
=> 1.41575616718198    ←誤差 0.001542
irb> euler(0,1,1,80)
=> 1.41498211573229    ←誤差 0.000768
```

おおむね、h が $\frac{1}{2}$ になると誤差も $\frac{1}{2}$ になっていると言えるでしょう。では 2 次のルンゲ–クッタ法:

```
irb> rungekutta2(0,1,1,10)
=> 1.41421571099439      ←誤差 0.00000214862
irb> rungekutta2(0,1,1,20)
=> 1.41421382625237      ←誤差 0.00000026387
irb> rungekutta2(0,1,1,40)
=> 1.41421359505333      ←誤差 0.00000003268
irb> rungekutta2(0,1,1,80)
=> 1.41421356643876      ←誤差 0.00000000406
```

今度は、h が $\frac{1}{2}$ になると誤差はおよそ $(\frac{1}{8})$ になっているようです。最後に 4 次のルンゲ–クッタ法:

```
irb> rungekutta4(0,1,1,10)
=> 1.41421357656941      ←誤差 0.0000000141963166
irb> rungekutta4(0,1,1,20)
=> 1.41421356323698      ←誤差 0.0000000008638950
irb> rungekutta4(0,1,1,40)
=> 1.41421356242633      ←誤差 0.0000000000532359
irb> rungekutta4(0,1,1,80)
=> 1.4142135623764       ←誤差 0.0000000000033080
```

今度は、h が $\frac{1}{2}$ になると誤差はおよそ $\frac{1}{16}$ になっているようです。こうしてみると、2 次のルンゲ–クッタ方だけ理論値より良くなっていますが (理論的には $\frac{1}{4}$ のはず)、これは解曲線が 2 次曲線なので例外的にあてはまりがよいためと思われます。

2 オブジェクト指向

2.1 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ました (図 1 左)。最初の方で説明したように、このようなモデルを**手続き型計算モデル**、このモデルに基づくプログラミング言語を**手続き型言語**と言います。

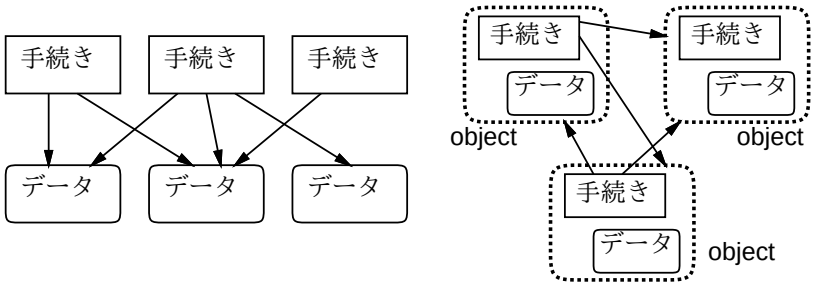

図 1: 手続き型モデルとオブジェクト指向

こ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きましたが、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ました:

- データ構造と手続きが分離していて、両者の対応が取りにくい。
- 手続きが複雑になると、どの部分が何を行っているかの把握が困難になる。
- 各データ構造がどの手続きからでも (原理的には) アクセスできるため、本来アクセスすべきでないデータ構造に触ってしまうことによるトラブルが起きやすい。

オブジェクト指向 (object-orientation) は、上記の点を克服すべく手続き型モデルを拡張した概念で (図 1 右)、プログラムが扱う対象を多様な**もの**、ないし**オブジェクト** (object) として捉える考え方です。

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できます。たとえばイスであれば「座る」「高さを調節」「移動させる」などの操作ができますし、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、それぞれ固有の色などもあります。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ありません。プログラミングもこれと同様にできれば人間にとってずっと扱いやすくなる、というのがオブジェクト指向の基本的なアイデアです。

今日では多くのプログラミング言語がオブジェクト指向を取り入れています。そのような言語 (**オブジェクト指向言語**) では、手続きとそれが扱うデータが組になっているため対応がつけ易く、個々の手続きは自分の担当しているデータのみを直接扱うため簡潔に保ちやすく、データは対応する手続き以外からは操作されないため、不用意に壊される心配が少なくなります。データを外部から直接アクセスされないようにすることを**カプセル化** (encapsulation) ないし**情報隠蔽** (information hiding) と呼びます。

2.2 クラスとインスタンス

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しょう。ものには種類ないし**クラス** (class) 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」を定義していく、というのが 1 つの方法です。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** (class based) のオブジェクト指向言語と呼びます。Ruby、Java、C++などの言語はクラス方式のオブジェクト指向言語です。

なお、別の方式として**プロトタイプ方式** (prototype based) のオブジェクト指向言語があります。これは、「お手本」となるオブジェクトを (概念的に) コピーして類似したオブジェクトを用意する方式で、JavaScript などが採用しています。

では、ものの種類の定義、つまり**クラス定義** (class definition) には何が含まれるべきでしょうか (図 2)。上述のように、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**手続き**ないし**メソッド**で表すのが自

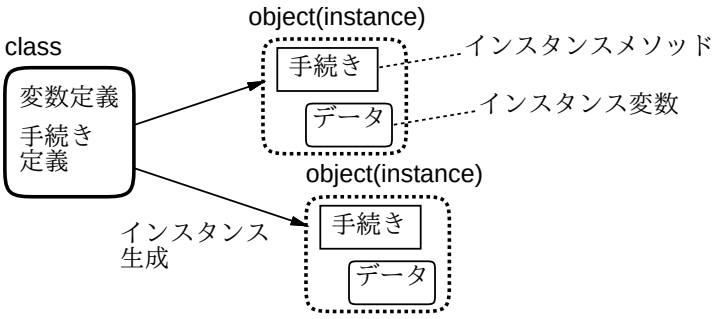

図 2: クラスからインスタンスを生成

然な方法です。これらをそれぞれ、**インスタンス変数** (instance variable)、**インスタンスメソッド** (instance method) と呼びます。

ただし、ここで言う変数とメソッドは、これまで使ってきた変数やメソッドとは少し違っています。つまり、あるクラスを定義し、それをもとに「そのクラスに所属するもの」 — オブジェクト指向言語の言葉で言えば**インスタンス** (instance — 実体と呼ぶこともあります) を生成したとすると、クラス内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付随した」ものとなります。

たとえば図 3 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` に入れています。この時、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ですが、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ます。つまりクラス定義に書かれているとおりのインスタンス変数群とインスタンスメソッド群を持つということです。

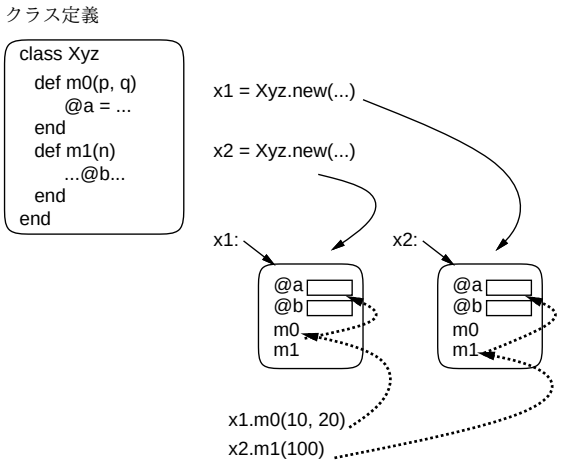

図 3: クラスとインスタンス

そして、インスタンスメソッドを呼び出す時には、単にメソッド名を言ったのでは「どのインスタンスに付随する」メソッドかが特定できないので、インスタンス x に対して「x. メソッド名」という形で指定します。これを**メッセージ送信記法** (message sending) と呼びます。ここまでにもこの記法は、配列などさまざまな (Ruby が提供してくれている) オブジェクトの機能を呼び出す時に利用していました。

そして、メッセージ送信記法で「`x1.m0(...)`」「`x2.m1(...)`」のように インスタンスメソッドを呼び出すと、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時は、それぞれ `x1`、`x2` のインスタンス変数が使われることになります。

たとえば、「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ですが、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、というわけです。

あと、Ruby ではクラスもオブジェクトなので、クラスに直接付随するメソッドである**クラスメソッド** (class method)、クラスに直接付随する変数である**クラス変数** (class variable) も存在します。クラスメソッドを呼び出す場合はメッセージ送信記法のオブジェクトのところにクラスの名前を指定します。

2.3 Ruby による簡単なクラスの定義

では Ruby の場合についてクラス定義の方法を説明しましょう。まずクラスは、次の構文により定義します:

```
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ます。そして、この中にメソッド定義を書くと、自動的にインスタンスメソッドになり、メッセージ送信記法で呼び出せるようになります。また、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「@」で始まります。

そして最後に、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 を使います。この時、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きます。つまり名前どおり、初期化のためにこのような仕組みになっているわけです。

ここでは使いませんが、クラスメソッドを定義する場合は「def クラス名. メソッド名 … end」のような def をクラスの外側に書いて定義します。また変数名を「@@」で始めるとその変数はクラス変数になります。

では、簡単なクラス定義の例を見てみましょう (これは先に説明で使った「犬」をクラスとして定義したものです):

```
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.to_s)
  end
end
```

initialize では名前を受け取り、その値でインスタンス変数@name を初期化します。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ります (喋る犬?)。addspeed では渡された値だけスピードを増して、それを表示します。動かしてみましょう:

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

```
speed = 5.0
=> nil
irb> b.addspeed(8.0)
speed = 8.0
=> nil
irb> 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お分かりになると思います。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なのです。

演習 1 この例題を打ち込んで動かせ。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) を追加せよ。とくに設定しない場合は 3 回ほえるものとする。[5]

演習 2 次のような機能と使い方を持つクラスを作成せよ。使用例の通りに使えることを確認すること。

a. 「覚える」機能を持つクラス Memory。put(x) で新しい内容を記憶させ、get で記憶内容を取り出す。

```
irb> m1 = Memory.new           # 作る
=> #<Memory:0x81d59e0 @mem=nil>
irb> m1.put(5)                 # 5 を覚えさせる
=> 5                           # put の返値は何でもいいことにする
irb> m1.get                    # 取り出す
=> 5                           # 5
irb> m1.get                    # 再度取り出す
=> 5                           # やはり 5
irb> m1.put(10)                # 10 を覚えさせる
=> 10
irb> m1.get                    # 取り出す
=> 10                          # 10
```

b. 「文字列を連結していく」クラス Concat。add(s) で文字列 s を今まで覚えているものに連結する (最初は空文字列から始まる)。get で現在覚えている文字列を返す。reset で覚えている文字列を空文字列にリセット。

```
irb> c = Concat.new            # 作る
=> #<Concat:0x81c7e94 @str="">
irb> c.add("This")             # 追加
=> "This"
irb> c.add("is")               # 追加
=> "Thisis"
irb> c.get                     # 取り出す
=> "Thisis"
irb> c.add("a")                # 追加
=> "Thisisa"
irb> c.reset                   # リセット
=> ""
irb> c.add("pen")              # 追加
=> "pen"
irb> c.get                     # 取り出し
=> "pen"
```

[5] 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があるはずです。

なお、文字列どうしを連結するのは「+」でできます。

c. 「最大 2 つ覚える」機能を持つクラス Memory2。put(x) で新しい内容を記憶させ、get で記憶内容を取り出す。2 回取り出すと 2 回目はより古い内容が出てくる。取り出した値は忘れる。覚えている以上に取り出すと nil が返る。

```
=> #<Memory2:0x80fdab8 @mem2=nil, @mem1=nil>
irb> m2.put(1)    # 1 を入れる
=> 1
irb> m2.put(3)    # 3 を入れる
=> 3
irb> m2.put(5)    # 5 を入れる
=> 5
irb> m2.get       # 取り出す → 5
=> 5
irb> m2.get       # 取り出す → 3
=> 3
irb> m2.get       # 取り出す → nil (2 つまでしか覚えてない)
=> nil
irb> m2.put(7)    # 7 を入れる
=> 7
irb> m2.put(9)    # 9 を入れる
=> 9
irb> m2.get       # 取り出す → 9
=> 9
irb> m2.put(11)   # 11 を入れる
=> 11
irb> m2.get       # 取り出す → 11
=> 11
irb> m2.get       # 取り出す → 7
=> 7
```

もし興味があれば、「最大 N 個覚える」もやってみるとよい。

2.4 有理数クラス

今度は、もう少し有用なものを作ってみます。これまで、実数の計算には誤差がつきものだという説明をしてきましたね。具体的には、浮動小数点計算では割り切れない除算は循環小数になるので、必ず誤差が生じます。

そこで代わりに、数値を $\frac{\text{分子}}{\text{分母}}$ という形で保持すれば誤差なく除算結果を保持できるはずです (もちろん、加減算の時は通分して計算し、最後に約分します。そしてもちろん、$\sqrt{2}$ や π などの無理数は扱えません)。

そのような**有理数** (rational number) クラスを作ってみましょう。このクラスでは、インスタンス変数@a と@b に分子と分母をそれぞれ保持するようにしています:

```
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 get_divisor
    return @b
   end
  def get_dividend
     return @a
  end
  def to_s
    return "#{@a}/#{@b}"
  end
  def +(r)
    return Rational.new(@a*r.get_divisor+r.get_dividend*@b,
                        @b*r.get_divisor)
  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
```

initialize のパラメタに代入が書いてありますが、これは**デフォルト値** (default value) 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ります。ですから、「Rational.new(3)」で $\frac{3}{1}$ になるわけです。initialize の中身がごちゃごちゃしていますが、これは (1) 分母が 0 の時 (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** (normalization — なるべく形を揃えること) を行っているからです。

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 get_divisor、get_dividend を用意しました。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** (accessor) と呼びます。Ruby では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ません。これにより、カプセル化が実現され、また後で内部のデータ表現を変えた時にも外部に影響が及ばないで済みます。

また、文字列への変換メソッド to_s も用意しました。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す。そして、演算としてはとりあえず加算だけを用意しました。加算は「+」で表したいので、メソッド名を「+」にしてあります。このようにして、演算子を定義できるのは Ruby の特徴の 1 つです。ただし、C++などの言語でも演算子定義が可能です。

では動かしてみましょう:

```
irb> a = Rational.new(3,5)
=> #<Rational:0x81f978c @b=5, @a=3>
irb> puts a
3/5
=> nil
irb> b = Rational.new(8,7)
=> #<Rational:0x81f00d8 @b=7, @a=8>
irb> puts b
8/7
=> nil
irb> puts a+b
```

```
61/35
=> nil
```

確かに、通分して計算してくれていますね。なお、なぜputsで打ち出しているかというと、putsは引数を文字列に変換して出力するためto_sを呼んでくれるからです。単にirbの機能で打ち出させるのだと、オブジェクトを表す「#<Rational ....>」というのが表示されてしまいます。

演習 3 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 4 **複素数** (complex number) 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 5 クラス定義を活用した「面白い」Ruby 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。

3 乱数とランダムアルゴリズム

3.1 乱数とは

乱数とは、簡単に言えば「ランダムな数」です。より正確に言うと、「ある分布に従う、互いに独立な事象を表す、確率変数の実現値の列」のことを**乱数列** (randam sequence) と言い、その中の個々の値が乱数です。互いに独立ということは、ある点までの乱数列が分かったからといって、次の乱数がいくつであるかは予測できないことを意味します。

また、**分布** (distribution) とは、どの範囲の値がどのくらい出現しやすいかを表すものです (図 4)。たとえば、区間 $[0,1)$ の**一様分布** (uniform distribution) であれば、乱数の範囲は 0 以上 1 未満で、その間のどの数も同じくらいの確からしさで出現します。これを**一様乱数** (uniform random number) と言います。また、偏りのないサイコロを振って出る目の数は 1 以上 6 以下の整数値ですが、どの数も同じ確率で出現するため、これも一様乱数です。この他によく使われる乱数としては、**正規分布** (normal distribution) に従う**正規乱数** (normal random numbers) があります。[6]

図 4: 乱数と分布

3.2 擬似乱数

擬似乱数 (pseudorandom number) とは、プログラムで順次計算を行うことで生成される数値の列で、乱数列のように思えるものを言います。さんざん学んできたように、プログラムの動作は完全に決定的なものなので、「でたらめな」数を生成するのは思いのほか難しいものです。

擬似乱数のアルゴリズムの代表的なものとして、次のものがあります:

- **自乗採中法** (middle-square method) — w ビットの数を 2 乗すると $2w$ ビットになるので、そこから中央付近の w ビットを取り出して「次の数」とする。これを繰り返すことで w ビットの乱数列を生成する。[7]
- **線形合同法** (linear congruential method)— $x_{i+1} = (x_i \times a + c)\ mod\ m$ により次々に値を生成していく。[8]

[6] 正規分布は中央が一番高く両側にすそを引いたツリガネ形の分布であり、試験の**偏差値** (standard score) などでおなじみです。試験が受験者の集団に対して易しすぎたり難しすぎたりヘンな問題だったりすると、分布が綺麗な正規分布でなくなるので偏差値による順位推定が役に立たなくて問題になったりします。

[7] 自乗採中法は古くからあるアルゴリズムですが、あまりよい乱数列は生成できないことが知られています。

[8] 線形合同法は MT の発明以前は主流のアルゴリズムでした。ただし、よい擬似乱数とするためには、パラメタ a, c, m の選定に注意が必要です。

- **メルセンヌツイスター** (Mersenne twister)、MT — 1997 年に松本　眞、西村拓士が開発した乱数アルゴリズム。

どのようなアルゴリズムでも、計算方法が決まっている以上、順次 x_i を生成していくうちに前に出てきたものが再度現れたら、それ以降の列は前と同じものの繰り返しになります。これを**周期** (period) といい、もちろん周期が長いものが望まれます。MT は 32 ビットで $2^{19937}-1$ という長い周期を保証できるという性質を持つという点で画期的でした。さらに周期の他に統計的独立性の検定などもクリアしています。

Ruby では、`rand(`N`)` で 0 以上 N 未満の整数の一様乱数、引数なしの `rand` で区間 $[0,1)$ の実数一様乱数が得られます。[9]

このほか、最近では**オペレーティングシステム** (operating system — OS、コンピュータ上で常に稼働し資源管理を行っている基本ソフトウェア) が乱数機能を提供してくれている場合があります。OS の乱数機能は、ユーザのキーボード入力やディスクの動作などの外部割り込みに基づいた「ランダムさ」を活用するので、擬似乱数のような周期の問題がありませんが、速い速度で乱数を生成消費すると「ランダムさ」の供給が追い付かなくなることがあります。また、最近では CPU チップ自体に物理的な乱数発生装置を持つものもあります。

3.3　ランダムアルゴリズム

ランダムアルゴリズム (randomized algorithm) とは、(擬似) 乱数を活用して、ランダムな振舞いを持たせたアルゴリズムを言います。これに対し、通常の決定的な動作を行うアルゴリズムは**決定的アルゴリズム** (deterministic algorithm) と呼ばれます。

ランダムアルゴリズムは、設計によっては決定的アルゴリズムよりすぐれた性能を持たせることができます。たとえば、1 億要素の配列があるとして、「その半分の 5 千万要素には値 a が入っているが、それがどことどこか所は分からない」場合と「まったく値 a が入っていない」場合とがあり、そのどちらであるかを判断する必要があるものとしましょう。

決定的アルゴリズムでは、どのような調べ方をしてもその「裏をかかれる」可能性があって半分は調べなければ確実な判断ができません。たとえば先頭から順番に値 a があるかどうか見ていくとすると、値 a が全部後半に詰まっているかもしれないので、最悪で 5 千万要素を見る必要があります。では後ろから順に見ればいいかというと、値 a が全部前半に詰まっているかもしれないので同じことです。1 つおきでも何でも同様です。

ここで、乱数を用いて 1 億の位置からランダムに 1 つ選び、その値が a かどうかを判断することを 1 万回繰り返したとしましょう。その結果 1 回も値 a に遭遇しなければ、「値 a はない」と判断してまず問題ありません。というのは、この判断が間違っている確率は $\frac{1}{2^{10000}}$ であり、それはこの計算をするコンピュータが故障する確率よりはるかに小さいからです。

このような、微小だが 0 でない「間違う」確率を持ったアルゴリズムを**モンテカルロアルゴリズム** (Monte Carlo algorithm) と言います。[10] これに対し、間違うことはないが運が悪い場合に性能が低下するアルゴリズムを**ラスベガスアルゴリズム** (Las Vegas algorithm) と言います。[11]

たとえば、クイックソートでどの要素をピボットとして用いるかを乱数で決めるようにすると、これはラスベガスアルゴリズムとなります。なぜなら、乱数がすべて「悪い要素 (その区間の最大や最小の要素)」を選び続ける確率は非常に小さいので、よほど運が悪くない限り高速に整列が行え、そして運が悪い場合は実行時間が長く掛かることになるものの、整列そのものはやはり正しく行えるからです。

3.4　モンテカルロ法

モンテカルロ法 (Monte Carlo method) とは、**シミュレーション** (simulation) などにおいて乱数を活用する手法を言います。たとえば、交通の流れを実際に観察する代わりに、乱数を用いてランダムに車を (プログラム内で) 発生させ、それらの車がどのように流れていくかを見ることで、さまざまな方法で交通信号を制御した場合の状況を簡単に試すことができ、それに基づいてよい制御方式を出すことができます。

また、モンテカルロ法は数値積分にも使うことができます (図 5)。具体的には、積分する範囲の関数値の最大より大きい値を選んで長方形の領域を考え、その範囲内に乱数で多数の点を打ち、関数値より下にある点の比率を求めます。積

[9] Ruby の `rand` の乱数アルゴリズムは最近の処理系 (バージョン 1.8 以降) では MT が使われています。
[10] モンテカルロはヨーロッパにあるカジノで有名な都市の名前です。
[11] ラスベガスは米国にあるカジノで有名な都市の名前です。

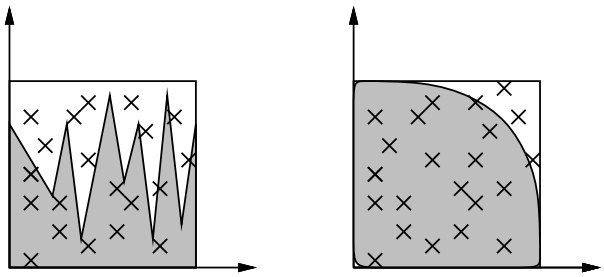

図 5: モンテカルロ法による数値積分

分とは「その関数の下側の面積を求める」ことですから、長方形の面積にその比率を掛けたものが積分値 (の近似値) として使えます。

そんな面倒なことをするよりシンプソンのアルゴリズムでよいのでは? しかし、シンプソンのアルゴリズムなどは、対象とする関数が連続かつ微分可能 (なめらか) でないと使えません。そのような性質が期待できないような分野では、モンテカルロ法が有力な手法の 1 つとなるのです。

たとえば半径 1 の四分の一円の面積を求めて (それを 4 倍することで) π の近似値を計算してみましょう: [12]

```
def pirandom(n)
  count = 0
  n.times do
    x = rand(); y = rand()
    if x**2 + y**2 < 1.0 then count = count + 1 end
  end
  return 4.0 * count / n
end
```

実行させてみると次のとおり:

```
irb> pirandom 10000
=> 3.13
irb> pirandom 100000
=> 3.14444
irb> pirandom 1000000
=> 3.141604
```

有効数字 3〜4 桁では使えない、と思いますか? 実際には、3〜4 桁の有効数字が得られれば十分な場合は結構あります。たとえば、来年の GDP の成長率が 5.11 だろうと 5.12 だろうと、3 桁目はさして重要ではないでしょう?

演習 6 モンテカルロ法で数値積分を行うときの、精度 (有効桁数) と試行の数との関係について考察せよ。円周率の例題を活用してもよいが、できれば別の関数を積分するプログラムを作って検討することが望ましい。

演習 7 乱数で点を打つのでなく、均一の細かさで格子点上に点を打って調べることが考えられる。この方法とモンテカルロ法の善し悪しについて考察せよ。何らかのテストプログラムを作って動かすこと。

3.5 乱数とゲーム

最後にお楽しみの話題としてゲームに言及しておきましょう。ゲームの中には、将棋や囲碁のように (先手後手を決める以外は) ランダム性を使わないものもありますが、多くのゲームでは (サイコロやカードのシャッフルなどを通じて) ランダム性を採り入れています。これは、ランダム性を採り入れることで、ゲームの「場面」が毎回違ったものになり新鮮

[12] もちろん円周は十分連続かつ微分可能ですが、それは置いておいて。

さが保たれ、また複数プレーヤで行う場合に「上手下手」以外の要因が入って下手な人にも勝つチャンスが生まれ、勝負の行方に興味が持てるようになるからです。

簡単なゲームとして「数当て」を作ってみます。そのルールは次のとおり:

- プログラムは内部で 4 桁の数を「思い浮かべ」る (4 桁の中に重複はない。また 0 もない)。
- プレーヤはその 4 桁の数を「当てる」ことをめざして、自分も 4 桁の数を入力する。
- プログラムは 2 つの 4 桁の数を照合して、「同じ位置に同じ数がある (これをヒットと呼ぶ)」個数と、「同じ数があるがただし違う位置にある (これをブローと呼ぶ) 個数とを数えて知らせる。
- プレーヤはその情報を見て再度チャレンジする。
- 10 回以内のチャレンジで当たればプレーヤの勝ち、さもなければプレーヤの負けとする。

Ruby プログラムを次に示します:

```
def kazuate
  a = ""; b = "123456789"; count = 0
  4.times do i = rand(b.length); a += b[i..i]; b[i] = "" end
  while true do
    print("your guess? ")
    s = gets; hit = 0; blow = 0
    4.times do |i|
      4.times do |j|
        if s[i] == a[j] then
          if i == j then hit +=  1 else blow += 1 end
        end
      end
    end
    if hit == 4 then puts "you win!"; return end
    count += 1
    if count > 9 then puts "you lose! answer = #{a}."; return end
    puts "hit = #{hit}, blow = #{blow}."
  end
end
```

ゲームの「やりとり」を行うために、キーボードから 1 行入力するメソッド gets を使っています。また、puts は改行してしまうので、プロンプトを出力するのに print を使いました。あと、文字列は配列と同様に添字を指定することで「何文字目」が取り出せたり、添字範囲を指定することで部分文字列が取り出せることも利用しています。

4 桁のランダムな数を作る方法がちょっと分かりづらいかもしれませんが、「まず a に空文字列、b に"123456789"を入れ、b の中からランダムに 1 文字選んでそれを a に連結し、b のその文字は空文字列に置き換える」ことを 4 回やっています。

本体ループは、プレーヤが入力した文字列とすべての位置の組合せで照合し、ヒットとブローの数を数えるわけです。

演習 8 上の例題プログラムを打ち込んで遊んでみよ。納得したら、乱数を使ったゲームを何か作ってみよ。上の例題の改良 (改造) 版でもよい。

A　本日の課題 7A

「演習 1」〜「演習 4」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、ペアの学籍番号 (または「個人作業」)、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．以下のアンケートの回答。

Q1．クラスの概念やその機能について納得しましたか。

Q2．乱数の有用性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 8」(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「12/2 の」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、ペアの学籍番号 (または「個人作業」)、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．乱数が使いこなせるようになりましたか。

Q3．課題に対する感想と今後の要望をお書きください。