プログラミングプロジェクト＃5 – ライブラリの内容と動画

久野　靖 (電気通信大学)

2021.9.6

1　はじめに

今回の主な内容は次の通りです。

- 2 次元配列と多次元配列
- 画像ライブラリの種明かしとより複雑な図形
- 動画の生成

手続きが使えるとプログラムを見通しよく書けるようになり、複雑な絵でも描けるようになると思います。ぜひマスターしてください。

2　前回演習問題の解説

2.1　演習 2 — 配列の演習

演習 2 は配列の演習です。最初は平均ですが、合計を求める変数 sum の初期値を「0.0」としているのに注意。これは、整数ばかりの配列を渡したときでも、合計が小数点つきになって個数で割算するのに小数点の計算が行われるためです。

```
def arrayavg(a)
  sum = 0.0
  a.each do |x|
    sum = sum + x
  end
  return sum / a.length
end
```

each は配列の各要素を順に取り出して来るメソッドでした。

次は最大。このような形で配列を使う場合は、ふつうは「とりあえず max に最初の値を入れておき、より大きい値が出てきたら入れ換える」方法になります。

```
def arraymax(a)
  max = a[0]
  a.each do |x|
    if x > max then max = x end
  end
  return max
end
```

次は最大の値が何番目に出てくるかなので、普通の計数ループにします。また、「何番目か」も変数に記録し、最大を更新した時にに同時に更新します。

```
def arraymaxno(a)
  max = a[0]
  pos = 0
  a.each_index do |i|
    if a[i] > max then max = a[i]; pos = i end
  end
  return pos
end
```

配列の各添字を列挙するには a.length.times を使えばよいのですが、ここに示したように配列のメソッド a.each_index を使うこともできます。

最大を 1 箇所だけ記録するのは変数でもできましたが、最大が複数あった時にその位置を全部打ち出すのには、(1) まず最大を求め、(2) その最大と等しいものがあったら位置を打ち出す、という形で 2 回ループを使う必要があります。

```
def arraymaxno2(a)
  max = a[0]
  a.each_index do |i|
    if a[i] > max then max = a[i] end
  end
  a.each_index do |i|
    if a[i] == max then puts(i) end
  end
end
```

平均より小さい値を打ち出すのもこれと同様です。

```
def arrayavgsmall(a)
  sum = 0.0
  a.each do |x| sum = sum + x end
  avg = sum / a.length
  a.each do |x|
    if x < avg then puts(x) end
  end
end
```

2.2 演習 3 — 複数のボールの演習

演習 3b と 3c のコードとできた絵 (図 1) を示しましょう。まず 3b ですが、これは最初の配列の初期化のところで、5 番目として中央の横線を増やしているだけです。

```
def fourballsb
  x = [40, 160, 160, 40, 40]; y = [40, 40, 160, 160, 100]
  dx = [20, 0, -20, 0, 20]; dy = [0, 20, 0, -20, 0]
  x.length.times do |i|
    6.times do
      fillcircle(x[i], y[i], 10, 0, 0, 255)
      x[i] = x[i] + dx[i]; y[i] = y[i] + dy[i]
    end
  end
  writeimage('pict5.ppm')
end
```

つぎに 3c ですが、これはそれぞれの線について「ボール何個分」を記録する配列 count を用意し、内側のループの回数として count[i] を指定しています。

```
def fourballsc
  x = [40, 160, 160, 40, 80]; y = [40, 40, 160, 160, 100]
  dx = [20, 0, -20, 0, 20]; dy = [0, 20, 0, -20, 0]
  count = [6, 6, 6, 6, 3]
  x.length.times do |i|
    count[i].times do
      fillcircle(x[i], y[i], 10, 0, 0, 255)
      x[i] = x[i] + dx[i]; y[i] = y[i] + dy[i]
    end
  end
  writeimage('pict6.ppm')
end
```

図 1: ボールの練習問題

3　2 次元配列と多次元配列

これまで、配列とは値が一直線に (1 次元的に) 並んだもの、というふうに説明してきました。それはそれでよいのですが、世の中には碁盤の目のように「縦横に (2 次元的に)」並んだものも存在します。そのようなものはどうしましょうか。

実は答えは簡単で、「配列のそれぞれの要素が配列」ならそれでよいのです。次の例を見てください。

```
irb> require 'pp'
=> true
irb> a = Array.new(5) do [0,0,0,0,0] end
=> [[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]]
irb> pp a
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
=> [[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]]
irb> a[3][2] = a[0][4] = 9
=> 9
```

```
irb(main):006:0> pp a
[[0, 0, 0, 0, 9],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 9, 0, 0],
 [0, 0, 0, 0, 0]]
=> [[0,0,0,0,9], [0,0,0,0,0], [0,0,0,0,0], [0,0,9,0,0], [0,0,0,0,0]]
```

「require 'pp'」というのは pp というメソッドを使えるようにする指示です。pp は複雑なデータ構造を見やすく表示してくれる機能を持っています。次に、長さ 5 の配列を作って変数 a に入れますが、それぞれの要素はまた「[0,0,0,0,0]」つまり 0 が 5 つならんだ配列です。pp で表示させると、このような「配列の配列」は縦横に並んだ形で表示されます。次に、配列 a の「3 行目、2 列目」と「0 行目、4 列目」に 9 を入れ、再度表示しています。

ここまででは作成時に全部の値が同じでしたが、ブロックにパラメタを指定することで「何番目の要素か」の値を取り出すことができるのでした (最初が 0 番目になるのに注意)。これを使って足し算の表のようなものを作ってみます。

```
irb> b = Array.new(5) do |i| Array.new(5) do |j| i+j end end
=> [[0,1,2,3,4], [1,2,3,4,5], [2,3,4,5,6], [3,4,5,6,7], [4,5,6,7,8]]
irb> pp b
[[0, 1, 2, 3, 4],
 [1, 2, 3, 4, 5],
 [2, 3, 4, 5, 6],
 [3, 4, 5, 6, 7],
 [4, 5, 6, 7, 8]]
=> [[0,1,2,3,4], [1,2,3,4,5], [2,3,4,5,6], [3,4,5,6,7], [4,5,6,7,8]]
```

このようにして 2 次元配列が作れますが、もし 2 次元配列のそれぞれの要素がまた配列だったら? そのときは「3 次元配列」になるわけです。3 次元の場合は a[4][2][1]」のように要素を取り出すのに添字が 3 つ必要になります。さらに大きな次元にものも「配列の配列の…の配列」として作ることができます。

演習 0 上の例を実際に実行してみよ。納得したら、図のような 5 × 5 の整数配列を作ってみよ。

(a)

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

(b)

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

(c)

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

(d)

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

(e)

9	8	7	6	5
8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	4	3	2	1

4 描画ライブラリの種明かし

それではここで、これまでお世話になってきた、画像を書き出す API のライブラリをきちんと説明しましょう。なお、ここで説明するライブラリはこれまでのものの強化版になっています。

まず、$で始まる変数名は**グローバル変数**を表し、複数のメソッドをまたがって共通に使える変数を意味します。ここでは$img は画像データとしてあちこちで書き込むのでグローバル変数にしていて、その初期化の内容はすべてのます目 (ピクセル) を [255,255,255] つまり RGB 最大の「真っ白」に初期化することとなっています。また、今回はファイルに連番を」つけるための変数$cnt もグローバル変数で、初期値は 1000 となっています。ただし。これまで通りファイル名を指定した場合はこの機能は使われません。

```
$img = Array.new(200) do Array.new(300) do [255,255,255] end end
$cnt = 1000
def pset(x, y, r, g, b)
  if 0 <= x && x < 300 && 0 <= y && y < 200
    $img[y][x][0] = r; $img[y][x][1] = g; $img[y][x][2] = b
  end
end
def writeimage(name = "img#{$cnt}.ppm")
  open(name, "wb") do |f|
    f.puts("P6"); f.puts("300 200"); f.puts("255")
    $img.each do|a| a.each do |p| f.write(p.pack("ccc")) end end
  end
  $cnt += 1
end
def clearimage
  $img.each do |a| a.each do |p| p[0] = p[1] = p[2] = 255 end end
end
```

pset では、指定された XY 座標が画像の範囲内 (0〜299、0〜199) のときだけ値の格納に進みます。値の格納は単に RGB 値を$img に入っている 3 次元配列に格納するだけです。writeimage は、ファイルへの書き出しがあるので少し難しいです。「open("ファイル名", "wb") do |f| ... end」はファイルに書き出すためのブロックで、このブロック中でパラメタ f に対して f.puts や f.write を使うことでファイルへの書き出しが行えます。f.puts では、PPM 形式ファイルのヘッダとして、画像の種別 (カラーのバイナリ形式) を表す文字列「P6」、画像の幅と高さ、そして RGB の最大値を出力しています。そのあと、$img のすべてのピクセルについて、メソッド pack を使って「3 バイトのバイナリデータ」に変換して f.write でそのまま出力します。その次にあるメソッド clearimage は、すべてのピクセルを「白」にリセットするもので、後で使います。

```
def fillrect(x0, y0, w, h, r, g, b)
  (y0-h/2).to_i.step(y0+h/2) do |y|
    (x0-w/2).to_i.step(x0+w/2) do |x| pset(x, y, r, g, b) end
  end
end
def fillcircle(x0, y0, rad, r, g, b)
  (y0-rad).to_i.step(y0+rad) do |y|
    (x0-rad).to_i.step(x0+rad) do |x|
      if (x-x0)**2 + (y-y0)**2 <= rad**2 then pset(x, y, r, g, b) end
    end
  end
end
```

fillrect と fillcircle はこれまで沢山使ってきました。どちらも、長方形や円の範囲内に相当する Y 座標の最小値から最大値まで、それぞれについて X 座標の最小値から最大値まで、y や x の値を変化させながら、長方形はその範囲を全部塗ります。円については、(x, y) が円の中に入っているという条件「$(x - x_0)^2 + (y - y_0)^2 \leq r^2$」を満たすときだけ、色を塗ります。

次は、新しい形の図形として、filltriangle を追加しました。これはかなり複雑で、何段にもわたって下請けが必要になります。まず原理を満てみましょう。平面は直線によって 2 つの領域に分けられますね。ですから、3 本の直線があれば、それによって分けられた中に三角形ができるわけです (図 2)。

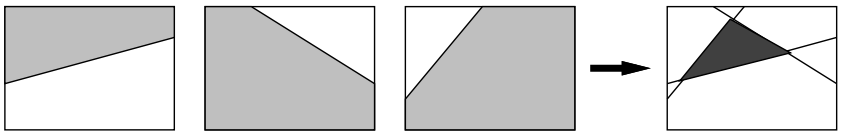


図 2: 三角形の内部の領域の判定方法

実は、線分 $(x_1, y_1) - (x_2, y_2)$ を与えたときに、任意の点 (x, y) が (x_1, y_1) から (x_2, y_2) を見たときの「左側か右側か」は外積という計算の符号の正負で判定できます。外積の計算は oprod というメソッドで行うように作りました。

なお、領域は三角形でなく、任意の頂点数であっても、凸な (へこみのない) 多角形であれば同様です。そこで、凸多角形の頂点の列と点を与えて、その点が凸多角形の中かどうかを判定するメソッド isinside を作りました。isinside はすべての返の線分について、上述の外積を計算し、負であれば右側にある (凸多角形の中にない) ので「いいえ」を返し、いずれでも負でなければ辺上または内部にある点なので「はい」を返します。

そこで fillconvex では、X 座標と Y 座標の配列を「左回りで」与えて (最初と最後は同じ頂点を指定)、Y 座標と X 座標の最小値と最大値を求め、この範囲内のすべての点について「凸多角形の内部であれば塗る」を実行しています。最後に filltriangle ですが、「頂点を左回り順で指定する」とか「最初と最後の頂点は同じものを指定する」とかが面倒なので、左回りと右回りのどちらでも大丈夫なように両方の順序で頂点を (最初と最後を重複させて) 指定して fillconvex を呼んでいます。

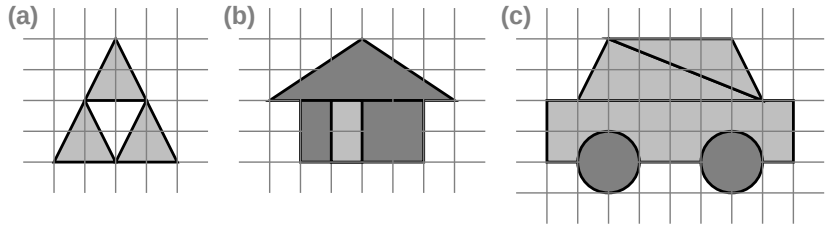
```
def filltriangle(x0, y0, x1, y1, x2, y2, r, g, b)
  fillconvex([x0, x1, x2, x0], [y0, y1, y2, y0], r, g, b)
  fillconvex([x0, x2, x1, x0], [y0, y2, y1, y0], r, g, b)
end
def fillconvex(ax, ay, r, g, b)
  xmax = ax.max.to_i; xmin = ax.min.to_i
  ymax = ay.max.to_i; ymin = ay.min.to_i
  ymin.step(ymax) do |j|
    xmin.step(xmax) do |i|
     if isinside(i, j, ax, ay) then pset(i, j, r, g, b) end
    end
  end
end
def isinside(x, y, ax, ay)
  (ax.length-1).times do |i|
    if oprod(ax[i+1]-ax[i],ay[i+1]-ay[i],x-ax[i],y-ay[i])<0
      return false
    end
  end
  return true
end
def oprod(a, b, c, d)
  return a*d - b*c;
end
```

もう 1 つ、fillline は $(x_0, y_0) - (x_1, y_1)$ に幅 w の線分を描きます。長方形とちがって斜めでもよいのが要点です。これは、線分を細長い頂点と考えて 4 隅の座標を計算し、fillconvex を呼ぶだけです。

```
def fillline(x0, y0, x1, y1, w, r, g, b)
  dx = y1-y0; dy = x0-x1; n = 0.5*w / Math.sqrt(dx**2 + dy**2)
  dx = dx * n; dy = dy * n
  fillconvex([x0-dx, x0+dx, x1+dx, x1-dx, x0-dx],
             [y0-dy, y0+dy, y1+dy, y1-dy, y0-dy], r, g, b)
end
```

では、新しい部品のうちでは一番よく使うと思われる「三角形」の練習を載せておきます。

演習 1 図に示すような、三角形を含んだ図形を描くメソッドを作成しなさい。



5 より進んだ絵

では三角形を活用して「海を行く船」の改良版を作ってみます。三角形以外にも、いくつか新しい技を取り入れました。できた絵は図 3 のようなものです。

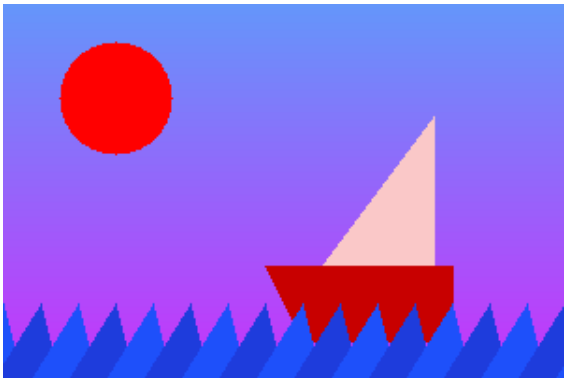


図 3: 海を行く船 2

ではプログラムの説明です。構造化グラフィクスで、個々の部品だけ改良したので、メイン部分は名前以外変わっていません。

```
def shiponthesea2
  sky; ship; wave
  writeimage('pict4.ppm')
end
```

船は三角形が使えるようになったので、かっこよくヨットにしました。

```
def ship
  filltriangle(170, 140, 230, 140, 230, 60, 250, 200, 200)
  fillrect(200, 160, 80, 40, 200, 0, 0)
  filltriangle(160, 140, 140, 140, 160, 180, 200, 0, 0)
end
```

波は三角波にして、一つおきにすこし色を変えました。

```
def wave
  17.times do |i|
    if i % 2 == 0
      filltriangle(i*20-25, 200, i*20+10, 200, i*20, 160, 30, 80, 250)
    else
      filltriangle(i*20-25, 200, i*20+10, 200, i*20, 160, 30, 60, 220)
    end
  end
end
```

空が一番難しくて、グラデーション (だんだん色が変わる) をつけています。原理は簡単で、空を 50 の細長い帯にわけ、それぞれすこし違った色で塗るだけです。分からなければ、i が 0 のとき、1 のとき、2 のとき... にどうなるか、確認してみなさい。太陽は前と変わっていません。

```
def sky
  50.times do |i|
    fillrect(150, 4*i, 300, 4, 100+2*i, 150-2*i, 250)
  end
  fillcircle(60, 50, 30, 255, 0, 0)
end
```

それでは。以下に新しいライブラリとプログラムを一緒にしたものを掲載しておきます。動画関係の機能の使い方は後で出て来ます。

```
$img = Array.new(200) do Array.new(300) do [255,255,255] end end
$cnt = 1000
def pset(x, y, r, g, b)
  if 0 <= x && x < 300 && 0 <= y && y < 200
    $img[y][x][0] = r; $img[y][x][1] = g; $img[y][x][2] = b
  end
end
def writeimage(name = "img#{$cnt}.ppm")
  open(name, "wb") do |f|
    f.puts("P6"); f.puts("300 200"); f.puts("255")
    $img.each do|a| a.each do |p| f.write(p.pack("ccc")) end end
  end
  $cnt += 1
end
def clearimage
  $img.each do |a| a.each do |p| p[0] = p[1] = p[2] = 255 end end
end
def fillrect(x0, y0, w, h, r, g, b)
  (y0-h/2).to_i.step(y0+h/2) do |y|
    (x0-w/2).to_i.step(x0+w/2) do |x| pset(x, y, r, g, b) end
  end
end
def fillcircle(x0, y0, rad, r, g, b)
  (y0-rad).to_i.step(y0+rad) do |y|
    (x0-rad).to_i.step(x0+rad) do |x|
```

```
      if (x-x0)**2 + (y-y0)**2 <= rad**2 then pset(x, y, r, g, b) end
    end
  end
end
def filltriangle(x0, y0, x1, y1, x2, y2, r, g, b)
  fillconvex([x0, x1, x2, x0], [y0, y1, y2, y0], r, g, b)
  fillconvex([x0, x2, x1, x0], [y0, y2, y1, y0], r, g, b)
end
def fillconvex(ax, ay, r, g, b)
  xmax = ax.max.to_i; xmin = ax.min.to_i
  ymax = ay.max.to_i; ymin = ay.min.to_i
  ymin.step(ymax) do |j|
    xmin.step(xmax) do |i|
     if isinside(i, j, ax, ay) then pset(i, j, r, g, b) end
    end
  end
end
def isinside(x, y, ax, ay)
  (ax.length-1).times do |i|
    if oprod(ax[i+1]-ax[i],ay[i+1]-ay[i],x-ax[i],y-ay[i])<0
      return false
    end
  end
  return true
end
def oprod(a, b, c, d)
  return a*d - b*c;
end
def fillline(x0, y0, x1, y1, w, r, g, b)
  dx = y1-y0; dy = x0-x1; n = 0.5*w / Math.sqrt(dx**2 + dy**2)
  dx = dx * n; dy = dy * n
  fillconvex([x0-dx, x0+dx, x1+dx, x1-dx, x0-dx],
             [y0-dy, y0+dy, y1+dy, y1-dy, y0-dy], r, g, b)
end

def sky
  50.times do |i|
    fillrect(150, 4*i, 300, 4, 100+2*i, 150-2*i, 250)
  end
  fillcircle(60, 50, 30, 255, 0, 0)
end
def ship
  filltriangle(170, 140, 230, 140, 230, 60, 250, 200, 200)
  fillrect(200, 160, 80, 40, 200, 0, 0)
  filltriangle(160, 140, 140, 140, 160, 180, 200, 0, 0)
end
def wave
```

```
  17.times do |i|
    if i % 2 == 0
      filltriangle(i*20-25, 200, i*20+10, 200, i*20, 160, 30, 80, 250)
    else
      filltriangle(i*20-25, 200, i*20+10, 200, i*20, 160, 30, 60, 220)
    end
  end
end
def shiponthesea2
  sky; ship; wave
  writeimage('pict4.ppm')
end
```

演習 2 改良された「海を行く船」をそのまま動かしてみなさい。うまく行ったら、次のような変更をしてみなさい。

a. 空の色を変えてみなさい。もっと夕焼けにする等。グラデーションはなくさないこと。
b. 船の帆にグラデーションをつけてみなさい。
c. 海を好きなように変えてみなさい。

6 動画の生成

ここまで静止画像はずいぶんやりましたが、最後に動画をやりましょう。動画の原理は「少しずつ違う画像を次々に表示すると動いて見える」だということはご存知ですよね。そしてコンピュータは、少しずつ違うものを沢山作るのを得意としています。実際にやってみましょう。

```
def movie5
  0.step(20) do |i|
    clearimage; fillcircle(20+i*8, 100, 20, 0, 255, 0); writeimage
  end
  0.step(20) do |i|
    clearimage; fillcircle(180, 100+i*5, 20-i, 255, 0, 0); writeimage
  end
end
```

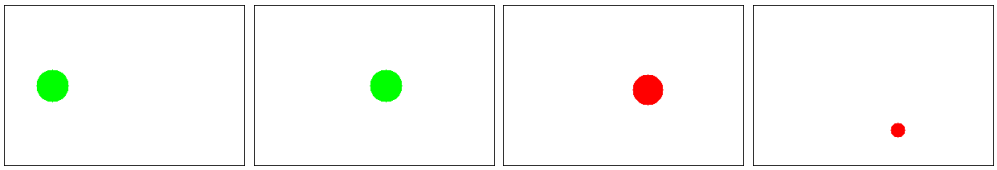


図 4: 動画のコマの例

このプログラムを動かすと、40 個くらいの画像ファイルができます。それを連続した動画として見るのには次のようにしてください。

```
animate img*.ppm
```

窓が開き、アニメーションが表示されます (図 4)。また、ブラウザなどで見られる動画ファイルにするのには次の方法によります。

```
convert -delay 5 img*.ppm anim.gif
```

最後に指定している出力ファイルの名前は自由にしてよいですが、形式は「.gif」を指定してください。このほか、gimp などの画像加工プログラムで動画に変換することもできます。

演習 3 上の例題をまずそのまま動かしなさい。動いたら、次のことをやってみなさい。

a. 円の色や動き方を変更してみる。円の色が徐々に変化するとなおよい。
b. 円ではなく別の図形を動かしてみる。図形の形や色が徐々に変化するとなおよい。
c. これまでに作った「絵の手続き」を使って、家や自動車などのものが動くようにしてみる。色が徐々に変化するとなおよい。

演習 4 これまでの例題に含まれていた絵の手続きや自作した絵の手続き (これから新たに作ってもよい) を用いて、簡単なストーリー性のある動画 (例: 太陽が沈むと家のあかりが灯るなど) を作ってみなさい。

本日の課題 5A

「演習 1」または「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 手続きを使った絵が作れるようになりましたか。

Q2. 動画の原理を理解しましたか。

Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題 5B

「演習 3」を題材としてプログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 動画を構想して作ることができるようになりましたか。

Q2. プログラミングのコツや難しいところは何だと思いますか。

Q3. ここまでの内容全体に対する感想やよかった点/悪かった点を教えてください。