J. Plasma Fusion Res. Vol.84, No.3 (2008) 155-162

講座 オープンソースソフトウェアを使った実践データ解析

4．Rubyを使って楽をしよう

大舘　暁
核融合科学研究所

（原稿受付：2008年1月30日）

4.1　はじめに

70年代からコンピュータを使ったデータ解析をやってこられた年配の研究者の方に「オブジェクト指向言語というのは何がうれしいのか．構造化プログラミングが流行した時はそのメリットがすぐわかったんだが.」という意味のことを聞かれたことがあります．オブジェクト指向の良さというのはなかなか見えにくい点があり，そのときは一言ではうまく説明できませんでした．本稿では，プログラマにとってプログラミングがもっとも楽しいといわれているオブジェクト指向言語 Ruby[1]を使って，いかに楽しく省力化して，日常の事務処理やデータ処理を楽に行うかを説明します．その過程で，オブジェクト指向の考え方を使うとプログラムの見通しが良くなることを，オブジェクト指向言語に触れたことのない皆様にわかっていただければと思います．Ruby の参考書としては，「たのしい Ruby」[1] と「プログラミング Ruby」[2]をあげておきます．前者は初心者むけの入門書で，後者は他のプログラミング言語に詳しい方に向いています．

4.2　Ruby 言語入門

Ruby は日本で開発されたスクリプト言語です．そのため日本語で書かれた参考書が多く出版され，日本語の解説サイトが多くあることは大きなメリットです．クリーンな文法を持ったオブジェクト指向の言語として知られ，日本では昔から大変人気のあるプログラミング言語です．海外でも，Ruby on Rails[2]という Web アプリケーションフレームワークが近年爆発的に支持されたのに伴い広く使われるようになってきました．Ruby のプログラムの例を以下に示しますが，本稿では，次のようにプログラムの断片とその実行結果を表します[3].

```
print "Hello\n" # => Hello
```

print "Hello\n"という文を実行するとHelloという出力が得られるという意味です．

本稿を試しながら読んでいただくためにはお手元の PC に Ruby をインストールするとよろしいかと思います．windows ユーザーは ActiveScriptRuby[4]というパッケージで一括インストールするのが楽です．Macintosh では標準で搭載されていますし，Linux/Unix ではそれぞれのパッケージシステムにしたがってインストールできると思います．Ruby には対話型に実行するための irb という環境が用意されています．以下のように起動すると簡単にテストを行うことができます．本稿でつかったスクリプトファイルとテストデータは特設サイトからダウンロードが可能です．

```
C:\Documents and Settings\ohdachi>irb
irb(main):001:0> print "hello\n"
hello
=> nil
```

Ruby のプログラムでは変数は宣言なしに使うことができます．

list 1
```
1  a = 1
2  x = 123.4
3  str = 'abc'
```

宣言は必要ありませんがそれぞれの変数に型はあります．

list 2
```
1  print a.class # => Fixnum
2  print x.class # => Float
3  print str.class # => String
```

それぞれ，整数型，浮動小数点型，文字列型を示します．変数にピリオドをつけて機能を呼び出すやり方が少々目新しいかもしれません．Rubyにおいてはすべての変数，数値

1　http://www.ruby-lang.org/
2　http://www.Rubyonrails.org/
3　＃という字を使って結果を示すのは，多くのスクリプト言語で＃から行末まではコメントを示すので，＃より後の部分が無視され，実行結果をプログラム中に表示してあってもプログラムをそのまま実行できるからです．
4　http://www.geocities.co.jp/SiliconValley-PaloAlto/9251/ruby/

Practical Data Analysis Using Open Source Software 4. Easy Programming with Ruby
OHDACHI Satoshi
author's e-mail: ohdachi@nifs.ac.jp

などはオブジェクトと呼ばれ，それぞれ特定の「クラス」に属します．それぞれのクラスには「メソッド」と呼ばれる専用の関数があり，オブジェクトにピリオドをつけて引き続いて書くことで呼び出すことができます[5]．".class"というメソッドはすべてのオブジェクトが共通に持っているもので，自分が属するクラスを示すメソッドです．整数型のオブジェクトは to_s というメソッドを使えば文字列にすることができます．逆に文字列を浮動小数点型に変換して演算するには以下のようにします[6]．

list 3
```
print 1.to_s              # => "1"
print "123.45".to_f * 2 # => 246.9
```

Ruby は非常に高機能な組み込み配列（Array クラス）を持っています．

list 4
```
arr = [1,2,3]
arr.class     # => Array
print arr[0]  # => 1
arr[1] = 'abc'
print arr     # => [1, 'abc', 3]
```

このように Ruby の配列には整数や浮動少数点だけでなく，文字列や任意のクラスを格納することができ，配列自体さえ持つことができます．要素を置き換えることもできますし，配列のサイズ自体も可変です．

list 5
```
a = [1,2,3]
print a.push(4)  # => [1,2,3,4]
print a.pop      # => [1,2,3]
```

制御構造は通常のものは当然用意されています．下記の例を見れば意味は想像できると思います．

list 6
```
x = 1.0
if x > 3.0
   y = 2.0
else
   y = 0.0
end
print y

i = 0
while i < 10
   print i, "\n"
   i += 1
end
```

他の言語と大きく見かけが異なる文法として，Ruby では以下に示すような block つきの記法が広く使われています[7]．ブロックとはプログラムの断片を中括弧{ }か do〜end で囲んだものです[8]．ブロック冒頭の| |部には変数名が書かれ，ブロックの中へ情報を伝える役割を果たします．

list 7
```
[1, 2, 3, 4, 5].each do |i|
  print i * 2, ' '
end # => 2 4 6 8 10
```

これは配列の各要素に対して，do と end で囲まれたブロックを実行するということを意味しています．要素はブロック中では変数 i で参照できるわけです．ここでは各要素を 2 倍した値を出力しています．見慣れない記法ですが慣れると非常に可読性が高いことがわかります．Ruby には使いやすく充実したクラスライブラリ群があり，多くの用途ではライブラリを利用することで，短いプログラムで多様な機能を発揮することができます．この標準ライブラリではブロックを使った洗練された記法が広く使われています．次節では標準ライブラリを使ったプログラミングを試してみます．

4.3 標準ライブラリを使ったプログラミング

それでは Ruby の標準ライブラリを使ってファイルの読み書きを行います．以下のような内容を持ったファイル（'data1.txt'）を読みこんで，簡単な処理をするプログラム（'testfile.rb'）を作ってみます．

list 8
```
# shotno,    bt,     ne,   Te
    70001,   1.5,    2.0, 3.0
    70002,   1.0,    3.0, 2.3
    70003,   2.5,    3.0, 0.5
#   70003,0.425, 999.0, 2.3
    70004,0.425,    1.0, 2.3
    70005, 2.75,    0.5, 5.0
    70006, 2.83,    3.0, 0.1
```

プログラムを実行するには ruby testfile.rb と入力します．

5 そのため，小数を持つ数値を示すときに123. のように小数点以下を省略するとエラーになるので注意してください．
6 awk や perl と違って自動的な型変換は行われません．
7 以前はイテレータという呼び方がされていましたが，イテレータとしての使い方以外にも応用範囲が広いため呼び方が変わりました．そのため古い本，古いサイトではイテレータと書かれている場合もあります．
8 両者には結合度などの差がありますが，普通に使う上では同等のものと考えられます．筆者は一行で書くときは{}，複数行にわたるときは do〜end の構文を使っています．

list9(testfile.rb)

```
1  data = []                    # 2次元の表として格納するためのコンテナを定義します.
2                               # data = Array.new と同じ意味です.
3  File.open('data1.txt').each_line do |line|
4     if line[0,1] != '#'       # 行頭が#の場合にはコメントと判断する.
5       columns = line.split(',').collect{|c| c.to_f }
6                               # 各行を',' で分割して, 浮動小数点としてアレイに格納する.
7       data.push(columns)      # 配列の配列として保存する.
8     end
9  end.close
10 data.each do |col|
11    print col[0], ' ', col[2] * col[3] / col[1]**2.0, "\n"
12 end
```

3行目はファイルをオープンして各行ごとに処理するときのおまじないだと思ってください．前節で紹介したブロック付きの記法を使っています．ここでオープンしたファイルは8行目でブロックが終了したときにクローズします．collectというメソッドは，配列の各要素を評価した結果を含む配列を返す機能を持ちます．ここでは読み込んだ行を','で分割し，生成された文字の配列を浮動小数点に変換しているわけです．そして，各行の，2列目，3列目の積を1列目の値の二乗で割った値を出力しています．この程度の例ですと，表計算ソフトや，グラフ化ソフト等でも同等の処理は簡単にできます．しかし，一度配列に読み込んでしまえば，プログラム言語としての強力な機能を使うことで，いくらでも柔軟で複雑な処理を行いえるはおわかりになると思います．

Rubyには多次元配列は組み込まれていません[9]．そのため配列の配列として2次元配列を扱い，data[0][1]といった形で要素にアクセスします．次の例では配列に対する演算例を見てみます．

list10

```
1  #list9の9行目までと同じにファイルからよみこむ
2  bt = []
3  data.each { |col| bt.push( col[1]) }
4  print bt.sort.uniq.join(' ')
```

この例では1列目のみを集めた配列btをつくり，大きさで並べ替えて(sort)，重複するものをとりのぞいて(uniq)出力しています．Unix/Linuxでのシェルを使った環境をご存知の方なら，単機能のプログラムをパイプで組み合わせるやり方を連想されると思います．このやり方はメソッドチェーンと呼ばれ，Rubyのメソッドや式が値を常に返すことから成立しています．Arrayには非常に多くのメソッドが実装されていて，それらをうまく組み合わせることで短いコードで非常に高い機能を実現でき，コードが短いことで大変読みやすいと思います．

メソッドチェーンの例をもう少し見てみます．

list11

```
1  print data.sort_by{|a| a[2]}[0..2].join(" ")
```

データの各行を2列目の値で並べ替え，上位3行を出力しています．配列の並べ替えのメソッドは(sort_by)は，ブロックをとることができ，並べ替え時に比べるべき対象(この場合は2列目の値　a[2])をブロック内に記述して各項の比較を行うことができるため，さまざまな並べ替えを簡単に実現することができます．並べ替えられた配列をjoinメソッド（配列を引数で区切って文字列としてつなぐメソッド）でつないで出力しています．メソッドチェーンを使うことで途中に一時的な変数をはさむ必要がなくなり，処理の内容に集中できるのがRubyの醍醐味のひとつだと思います．

配列に似ていて少し違う概念にハッシュ(Hash)[10]があります．配列では添字に数値を指定しますが，ハッシュでは添字として任意のオブジェクトが利用できます．

list12

```
1  h = {}              # => Hash の初期化
2  h['Japan'] = 81     #
3  h['Germany'] = 49
4  h.update( {'China' => 86, 'France' => 33} )
5                      # 一度に複数の追加もできます.
6  print h['Japan']    # => 81
```

9 多次元の数値配列を高速に取り扱うにはNArray（http://narray.rubyforge.org/）があります．次章で紹介される電脳Ruby（http://ruby.gfd-dennou.org/）などで利用されています．

10 連想配列と呼ばれることもあります．

それでは先ほどの表の検索をするためにハッシュを使ってみましょう．ここでは添え字には1行目のデータ（shotno）を使います．1行目（shotno）が70003の2行目（bt）の値が出力されるていることがわかると思います．

list13
```
1  bt = {}
2  data.each { |col| bt[ col[0].to_i ] = col[1] }
3  print bt[70003] # => 2.5
```

次にもう少し高機能な標準ライブラリと正規表現の使い方について紹介します．

数値データを取り扱うときに重要になるのが配列の扱いだとすれば，文字データを処理するときに役に立つのが正規表現関連のライブラリです．正規表現というのは文字列のパターンを一般化して特別な記号（メタ文字）を使った文字列で表現するやり方です．たとえば，pではじまりaで終わる文字列は/p.*a/とかかれます．ここで「.」は任意の一文字「*」はその0文字以上の繰り返しを意味し，この正規表現はpa，plasma，phenomenaなどという文字列をあらわします．正規表現をRubyで使う時には「//」で示します．

list14
```
1  print /p.*a/ =~ 'print'          # => false
2  print /p.*a/ =~ 'plasma'         # => true
3  print Regexp.last_match, "\n"    # => 'plasma'
4  print /p(.*)a/ =~ 'plasma'
5  print Regexp.last_match(1), "\n" # => 'lasm'
```

正規表現と文字列とは「=~」という演算子で比較され，正規表現が文字列を表現しているときはマッチしたといいます．マッチした文字列はあとからRegexp.last_matchなどをつかって取り出すことができます．正規表現中にまる括弧「( )」があれば，マッチした文字列から( )内の文字列のみを取り出すこともできます．

「.」，「*」以外に良く使われる正規表現に「[]」があります．これは文字クラスとよばれ，括弧内に列挙したいずれかの文字とマッチします．

list15
```
1  print /[abc]/ =~ 'a'       # => true
2  print /[abc]/ =~ 'A'       # => false
3  print /[a-z]*/ =~ 'cde'    # => true
4  print /-?(([0-9]+)|([0-9]*\.[0-9]*)([eE][-+]?[0-9]+)?)/ =~ '1.23E-45'
5                             # => true
```

文字クラス中に「-」があると，-の前後の文字の間の文字を意味します．[a-z]なら小文字すべてを意味します．上の3行目を参照してください．「+」は「*」と似ていますが，直前の正規表現の1文字以上の繰り返しを意味します．「?」は直前の正規表現0または1文字の繰り返しです．「|」は記号の前の正規表現と後の正規表現の論理和を意味します．この正規表現を使えば，上にあるように浮動小数点だけを示すようなパターンとマッチさせることもできます．このようなマッチングをプログラムで記述するのは大変な労力ですが，正規表現を使うことで簡単に，文字データから必要な情報を抜き出すことができます．

正規表現の応用例としてRubyに標準添付されているopen-uriライブラリを使ってwebページの情報を取り出して，そのファイルを検索してみます．

list16
```
1  require 'open-uri'
2
3  author = '○○○○'       # => 検索する文字列
4  File.open( 'result.html', 'w' ) do |f|
5     uri = URI.parse('http://scholar.google.co.jp/scholar?hl=ja&q=' +
6              author + '&btnG=Google+%E6%A4%9C%E7%B4%A2&lr=')
7     f.write uri.read
8  end
```

requireというのはRubyのライブラリをロードする命令で，open-uriの機能を呼び出すためのおまじないです．このプログラムは論文検索を行うことができるGoogle Scholar（http://scholar.google.co.jp/scholar）で任意の文字

列（○○○○）にたいして検索を行い，結果を result.html というファイルに書き込みます[11]．ここでは File.open にブロックを与える記法を使いましたので，6 行目でブロック終了するとともにファイルは自動的にクローズされます．

html ファイルでは外部のリンクは Link といった形であらわされますからこのLinkの部分を取り出せば，どのような論文にリンクができているかを取り出すことができます[12]．実行すると論文名が出力されるのがわかると思います[13]．

list17

```
1  File.open('result.html') do |f|
2    s = f.read
3    s.scan( /\<a href=[^<>]*\>([^<>]*)\<\/a\>/ ) do |link|
4      print link, "\n"
5    end
6  end
```

ここでは文字列を正規表現と連続させてマッチさせる scan というメソッドを使い，'result.html'中の外部リンクを抽出しています．

このような正規表現や open-uri といった高度な機能が組み込みまれていて簡単に呼び出すことができるのも Ruby の優れた点の一つです．文献検索システムのデータを自動で取得できると，論文中の文献リストの作成や，自分の出版リストの作成などが簡単に行えるようになります．

4.4 クラスを使ったプログラミング

それではより複雑な構造を持ったデータを扱ってみましょう．これまでは組み込みのクラス，たとえば配列を扱う Array クラスなどを利用してきたわけですが，今度は利用したいデータ構造にあわせて自前のクラスを定義することで，複雑なデータ構造に対応します．

ここでは，少しマニアックですが，核融合科学研究所の大型ヘリカル装置 LHD の平衡磁場の磁気面のデータファイルを対象にしてみます．

LHD の k 番目の磁気面の θ，ϕ（トロイダルアングル）でラベルされる位置の座標 (R, Z) は

$$R = \sum_{i=1}^{max-mode_k} R_{ki} \times \cos(\alpha_i) \quad (1)$$

$$Z = \sum_{i=1}^{max-mode_k} Z_{ki} \times \sin(\alpha_i) \quad (2)$$

$$(\alpha_i = \theta \times M_i - \phi \times N_i) \quad (3)$$

というように R_{ki}，Z_{ki} という展開係数を使って計算します．この R_{ki}，Z_{ki}，M_i，N_i と k 番目の磁気面における物理量はあらかじめ平衡コードによって計算されたものがファイルの形で提供されています．下記にファイルの例を示します．jr が磁気面のラベルで，1-2番目のラベルに対しての展開係数が示されています．ブロックの最後にこの磁気面に対応する物理量（規格化小半径，回転変換など）が書かれています．

list18

```
1   nrho=31  modnum=98
2   jr= 1                                                  #1 番目の磁気面のデータ
3   m= 0  n=  0  rmn= 3.60056E+00  zmn= 0.00000E+00
4   m= 0  n=-10  rmn= 3.62157E-02  zmn= 3.51149E-02
5  #↑     ↑       ↑               ↑
6  #Mi    Ni       Rmn             Zmn
7  ..
8  ..
9   m= 7  n=-60  rmn= 0.00000E+00  zmn= 0.00000E+00
10  rho=0.00000E+00  1/q=3.78164E-01  Vp= 6.33758E-01
11  b0= 2.98339E+00  et=-1.14997E-02  eh= 2.14757E-03
12  nrho=31  modnum=98
13  jr= 2
```

11 ここで，Google Scholar にこの文字列を送る意味はわかりにくいと思います．web ページと実際にどのようなデータのやり取りをしているかを調べた結果から決めたものです．調べる方法はいろいろあります．ブラウザに Firefox を使っている場合は add-on 'liveHTTPheaders'をインストールするのが簡単な方法のひとつです．

12 html ファイルは正規表現のみで情報を取り出すのが難しいので，実用的な処理を行う時には Hpricot（http://code.whytheluckystiff.net/hpricot/）などの専用のパースライブラリを使うのが便利です．

13 他の外部リンクも出力されてしまいますが．

```
14   m= 0  n=  0  rmn= 3.60056E+00  zmn= 0.00000E+00
15   m= 0  n=-10  rmn= 3.62157E-02  zmn= 3.51149E-02
16  ..
```

このスタイルのファイルを読み出すプログラム('readflx.rb') は少し長くなるのでサポートサイト上からダウンロードしてください．ファイル読み出しの核の部分は以下のようになります．

list19

```
1   def loadflux
2
3    #配列の初期化部は省略
4
5    for i in 0 .. @nrho-1
6     @file.gets
7     for j in 0 .. @modnum-1
8      temp = @file.gets
9      t = temp.split(/ *[a-z]+= */)
10     @mmd[i][j] = t[1].to_i ; @nmd[i][j] = t[2].to_i
11     @rmn[i][j] = t[3].to_f ; @zmn[i][j] = t[4].to_f
12    end
13
14       t = @file.gets.split(/ *[a-z0\/]+= */)
15       @rho[i] = t[1].to_f ; @iota[i] = t[2].to_f ; @vp[i] = t[3].to_f
16       t = @file.gets.split(/ *[a-z0\/]+= */)
17       @b0[i] = t[1].to_f ; @et[i] = t[2].to_f ; @eh[i] = t[3].to_f
18     end
19   end
```

splitは文字列を分割するときに正規表現も使うことが可能なので，ここでは各行を等号を目印に分割して，数値データを取り出しています．@ではじまる変数名がありますが，インスタンス変数と呼ばれるものです．これから磁気面データを扱うクラス「Flux」を定義していきますが，そのクラスから生成される実体である「インスタンス」が保持する変数で，インスタンスごとに別の値をとることができます．

それではクラス「Flux」を定義してみます．

list20

```
1  class Flux
2    attr_accessor :rho, :iota, :vp, :b0, :et, :eh, :nrho
3                                  # 外部から参照できるインスタンス変数の定義
4    def initialize(filename)     # 初期化ルーチン
5      @file = File.open(filename)
6      @nrho = 0
7      @modnum = 0
8      loadflux
9      @file.close
10   end
11   def mkrz(phi, theta, nrho)  # 実際の座標を計算する
12     rtemp = 0.0
14     ztemp = 0.0
14     for mode in 0 .. @modnum-1
15       angle = theta * @mmd[nrho][mode] - phi * @nmd[nrho][mode]
16       rtemp += @rmn[nrho][mode] * Math.cos(angle)
17       ztemp += ztemp + @zmn[nrho][mode] * Math.sin(angle)
18     end
```

```
19
20      return rtemp, ztemp
21    end
22  end
```

クラスではインスタンスを生成するときにinitializeというメソッドが自動的に呼ばれます．このときに読み込むべきファイル名(filename)を指定します．インスタンス変数を初期化した後に，さきほどのloadfluxルーチンを使って与えられたファイルから読み取った変数に内容を代入していきます．また，ここでmkrzというメソッドを定義していますが，n番目の磁気面(nrho)のトロイダル角phi，擬ポロイダル角thetaの座標r, zを定義どおりに計算します．このとき，フーリエ展開の係数@rmn，@zmnなどのインスタンス変数はインスタンス内外に保持されていますがクラスの外からはアクセスすることができないようにしています．外部で参照する必要のない情報を隠蔽して，不用意に変更されることなどを防ぐ意味があります．

ではこのクラスを使ってみましょう．

list21

```
1  require 'readflx.rb'
2  f360 = Flux.new('lhd-r360.flx')
3  r, z = f360.mkrz(0.0, 0.0, 0) # => 3.63685362881595 0.0
4                                #     Rubyのメソッドは複数の返り値をとることができます．
5  print r, z, "\n"
```

ここではLHDの標準的な配位のファイル('hd-r360.flx')を使用して最も内側の磁気面の座標を求めています．

次に複数の磁気面で計算した結果を比べてみます．

list22

```
1  require 'readflx.rb'
2  f360 = Flux.new('lhd-r360.flx')
3  f375 = Flux.new('lhd-r375.flx')
4
5  nrho = f360.nrho
6
7  (0 .. 360).each do |t|
8    theta = t.to_f * 2 * Math::PI / 180.0     # Math::PIは定数π
9    r1, z1 = f360.mkrz(0.0, theta, nrho - 1)
10   r2, z2 = f375.mkrz(0.0, theta, nrho - 1)
11   print "#{theta}, #{r1}, #{z1}, #{r2}, #{z2}\n"
12 end
```

二つの磁場配位の最外郭磁気面の座標が出力されました．各磁気面の展開係数はそれぞれのインスタンス(f360, f375)が保持するインスタンス変数という形で持っているため，複数のデータファイルを取り扱う場合でもコードがあまり複雑化しないことがわかります．これがクラスをつかった抽象化のメリットです．次に，ここまでの総まとめとして，ここで読み込んだ二つの磁気面を図面として出力してみます．

list23(draw-flux.rb)

```
1  require 'graph.rb'
2  require 'canvas.rb'
3  require 'readflx.rb'
4
5  f360 = Flux.new('lhd-r360.flx')
6  f375 = Flux.new('lhd-r375.flx')
7
8  nrho = f360.nrho
9
10 g = Graph.new([0], 0, 1, :xrange => [3.2, 4.2], :yrange => [-1.0, 1.0])
11 g.xaxis.title = 'R [m]'
12 g.yaxis.title = 'Z [m]'                                  # グラフの枠などを準備
```

```
13
14 (0 .. nrho-1).each do |irho|
15   data1 = []
16   data2 = []
17   (0 .. 360).each do |t|
18     theta = t.to_f * 2 * Math::PI / 180.0
19     data1.push( f360.mkrz(0.0, theta, irho - 1) )
20     data2.push( f375.mkrz(0.0, theta, irho - 1) )
21   end
22   g.add(data1, 0, 1, :gtype => Ogre::Line, :symtype => 0) # グラフに線を引く
23   g.add(data2, 0, 1, :gtype => Ogre::Line, :symtype => 1)
24 end
25
26 PSCanvas.new('flux.ps') do |p|              # postscript ファイルに出力
27   p.setpart([0.1, 0.47], [0.95, 0.82])
28   g.plot(p)
29 end
```

ここで使ったプロットライブラリ[14]については詳しくふれませんが，磁気面ごとに data1，data2 という配列に座標を格納し，22-23行目でグラフに線を追加してることがわかると思います．完成版のプログラム（testflux2.rb）を実行すれば**図 1** に示すようなグラフができます．

細かい作業をクラス内に閉じさせることで，そのクラスを使うプログラムは驚くほど簡潔になりました．クラスを作るところで労力を使っておけば，それを利用する時にはきれいにプログラムをまとめることができることがわかると思います．当然ながらプログラムの見通しがよくなれば，バグが入り込む余地がへり，プログラムを使って解決したい問題に集中しやすいでしょうし，このクラスを使って別の計算をするときも，少ない手間であたらしい処理が書けるはずです．

この磁気面データの例のように，科学技術の分野ではデータ構造が複雑でうまくクラスを作ると使い勝手が良くなるという事例は多いと思います．データをどのようなクラスに反映させるかはなかなか難しい問題で，はじめはうまく設計できないかもしれません．Ruby に標準添付されたライブラリ群は，コンピュータの多彩な機能をうまく整理してクラスに落とし込んでいると高い評価があります．身近にある Ruby の標準ライブラリを使いながら，使いやすいクラス設計について勉強していくというのが近道だと思います．

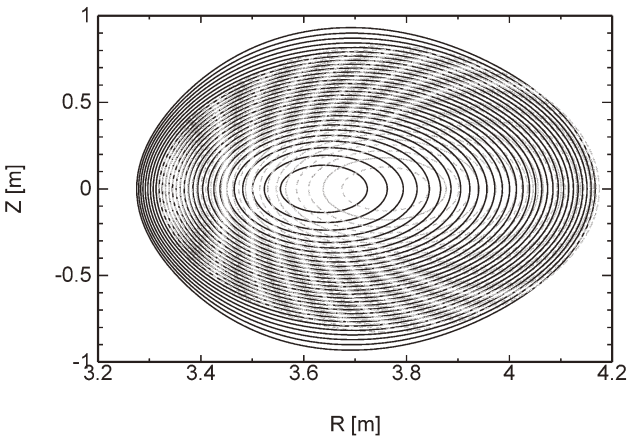

図 1　磁気面の出力例．磁気軸 3.6 m と 3.75 m の磁気面の比較．

4.5　おわりに

ruby のようなすばらしい言語を無償で公開されている，まつもとゆきひろさんをはじめとする ruby コミュニティーに感謝します．rubyist magazine（http://jp.rubyist.net/magazine/）の興味深い連載を参考にさせていただきました．この原稿を書く過程でも本文中のコード断片をテストするためのプログラムは Ruby を使って書きました．また個人的には Ruby を文献リストの作成，計測機器の制御，グラフの描画など研究上で楽しく活用しています．Ruby には魅力的な機能が多いので多彩な側面を紹介しようとして，詰め込みすぎだったとは思いますが，読者のみなさまに Ruby でのプログラミングの楽しさを伝えることができたとすればこれに勝る喜びはありません．Ruby 言語の詳細をお知りになりたい方は成書や，Ruby リファレンスマニュアル（http://www.sakalab.org/prog-ruby/ruby-man-ja-html-20051129/）などを参照ください．この原稿を書くチャンスをくださった編集委員のみなさま，なかでも原稿にコメントをくださった後藤基志さん，鈴木康浩さんに深く感謝いたします．

参 考 文 献

［1］高橋征義，後藤裕蔵（著）：たのしい Ruby—Ruby ではじめる気軽なプログラミング（ソフトバンククリエイティブ，2002）．
［2］D. Thomas, C. Fowler, A. Hunt：プログラミング　Ruby　第 2 版　言語篇，ライブラリ篇，（オーム社，2006）．

14　Ogre といい筆者の自作のライブラリです．特設サイトからダウンロードできます．