卒業論文

Ruby から Maple を呼び出す インターフェースライブラリ開発

関西学院大学　理工学部　情報科学科

3528 村瀬　愛理

2017 年 3 月

指導教員　　西谷　滋人　教授

目次

1　概要

西谷研究室での数値計算を用いた研究で多く使われるのが，Maple と Ruby である．Ruby ではできない計算を Maple にさせていたが，別々のソフトウェアを使うよりも Ruby のみで完結させるためにインターフェースライブラリを開発することにした．今回の研究では，Maple のコマンドライン実行される計算エンジン部に着目し，そこに働きかけて操作した．整数論関係から 7 つ，行列から 6 つの関数を選抜し実装した上で，関数に応じた wrapper を作り正しい出力型を取れるようにした上で，動的メソッドの実装後と実装前でどちらのプログラムがどう良いか比較した．そして，正しく計算できているか，または Ruby のみでの計算よりも数式処理能力が優れているかを検証し，成功した．今後は今回実装しなかった関数や，Maple の特性である綺麗なグラフを出力できるような関数を実装できれば良いと思う．

2 序論

Ruby は，まつもとゆきひろ氏によって開発されたオブジェクト指向スクリプト言語である．他にもテキスト処理に適した正規表現や高階関数，ガベージ・コレクションなどの特徴を持っている．フリーソースソフトウェアであるため，誰でも自由に使用することが可能である．

一方，数値計算分野においては Python が多用される．Python は Ruby と同じオブジェクト指向のスプリクト言語である．この 2 つは度々比較され，どちらが優れているのかを議論されてきた．2 つのスプリクト言語の決定的な違いは何を得意としているかである．Ruby は Web 分野を得意とするのに対し，Python は数値計算やビッグデータを得意としている．逆に Ruby は数値計算には弱く，Python は Web 分野には弱い．もちろんこの議論に答えはなく，本来は自分が何を目的としたプログラムを作るのかで使い分けるのが理想だろう．しかし，Ruby を使い慣れている人が数値計算をするためだけに Python を勉強し直すよりも，Ruby 上で数値計算ができる方が良いのは明らかである．

西谷研究室では数値計算を用いた研究を行っている．その研究で度々使われるのが数式処理ソフトウェアの 1 つである Maple である．Maple は，1980 年にカナダ・ウォータールー大学で生まれた数式処理技術をコアテクノロジーとして持つ科学・技術・工学・数学（STEM：Science, Technology, Engineering and Mathematics）に関する統合的計算環境である`{{cite(listings1)}}`．特徴として，たくさんの数学関数が用意されていること，大きな桁数の計算が可能であること，グラフの描画が簡単であり，かつ 3 次元のグラフの描画にも対応していることなどが挙げられる．数式を入力するだけで簡単に解を得ることができることから，多くの場で用いられている．

一方でソフト開発には Ruby を用いている．Ruby は数値計算関連の環境設備が遅れているため，Ruby のみで高等な関数，例えば，大きな素数を生成したり，最小公倍数を求めるなどの処理を行うのが難しい．また，扱える数値の桁数が計算内容によっては足りないということも考えられる．一方で，Ruby 以外の数式処理ソフトウェアなどを立ち上げて，別々に作業したり，慣れない別の言語を勉強し直したりするよりも Ruby のみでプログラミングする方が，開発速度の格段の向上が期待できる．そこで本研究では，Maple を Ruby 上で呼び出し，Maple に高等な関数や桁数の大きな数値を用いた計算をさせて，その結果を Ruby が取得するインターフェースライブラリの開発を目的とする．

3　手法

3.1　Maple との通信手法

Maple は一般的に，グラフや数式の綺麗な出力や，数式の入力を初心者が直感的におこなえるように Java で作られた GUI を使って実行する．それとは別に command line で実行される計算エンジン部が用意されている．そこで, 開発する Ruby ライブラリでは，このエンジンに直接働きかけて操作する．Ruby で外部コマンドを実行する gem library の systemu を使って，出力を得るようにしている．Ruby code で要求コードを受け取った場合，そのコードを tmp.mw に書き込む．それを Maple で実行し，結果をテキストファイルで受けとることで出力を得る．

3.2　Maple 関数の類型化

今回，数多く存在する
実装した．

関数名	振る舞い	入力型	出力型
nextprime	次の素数を求める	int	int
lcm	最小公倍数	int, int	int
gcd	最大公約数	int, int	int
rand	乱数生成	int	int
isprime	素数判定	int	boolean
ifactor	素因数分解	int	string
mod	剰余	int, int	int

図 1　実装した整数論に関する関数の役割と入出力

関数名	振る舞い	入力型	出力型
importmatrix	textファイルから 行列を読み込む	string, stiring	int
matrix	行列生成	int, int, int	array
matrixinverse	逆行列	array	string
determinant	行列式の解	array	float
trancepose	転置行列	array	string
eigenvectors	固有値, 固有ベクトル	array	string

図 2　実装した行列に関する関数の役割と入出力

3.3　出力の切り替え

Maple から受け取ったままの出力は，値の前にスペースがたくさん入っていることや，出力が String 型であることから，その数値を使って計算をするようにプログラミングしていた場合に支障をきたす．このため，関数ごとに正しい型で出力できるように wrapper を作る．例えば，int 型で出力が欲しいものは exec を exec_i から呼び出すことで対応する．このように boolean や float といった出力型に応じて，exec_b，exec_f のように関数を増やしていく．

4　実装

4.1　mapleruby の基本動作

入力された値の次の素

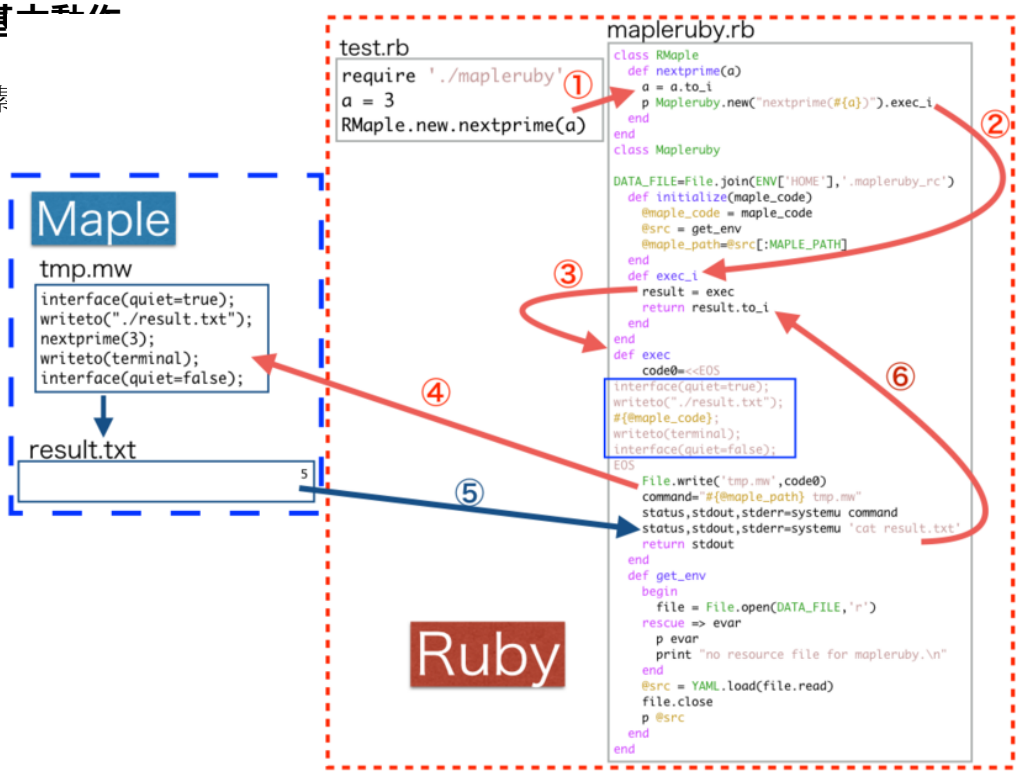

図 3　mapleruby の基本動作

1. mapleruby を require した上で使いたい関数を使う．RMaple.new.hogehoge の hogehoge に使いたい関数名を入れる．今回は nextprime で説明を進める．
2. RMaple クラス内の nextprime 関数が呼び出され，a に 3 が入る．この時 nextprime は入力された値が int 型になるように関数内で to_i してある．その後，Mapleruby クラスの exec_i 関数へ”nextprime(3)”が出力される．この出力された文字列がそのまま Maple での計算に使われる．
3. 出力された文字列をさらに exec 関数へ出力する．
4. 青四角内の内容を Maple へと出力する．この時`#{@maple_code};`となっている部分に先ほどの”nextprime(3)”が入る．青四角の内容が Maple に出力され実行されることで得られた答えが result.txt に出力されるようになっている．
5. result.txt に出力された内容を Ruby 側で受け取り，exec_i に再び返す．

6. 返された値を to_i することで int 型に直して解を出力する．

4.2 出力の切り替えの実装例

先ほどと同様に nextprime を例に挙げると exec_i は，exec で maple に式を送った後 maple から受け取った値を to_i し，int 型にしてから返すようになっている．もし使われた関数が素数判定を true/false で出力する isprime だった場合は，出力は boolean 型が好ましいため受け取った値を boolean 型にする exec_b を用いている．このように整数論に関する関数は，出力に応じて int 型で解を得たい場合は exec_i，float 型なら exec_f，string 型なら exec_s とすることで切り替えられるようになっている．

一方で，行列の場合は出力に切り替えについて例外が存在する．なぜなら，Maple の CUI 版は行列の表現が図 4 のように独自のもので，それが result.txt を通して Ruby に出力されるからだ．例えば，行列を生成する関数 matrix は以下の図 3-2 のように解を出力する．

```
> with(LinearAlgebra):
> matrix(3,3,[[1,4,7],[2,11,8],[3,6,0]]);
                                [1     4    7]
                                [             ]
                                [2    11    8]
                                [             ]
                                [3     6    0]
```

図 4　MapleCUI 版での行列の表示

この空白部分には半角スペースや改行が入っている上，余分な括弧が付いている．この関数を使う際に行列を生成して出力するだけなら問題ないが計算に数値を使う場合 Ruby の方で都合の良い出力型に変える必要がある．そのための wrapper を考える必要がある．

そこで，int 型の要素を持つ listlist 構造で出力されるのが最も良いと考えた．今回実装した行列の関数の多くは，listlist 構造のものを Maple の convert という listlist 構造から Maple 内で扱う行列の形に変換できる関数も一緒に Ruby から Maple に送るようにしているためである．行列生成した後理想の型で値を得るためには，いらない記号や空白を取り除き要素を int 型にす
う wrapper を作ってみ

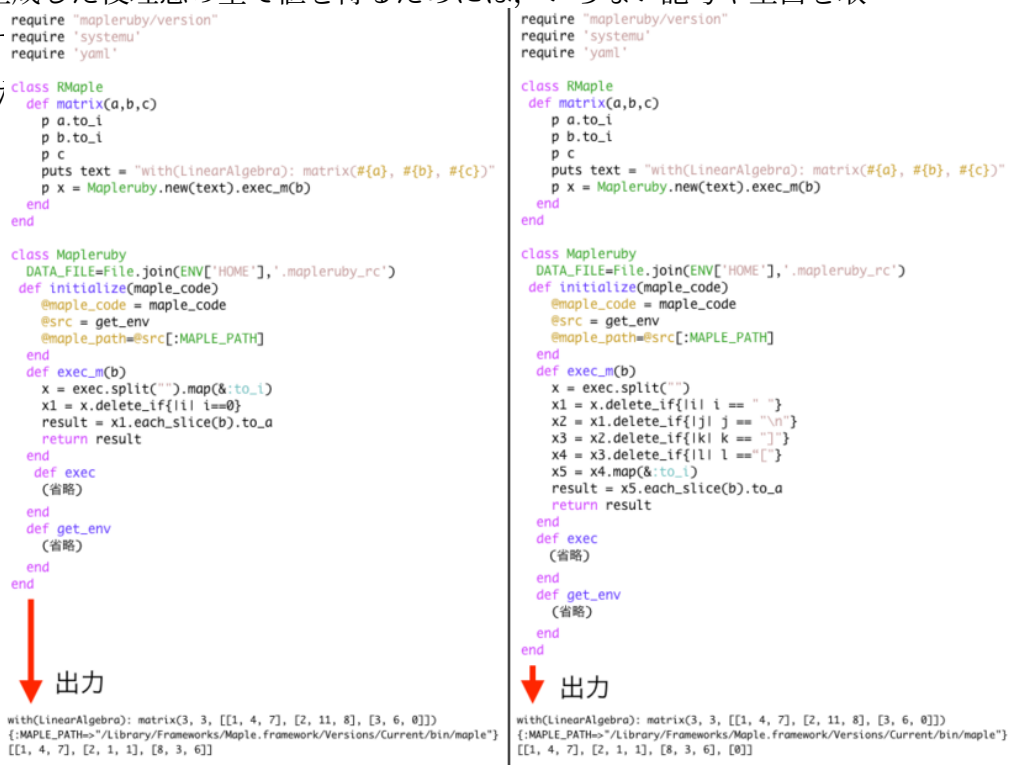

図 5　exec_m(b) を実装した際の失敗例

b には生成した行列の列の数が入る．実装する際に初めに考えたのが図 3-3 の左側のプログラムである．まず Ruby の split メソッドを使って 1 文字ずつに分け，分けた要素全てを int 型に変換する．空白は int 型に変えた際 0 になるため，0 になった空白部分を delete_if メソッドを用いて削除し，最後に each_slice メソッドを用いて 1 行目，2 行目... と分けてそれぞれ配列に入れ，出力したかった行列と同じような listlist 構造になるはずだった．しかし，listlist 構造への変換はできていたが途中 delete_if メソッドにより 0 を消してしまったため，行列の要素で 0 が含まれていた場合に行列の要素まで消えてしまった．しかも，1 文字ずつ分けているため 2 桁以上の桁数を持つ要素はばらばらになってしまった．

そのことを踏まえて，右側のプログラムを作成した．今度は 1 文字ずつ分けるところまでは先ほどと一緒で，空白を丸ごと消そうとするのではなく delete_if メソッドで” ”(空

白)，"\n"，”[”，”]”を順番に削除した後に to_i し，先ほどと同じように each_slice メソッドで listlist 構造になるようにした．こうしたことによって．要素に 0 が含まれていてもきちんと出力されるようように実装することで期待

```
require "mapleruby/version"
require 'systemu'
require 'yaml'

class RMaple
  def matrix(a,b,c)
    p a.to_i
    p b.to_i
    p c
    puts text = "with(LinearAlgebra): matrix(#{a}, #{b}, #{c})"
    p x = Mapleruby.new(text).exec_m(b)
  end
end

class Mapleruby
  DATA_FILE=File.join(ENV['HOME'],'.mapleruby_rc')
  def initialize(maple_code)
    @maple_code = maple_code
    @src = get_env
    @maple_path=@src[:MAPLE_PATH]
  end
  def exec_m(b)
    x = exec.gsub(/[^\d]/, " ")
    x1 = x.split(" ").map(&:to_i)
    result = x1.each_slice(b).to_a
    return result
  end
   def exec
    (省略)
  end

  def get_env
    (省略)
  end
end
```

出力結果

```
with(LinearAlgebra): matrix(3, 3, [[1, 4, 7], [2, 11, 8], [3, 6, 0]])
{:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/Versions/Current/bin/maple"}
[[1, 4, 7], [2, 11, 8], [3, 6, 0]]
```

図 6　exec_m(b) の完成形

完成形では，まず gsub メソッドで数字以外の記号を空白に置き換え，split メソッドで空白を指定することで空白を区切りとした配列にした後 int 型に直す．直した配列はその後，先ほどと同様に each_slice メソッドを用いて listlist 構造になるようにしている．

4.3　動的メソッドを用いての実装

一通り実装した後，次に動的メソッドを用いて実装することにより重複コードを減らすように試みたバージョン 2 を作成した．

初期バージョンでは，関数ごとに各引数を好ましい型に変換した後 Mapleruby クラスに遷移していた．バージョン 2 では，各数学関数は Maple での関数名と引数のみになり，新たに作った main_i 関数にそれらを送ることで初期バージョンと同様の動作を実装している．main_i 関数の第二引数が可変長引数になっているのは関数によって入力されている引数の個数が違うためである．例えば main_i 関数は出力が int 型である関数に対して使っており，図 7 のように実装した関数の重複部分や出力に応じて分類して，それぞれ関

```ruby
require "mapleruby/version"
require 'systemu'
require 'yaml'

class RMaple
  def lcm(a,b)
    a = a.to_i
    b = b.to_i
    p Mapleruby.new("lcm(#{a},#{b})").exec_i
  end
  def mod(a,b)
    a = a.to_i
    b = b.to_i
    p Mapleruby.new("modp(#{a},#{b})").exec_i
  end
end

class Mapleruby
  (省略)
end
```

```ruby
require "mapleruby/version"
require 'systemu'
require 'yaml'

class RMaple
  def lcm(a,b)
    main_i :lcm, a, b
  end
  def mod(a,b)
    main_i :modp, a, b
  end
  def main_i(name,*list_a)
    p name
    p list_a
    p Mapleruby.new("#{name}",list_a).exec_i
  end
end

class Mapleruby
  (省略)
end
```

図 7　左が初期バージョン，右がバージョン 2

数を追加している．

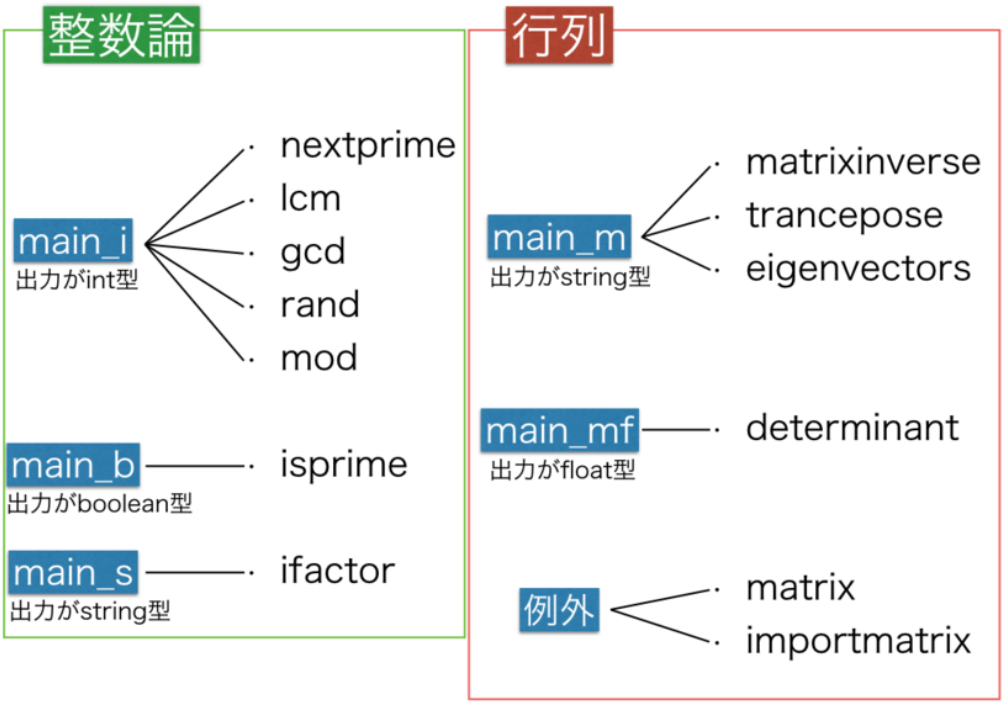

図 8　関数の分類

matrix は他に exec_m(b) を使う関数がないため，importmatrix は他と重複するコードがないため例外としている．

5 検証

5.1 RSA 暗号を用いた検証

整数論に関する関数は RSA 暗号の計算を用いて，Ruby のみで計算する場合と mapleruby で計算した場合でどのような差があるか検証する．検証のために以下のようなプログラムを用意した．

```
1  #rsa_org.rb
2  require 'prime'
3  include Math
4
5  def rsa(input)
6    c = input.to_i
7    print "平文>>>␣#{c}\n"
8
9    big_num = sqrt(c).to_i
10   num = 1000
11
12   p,q,n=0,0,0
13   p = Prime.each.find{|e| e >= rand(big_num+1..big_num+num)}
14   q = Prime.each.find{|e| e >= rand(big_num+1..big_num+num)}
15
16   n = p*q
17   l = (p-1).lcm(q-1)
18
19   print "素数␣p␣>>>␣#{p}\n"
20   print "素数␣q␣>>>␣#{q}\n"
21   print "N␣>>>␣#{n}\n"
22   print "L␣>>>␣#{l}\n"
23
24   for e in 2..l do
25     break if e.gcd(l)==1
26   end
27
28   print "公開鍵>>>␣E␣=␣#{e},␣N␣=␣#{n}\n"
29
30   for d in 2..l do
31     break if (e*d)%l==1
32   end
33
34   print "秘密鍵>>>␣D␣=␣#{d},␣N␣=␣#{n}\n"
35
36   m = c**e % n
```

```
37    re_c = m**d % n
38
39    print "暗号化>>>␣#{m}\n"
40    print "復号化>>>␣#{re_c}\n"
41  end
42
43  rsa(ARGV[0])
```

このプログラムは Ruby だけで書かれている．例えば平文を 256 と入力すると，

```
1  /Users/eri/mapleruby/lib% ruby rsa.rb 256平文
2  >>> 256素数
3    p >>> 107素数
4    q >>> 83
5  N >>> 8881
6  L >>> 4346公開鍵
7  >>> E = 3, N = 8881秘密鍵
8  >>> D = 1449, N = 8881暗号化
9  >>> 1007復号化
10 >>> 256
```

このような形で，出力してくれる．

5.1.1 RSA 暗号とは

RSA 暗号は桁数が大きい合成数の素因数分解問題が困難であることを安全性の根拠とした公開鍵暗号の 1 つである．1977 年に発明され, 発明者であるロナルド・リベスト (Ron Rivest), アディ・シャミア (Adi Shamir), レオナルド・エーデルマン (Len Adleman) ら 3 人の Family name の頭文字をつなげてこのように呼ばれている{{cite(listings2)}}.
アルゴリズムは,

- p, q: 任意の素数.
- N: p, q をかけた数.
- L: p - 1 と q - 1 の最小公倍数.
- E(Encryption exponent, 暗号化指数): L と互いに素な数．L との最大公約数が 1 となる数.
- D(Decryption exponent, 復号化指数): E*D mod L = 1 となる数.

以上 6 つの値を用いて，平文を暗号化していく．ここでの数 E と数 N のペアが公開鍵, 数 D と数 N のペアが秘密鍵となる．平文を暗号化する際は,

平文^E mod N（平文を E 乗して N で割った余り）

を行い，逆に復号化する際は，

暗号文^D mod N（暗号文を D 乗して N で割った余り）

を行う．

5.1.2 Ruby のみで計算した場合

Ruby のみで計算した場合，このプログラムでは，平文が 1000000 を越えたあたりから正しく出力ができなかったり数が大きすぎて計算ができないということが起こる．下記は平文が 2000000 だった場合の結果である．

```
1  /Users/eri/mapleruby/lib% ruby rsa_org.rb 2000000平文
2  >>> 2000000素数
3   p >>> 1459素数
4   q >>> 1493
5  N >>> 2178287
6  L >>> 1087668公開鍵
7  >>> E = 5, N = 2178287秘密鍵
8  >>> D = 652601, N = 2178287
9  rsa_org.rb:36: warning: in a**b, b may be too big暗号化
10 >>> 1481122復号化
11 >>> NaN
```

5.1.3 mapleruby を使った場合

まず先ほどの rsa_org.rb を mapleruby を用いた rsa.rb に書き換えた．

```
1  #rsa.rb
2  require './mapleruby'
3  include Math
4
5  def rsa(input)
6    c = input.to_i
7    print "平文>>> #{c}\n"
8
9    big_num = sqrt(c).to_i
10   num = 1000
11
12   p,q,n=0,0,0
```

```
13    p = RMaple.new.nextprime(rand(big_num+1..big_num+num))
14    q = RMaple.new.nextprime(rand(big_num+1..big_num+num))
15
16    n = p*q
17    l = RMaple.new.lcm(p-1,q-1)
18
19    print "素数␣p␣>>>␣#{p}\n"
20    print "素数␣q␣>>>␣#{q}\n"
21    print "N␣>>>␣#{n}\n"
22    print "L␣>>>␣#{l}\n"
23
24    for e in 2..l do
25      break if RMaple.new.gcd(e,l)==1
26    end
27
28    print "公開鍵>>>␣E␣=␣#{e},␣N␣=␣#{n}\n"
29
30    d = Mapleruby.new("eval(1/#{e}␣mod␣#{l})").exec_i
31
32    print "秘密鍵>>>␣D␣=␣#{d},␣N␣=␣#{n}\n"
33
34    x = Mapleruby.new("#{c}^#{e}").exec_i
35    m = RMaple.new.mod(x, n)
36
37    re_c = Mapleruby.new("#{m}^#{d}␣mod␣#{n}").exec_i
38
39    print "暗号化>>>␣#{m}\n"
40    print "復号化>>>␣#{re_c}\n"
41  end
42
43  rsa(ARGV[0])
```

このプログラムに関しては，mapleruby の rand 関数では乱数が同じ値になって計算がうまくいかなくなるため Ruby の rand 関数を使用している．rsa.rb を使った場合は，1000000000 までの暗号化が可能であると分かった．

```
1  /Users/eri/mapleruby/lib% ruby rsa.rb 1000000000平文
2  >>> 1000000000
3  {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
       Versions/Current/bin/maple"}
4  31699
5  {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
       Versions/Current/bin/maple"}
6  31657
```

```
7 {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
      Versions/Current/bin/maple"}
8 167238648素数
9  p >>> 31699素数
10  q >>> 31657
11 N >>> 1003495243
12 L >>> 167238648
13 {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
      Versions/Current/bin/maple"}
14 2
15 {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
      Versions/Current/bin/maple"}
16 3
17 {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
      Versions/Current/bin/maple"}
18 4
19 {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
      Versions/Current/bin/maple"}
20 1公開鍵
21 >>> E = 5, N = 1003495243
22 {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
      Versions/Current/bin/maple"}秘密
      鍵
23 >>> D = 100343189, N = 1003495243
24 {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
      Versions/Current/bin/maple"}
25 {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
      Versions/Current/bin/maple"}
26 96903649
27 {:MAPLE_PATH=>"/Library/Frameworks/Maple.framework/
      Versions/Current/bin/maple"}暗号
      化
28 >>> 96903649復号化
29 >>> 1000000000
```

MAPLE_PATH **の文章のところで** mapleruby **が動いている．**

5.2 行列での検証

まず，以下のようなプログラムを作った．

```
1  require './mapleruby'
2
3  a = 3
4  b = 3
5  c = [[1,2,1],[4,5,6],[7,8,9]]
6
7  p x = RMaple.new.matrix(a, b, c)
8
9  p RMaple.new.matrixinverse(x)
10 p RMaple.new.eigenvectors(x)
```

これは 3 行 3 列の行列を
プログラムである．これ

```
/Users/eri/mapleruby/lib% cat result.txt
                                [1    2    1]
                                [            ]
                                [4    5    6]
                                [            ]
                                [7    8    9]
```

matrix関数で生成された行列①

```
/Users/eri/mapleruby/lib% cat result.txt
                              [-1/2    -5/3    7/6 ]
                              [                    ]
                              [ 1       1/3    -1/3]
                              [                    ]
                              [-1/2      1     -1/2]
```

①の行列を
matrixinverse関数を用いた結果

```
/Users/eri/mapleruby/lib% cat result.txt
[          15.28732829          ]
[                               ]
[-0.143664146 + 0.6097889260 I],
[                               ]
[-0.143664146 - 0.6097889260 I]

    [0.1603653687 , -0.9286675356 + 1.185380029 I ,

    -0.9286675356 - 1.185380029 I]

    [0.6455963335 , -0.3303739256 - 0.9609839090 I ,

    -0.3303739256 + 0.9609839090 I]

    [1. , 1. , 1.]
```

①の行列を
eigenvectors関数を用いた結果

図 9　プログラムの出力結果

これと同じものを Maple で計算すると図 10 のようになる．
よって正しく行列の計算ができた事が分かる．

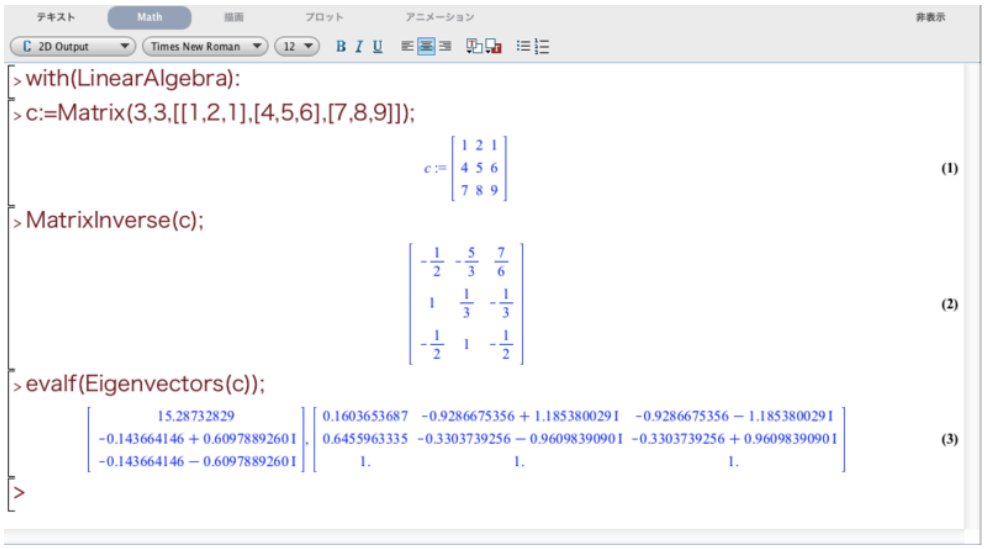

図 10　Maple の出力結果

6　考察

6.1　初期バージョンとバージョン 2 の比較

初期バージョンとバージョン 2 では，後者の方がプログラム自体の行数は多い．しかしそれは動的メソッドを実装するにあたって重複分をまとめた関数を増やしたためである．今後新たに数学関数を実装していくと仮定した場合，既に重複分がまとめてある関数を実装するならば，プログラムに書き足すのは実装する数学関数についての関数のみであるので，かなり簡潔で済む．一方で，既に実装されているものと重複がないまたは出力が同じでない場合，初期バージョンと同じようにプログラムを書くことになる．加えて初期バージョンは Maple に
期バージョンの方で実装

	初期バージョン	バージョン2
利点	Mapleのプログラムに慣れた人には， 一目でMapleに送る式が分かるため 新しい関数を実装するのが容易．	一度重複分をまとめた関数を プログラムしてしまえば， その後新たに関数を実装したとしても プログラム量が少なくて済む． 将来的に見てプログラムが短くなる．
欠点	コピーアンドペーストしたような 同じ内容の関数が多くできてしまう． 結果としてプログラムが長くなる．	例外にあたる関数や， 今までに実装されてない出力を持つ 関数の場合，実装時に結局初期バージョンと同様のプログラムしなければならない．

図 11　各バージョンの利点，欠点

6.2　mapleruby を使うメリット，デメリット

最大のメリットは，Ruby だけで桁数の大きい計算や複雑な数学関数を必要とする計算を完結させられることである．今後扱える関数を増やせば，利用者が Maple について詳

しくない人でも Ruby のプログラムを書くだけで数値計算処理が可能になるだろう．また Ruby ライブラリにある Test::Unit と並行して使用すれば，求めたい解が分かっている際に自分が書いたプログラムで正しく解が導けるのかテストすることも可能である．

デメリットは大きく 3 つある．1 つ目は Ruby は無償で使えるが Maple は有償である点．よって，利用者が限られてしまう．2 つ目は RMaple クラスの中に使いたい関数が実装されてない場合，自力で関数を実装するか Mapleruby クラスを直に使うかしなければならない点．よってプログラミングに Maple の知識が必要になってくるためメリットで挙げた「Ruby プログラムを書くだけで数値計算処理が可能になる」のが難しくなる．3 つ目は桁数の大きい計算は確かにできるものの，処理に多少時間がかかってしまう点．処理速度を上げようと思うと Maple 自体の処理速度を上げなければならない．

7 おわりに

7.1 今後の課題

今回は限られた関数のみを選抜して実装したが, 他にもたくさんの数学関数が Maple には用意されている．RSA 暗号のプログラムを mapleruby で実装した際に直接 Mapleruby クラスに送ることで対応した等式の解を出力する eval のようなよく使われるであろう関数へ対応させる事や，累乗や四則混合の簡単な計算は現状直接 Mapleruby クラスに送るような形をとっているため，そちらについてもうまく対応させたい．他にも桁数が大きな数値はそもそも Ruby の変数が扱いきれない場合も考えられるので，そこにうまく対処できるような関数が欲しい．

また，Maple が綺麗にグラフを描画できる数式処理ソフトウェアであることを利用して，mapleruby もグラフ描画に対応させる．Maple は CUI 版でのグラフがかなり見にくく，二次元ならまだしも三次元になると何が何だか分からないグラフになるため，画像としてのグラフを出力できるような関数を実装する．

8 参考文献

listings1 「Maple（メイプル）とは」，サイバネット，http://www.cybernet.co.jp/maple/product/ アクセス.

listings2 「RSA 暗号」，Wikipedia，https://ja.wikipedia.org/wiki/RSA 暗号，2017/02/01 アクセス.