Crystal による数値計算入門

牧野淳一郎

March 17, 2020

PDF 版は こちら[1]

[1] ../intro_crystal.pdf

Contents

Chapter 1

インストール (2020/1/18)

Crystal による数値計算入門です。この入門では、常微分方程式、特に多体問題等の運動方程式、力学系の数値積分と結果の解析に関する基本的な知識と技法を身につけることを目標とします。

今回はまずインストールから。

Crystal 公式の *Web* ページ[1]に書いてあるのでその通りに、という話ですが、 Windows10 であれば、WSL で Ubuntu をいれてそっちでやりましょう。

WSL のいれかたは一杯ドキュメントがありますが、たとえば ここ[2]に従うことでよいかと思います。

Crystal のインストールができたら、ターミナルのコマンドプロンプトに crystal といれると以下のような出力がでるはずです。

```
|gravity> crystal
Usage: crystal [command] [switches] [program file] [--] [arguments]

Command:
    init                     generate a new project
    build                    build an executable
    docs                     generate documentation
    env                      print Crystal environment information
    eval                     eval code from args or standard input
    play                     starts Crystal playground server
    run (default)            build and run program
    spec                     build and run specs (in spec directory)
    tool                     run a tool
    help, --help, -h         show this help
    version, --version, -v   show version

Run a command followed by --help to see command specific information, ex:
    crystal <command> --help
```

(以下、 「—gravity¿ 」 はプロンプトだとします) -v でバージョンがでるとのことなのでやってみます。

[1] https://crystal-lang.org/install/

[2] https://www.atmarkit.co.jp/ait/articles/1903/18/news031.html

```
|gravity> crystal -v
Crystal 0.33.0 [612825a53] (2020-02-14)

LLVM: 8.0.0
Default target: x86_64-unknown-linux-gnu
```

さて、どの言語でもまずは Hellow world! ということで

```
print "Hellow World!\n"
```

と 1 行書いた、hellowworld.cr というファイルを作って

```
|gravity> crystal run hellowworld.cr
Hellow World!
```

これで、このプログラムの実行ができたことになります。ファイルの拡張子は.cr にするのが普通のようです。 crystal run foo.cr で、foo.cr というプログラムがコンパイルされてできた実行ファイルが実行されます。

`print` なんとか

は画面 (ファイル等にもできますが) に 「なんとか」を出力する、Crystal の文法的には関数で、 print という関数があらかじめ定義されている、ということになります。そのあとの ”Hellow World!\n” ですが、「”」で囲まれたものが「文字列」を直接表すもの (「文字列」型の定数) ということになります。C 言語等と同じで文字列定数の中で ”\” は特別な意味があり、

```
"\n" 改行
"\"" 「"」を表示する時
"\\" 「\」を表示する時
"\t"  タブ
```

というあたりが使うことがあるものです。

この章のまとめです。

- Crystal のインストール方法は *Crystal* 公式の *Web* ページ[3]を参照。
- Windown 10 なら WSL をいれて Ubuntu を動かすとあとが楽。
- crystal で簡単なヘルプが、 crystal run foo.cr でプログラムの実行ができる。
- プログラムの中では print で出力できる。文字列は 「”」で囲む。文字列の中では ”\n”で改行になる。

次から、もうちょっと本格的なプログラムにはいっていきましょう。

[3] `https://crystal-lang.org/install/`

1.1 課題

1. 自分の計算機に Crystal をインストールし、

```
crystal
crystal -v
```

   をそれぞれ実行し、結果が本文と同様のものであることを確認しなさい

2. Hellow World! を出力するプログラムを作成、実行し、結果を確認しなさい。

1.2 参考資料

https://crystal-lang.org/reference/overview/

https://crystal-jp.github.io/introducing-crystal/assets/pdfs/introducing-crystal.pdf

Chapter 2

文法1 変数と型 (2020/1/18)

2.1 変数とプログラムの基本

Crystal による数値計算入門 2 回目です。多くのプログラム言語、特に Crystal の文法のもとになっている Ruby と同じように、 Crystal では「変数」というものを使って、それに「値」をいれたり、その「値」を使って計算したりできます。

```
a=1
b=2
print a+b, "\n"
```

を作って、実行すると

```
|gravity> crystal run add.cr
3
```

という出力になります。 関数は、普通は foo(a,b) といったふうにその引数を 括弧でくくりますが、Crystal は近年の多くの言語と同じように関数のあとの括弧を省略できます。なので

```
print(a+b, "\n")
```

と

```
print a+b, "\n"
```

は同じ意味になります。さて、

```
a=1
```

は、多くのプログラム言語で使っている代入の表現で、等号の左側の変数の値を右側の式の値にします。もしも、左側の変数にすでに何か値がはいっていたら、その値は捨てられて新しい値に書換えられます。 「=」という等号記号なのに、数学的な等号ではなくて、「代入」であることには注意が必要です。

「a」は変数名で、ここでは 1 文字ですがアルファベットの小文字から始まり、アルファベット、数字、「_」(アンダースコア) からなる文字列であればなんでも使えます。長さの制限も特にないはずです。つまり、

```
a1
a_1
abcdefghijklmnopqrstuvwxyz01234567890_ABCDEFGHIJKLMNOPQRSTUVWXYZ
z_y_x_w_v_u_____________0
```

といったものはどれも変数名に使えます。

プログラムの意味としてもうひとつ重要なことは、プログラムは上から 1 行づつ順番に実行される、ということです。つまり

```
a=1
a=2
b=a
```

と

```
a=1
b=a
a=2
```

は結果が違い、上では b は 2、下では 1 になります。

2.2 型 (クラス)

さて、a という名前の変数の値が 1 になるわけですが、計算機の中で 1 という値を表現する方法は色々ありえます。代表的なものが、

- 整数
- 浮動小数点数
- 文字列

といったところです。例えば Fortran では (暗黙の宣言は使わないとして)

```
integer i
real*8  a
```

といった形で、また C/C++ なら

```
int i;
double a;
```

といった形で最初に変数の「型」を宣言し、それから使う、ということをします。一方、Python や Ruby (と書くのは面倒なので以下大抵単に Ruby を例にします) では、 Crystal の例と同様に

```
a=1
```

といきなり書くことができます。Ruby では、言語が動的である、つまり、型が実行時に定まることで、宣言を不要にしています。上のケースでは、右辺の式 (というか定数 1) が、Ruby の文法として「整数定数」を表すことに決まっているので、その値が整数になり、それを代入される変数 a の型は、代入された時点で整数になるわけです。Crystal は、コンパイルされる言語なので実行時ではなくコンパイルの時点で型が決まります。明示的に型が宣言されていない変数については「型推論」を行います。

```
a=1
```

の場合では、右側の型が整数なので、左側の型も整数にしておこう、ということになるわけです。

```
a=1.0
```

と書けば、 1.0 は浮動小数点数型定数なので、a の型も浮動小数点型になります。

```
a=1
print a.class,"\n"
a=1.0
print a.class,"\n"
b=2
print a+b, "\n"
```

を実行すると

```
Int32
Float64
3.0
```

となり、 1 の代入後の a の型 (クラス) は Int32、 1.0 の代入の後は Float64 になっていて、 a+b の計算結果も 3.0 となっていて b が浮動小数点に変換されてから計算されたことがわかります。ここで、 変数名. 関数名 とコンマでつなぐのは、 その関数 がその変数の属する型の「インスタンスメソッド」で、そのメソッドを foo に適用している、ということです。

といわれてもよくわからないですよね、、、 bar が (古い)Fortran や C でのやサブルーチンと思ってもらってよいのですが、2 つ違いがあります。

1. Fortran や C では 関数名 (変数名) と書くところを、 変数名. 関数 (メソッド) 名 と書く
2. 変数の型が違えば、関数名が同じでも別の関数になって、動作を別に定義できる。

1 は本当に見かけの問題ですが、どうしてこうしたいかは次の章で出す例で詳しく解説します。2 についても、次の章でもう少し色々なメソッドの例をみてからにしましょう。

Crystal の整数型には Int8, Int16, Int32, Int64, Int128 と、それぞれの符号なし型である UInt8, ... Uint128 の 12 種類、浮動小数点型には Float32 と Float64 の 2 種類があります。整数定数は、

```
1 : Int32
1_i8 :  Int8
1_u8 :  UInt8
```

で、同様に 16, 32, 64, 128 を指定できます。浮動小数点型定数は

```
1.0:     Float64
1.0_f32: Float32
1.5e10:  Float64
```

という感じです。

2.3 まとめ

- Crystal のプログラムは、 crystal run foo.cr (foo.cr はプログラムがはいったらファイルの名前) で実行される。Ruby と違ってコンパイルして実行されるが、このコマンドで全部やってくれる。
- プログラムの中では文字列で名前を決めた「変数」に、「値」をいれたり、その変数を使った数式を書くことで計算させてその結果を別の変数にいれたり、出力したりできる。
- 整数と浮動小数点数は「型」が違う。変数の型は、代入される値の型になる。
- よく使う型としては整数 (Intxx, UIntxx)、浮動小数点 (Float32, Float64)、文字列 (String) がある。型名は大文字で始める。
- 整数を使って計算した結果は整数に、浮動小数点数を使って計算した結果は浮動小数点数になる。
- 変数.class で、その変数の型を表す文字列になる。ここで、class は「メソッド」で、他の言語でいうところの関数だが、その辺は次の章でもう少し詳しく述べる。

2.4 課題

1. 2 つの変数 a, b に整数値をいれ、四則演算 (加減乗除) の結果を出力するプログラムを作成・実行し、結果が正しいことを確認せよ。
2. 浮動小数点 (Float64) の値で同様なことを行え。
3. 2 つの変数 a, b に値をいれ、出力したあと、a, b の値を入れ換えて、また出力するプログラムを作成、実行して結果が正しいことを確認せよ。
4. 整数と浮動小数点数の四則演算の結果がどうなるか、値と型を、Crystal のドキュメントの記載を確認すると共に、実際のプログラムを作成して確認せよ。

2.5 参考

https://crystal-lang.org/api/0.32.1/Int.html

https://crystal-lang.org/api/0.32.1/Float.html

Chapter 3

文法2 入力と繰り返し (2020/1/18)

Crystal による数値計算入門 3 回目です。前章では、プログラムの中で変数に数値をいれて、それを使って計算し、その結果を出力してみました。しかし、これなら電卓でもできるわけで、あんまりプログラム書いて嬉しい感じがしません。また、計算機の能力を有効に使っている感じもあまりしません。

というわけで、この数値計算入門では、手計算ではできないような繰り返し計算で数値的に微分方程式を解いていくわけですが、その前にプログラムへの入力とその繰り返しの方法をみておきます。

3.1 入力

まずは前章と同じ変数 a, b ですが、プログラムの中で値を設定するのではなく、キーボードから入手できるようにしてみましょう。

```
print "enter a:\n"
a=gets.to_s.to_i
print "enter b:\n"
b=gets.to_s.to_i
print a+b, "\n"
```

これに 2, 4 という入力を与えた結果は以下のようになります。キーボードからの入力としては enter a: がでてから 2 をいれてリターン、次に enter b: がでてから 4 をいれてリターン、だと思いますが以下の例は入力が先にでています。なお、 crystal foo.cr と、 run を省略しても run があるのと同じになるようです。

```
|gravity> crystal add-with-input.cr
 2
 4
enter a:
enter b:
6
```

gets.to_s.to_i という大変謎な表現がでてくるので、が何か、というのをみていきます。

gets は、標準入力からの入力 (改行まで) を読みとり、それを文字列として返します (String 型)。但し、入力がなかった時、例えば、入力をファイルからとして、ファイルの最後まで読んでしまった時には、String 型ではない、 nil というものを返します。これは Ruby の nil と同じで、「そこに何もない」ということを表すもの、つまり、この場合、 gets が文字を読まなかった、ということを表すものです。Crystal では (Ruby と同様) nil は Nil という型の、もつことができる唯一の値です。なので、 gets という関数は、 String 型または Nil 型の値を返すことができる、ということになります。このような、複数の型をもてる、ということを Crystal では

```
String|Nil
```

と表現し、特に、基本的にある形 Foo なんだけど Nil になれる、ということを

```
Foo?
```

と表現します。従って、関数 gets の型は String ではなく String? である、ということになります。こういう仕掛けにしておくことで、文字列が返ってこなかった時の処理をプログラム側で容易に記述できることになります。

なので、ちゃんとしたプログラムにするなら、 gets の返した値が String 型であるかどうかをチェックして、そうでなかったらエラー終了するとか、さらに文字列が数字でなかったらエラー終了するとかするべきですが、その辺はコンパイラと実行時ライブラリに任せるなら、Nil が返ってきたら無理矢理 (長さ 0 の) 文字列にしちゃう、文字列であればそのままに、という関数を使えばいいことになります。それをするのが to_s です。

3.1.1 関数と「メソッド」

関数というと、 to_s(なんとか) のように書くのが数学での普通ですが、いわゆる「オブジェクト指向言語」ではある型の変数や定数に対する関数をなんとか.to_s のように、 変数 + "." + 関数という形で書けるようになっています。これを、インスタンスメソッドと呼びます。インスタンスとは、ある型 (クラス、ここでは型とクラスは同じものです) の、変数ないし定数のことです。単にメソッドといわないのは、インスタンスメソッドの他にクラス自体のメソッド、クラスメソッドというものがあるからですが、通常メソッドというとインスタンスメソッドのことになります。

また、同じ名前でもクラス毎に別の関数として定義できるので、この例のように、 gets.to_s で、gets が nil を返すと Nil#to_s が、String を返すと String#to_s が (この場合何もしないで受け取った文字列を返す) 呼ばれることになります。

さらに、ある型の値を整数型に変換する関数 (メソッド) が to_i です。普通の関数だと to_i(to_s(gets)) となるわけで、括弧の対応を考えつつ後ろから読んでいかないと意味がわからないわけですが、gets.to_s.to_i だと、 「標準入力を読んで、文字列に強制して、整数に変換する」となって理解しやすい、というのがこの形式のメリットであり、それ以上の本質的な意味はないと思いますが、理解しやすい、というのはプログラミングの上では極めて重要です。

「オブジェクト指向言語」以前の言語、例えば Fortran77 や C 言語では、ある名前の関数が受け取る引数は決まった型のものでした。Fortran77 では、言語の側で提供する数学関数や read/write は特別だし、C でも varargs という機能で scanf/printf といった入出力関数を実装していますが、我々が書くプログラムで使う機能ではありません。オブジェクト指向言語では、引数の型や数が違う関数は「別のもの」になり、違う型のメソッドは従って全て別のものになります。

なお、この、「別のものになる」という機能が必ずしも嬉しくないことがあります。このノートでの主題の 1 つになりますが、常微分方程式の数値積分を考えてみます。そうすると、従属変数が 1 つだと、Float64 なり Float32 ですが、多変数だと配列 (Array。本章の後半でてきます) になるし、もっと複雑なデータ型を使いたいこともあります。そうすると、型毎に数値積分公式、例えばルンゲクッタとかシンプレクティック公式とかいったものを実現する関数を書く必要がでてくるわけです。

Ruby のような動的言語では、型を指定しないで関数を書くことで、受け取った型がもっているメソッドを使って数値積分公式を実現することができます。Crystal でも、実行時ではなくコンパイル時に型推論をしますが、同様のことができます。(あんまり意味がわからないかもですが、次章あたりで実例をみます)

3.2 入力と繰り返し

とはいえ、単にキーボードからいれた数字を変数に入力するだけでこんな大変なのかよ？C++ でも Fortran でももっと簡単だぜ、と思われたのではないかと思います。確かに、1 つ読むだけなら、

```
C++:      a<<cin;
Fortran:  read(*,*) a
```

ですむわけで、こっちのほうが簡単です。

とはいえ、多少複雑な処理を、となると Crystal (というかその元になっている Ruby) ではより簡潔かつ間違いにくく書ける、という場合があります。以下、そういう例をみていきます。

今、 sum-sample.in というテキストファイルに以下の数値がはいっているとします。

```
1
2
3
4
5
6
7
8
9
10
```

この合計を計算し、出力するプログラムを考えてみます。普通の言語での考え方は、

1. 合計をいれる変数の値を 0 にする
2. 入力ファイルの終わりに到達するまで、1 行読んで、整数に変換した値を合計をいれる変数に加算、を繰り返す。
3. ファイルの終わりに到達したら変数を出力する。

となります。以下はそれを素直に表現したものです。

```
sum=0
while s=gets
  sum += s.to_i
end
print "sum=#{sum}\n"
```

実行結果は

```
|gravity> crystal sum.cr < sum-sample.in
sum=55
```

”¡” は「リダイレクト」で、標準入力を、キーボードから入力する代わりにファイルから読むようにします。

このプログラムでは 3 つ新しいことがでてきます。

1. while
2. +=
3. 文字列の中の #{sum}

以下順番に解説します。

3.2.1 While と条件式

まず while は

```
while 条件式
  色々処理
end
```

の形で、まず条件式が真であれば「色々処理」の部分を実行、また条件式を評価して、、、を、条件式が偽になるまで繰り返すものです。では「真」とか「偽」は何か、ということですが、 Crystal の文法では

- nil は「偽」
- Bool 型の false は「偽」
- いわゆる「null pointer」(とりあえず解説はあとまわし) は「偽」
- それ以外は全て「真」

です。Bool 型は true と false の 2 つの値をとる型で、通常の論理演算を行うことができます。

```
== 比較演算子。一致していれば真
!= 比較演算子。一致していなければ真
^  排他的論理和
|  論理和
&  論理積
!  否定
```

なお、 C の影響を受けた多くの言語と同様、 &, — と &&, —— があり、前者は「ビット毎の論理演算」、後者は「真か偽か」を返すものですが、 &&, —— はちょっと変わっていて、

```
&&: 左辺が偽でなければ、右辺の値、左辺が偽ならその値
||: 左辺が偽でなければ、左辺の値、左辺が偽なら右辺の値
```

となります。また、 これも C と同様、代入も「式」であり、その値は左辺に代入された値となります。なので、

```
while s=gets
   色々
end
```

と書くと、

1. まず s=gets を実行し
2. その結果が nil でなければ、つまり文字列であれば「色々」の部分を実行し、1 に戻る。でなければ繰り返しを終了する

という処理をすることになります。

Int、Float に対しては大小比較

```
<
<=
>
>=
```

があり、常識的な意味になります。

3.2.1.1 While 以外の制御構造

while は条件が成り立っている間の繰り返しですが、制御構造としては、他に if, unless, until があります。ここでまとめておきましょう。

3.2.1.2 if と unless

```
if 条件式 1
  実行部 1
elseif 条件式 2
  実行部 2
....
else
  実行部 n
end
```

の形で、条件式 1 が真なら実行部 1 を、条件式 1 が偽で条件式 2 が真なら実行部 2 を、、、と elsif が沢山あれば順番にチェックしていって、全て偽なら (else の部分があれば) 実行部 2 を実行します。

`unless` は、`if` の反対で

```
unless 条件式 1
  実行部 1
else
  実行部 2
end
```

で、条件式 1 が偽なら実行部 1 を、真なら 2 を実行します。

なお、if ... end まで全体も式であり、値をもちます。これは、実際に実行された実行部の値になります。普通のプログラムであんまり使わないですが、自分で関数を定義する時にはその返す値を関係に表現できます。

また、 Ruby でもよく使いますが、 if をあとに置く、以下のような形式があります

```
a = x if x > 0
```

これは

```
if x > 0
  a=x
end
```

と同じです。

3.2.1.3 until

```
until 条件式
  色々
end
```

は

```
while !条件式
  色々
end
```

と同じです。

3.2.2 +=

C 言語 と同様に +=、 -=、 *= といった演算が定義されています。但し、 ++ (+=1)、– 等はありません。 += に限らず、全ての 2 項演算子について = がついたものが機械的に利用可能で、

```
a += b
```

は

```
a = a + b
```

と同じ (+のところを任意の演算子として) です。なので、この列での

```
sum += s.to_i
```

は、 sum に s を整数に変換した値を加算する、となります。

3.2.3 文字列の中の #{sum}

文字列の中に #{ 式 } と書くと、その部分が式の値 (を、 to_s で文字列に変換したもの) で置き換えられます。

3.2.4 ここまでのまとめ

ということで、もう一度

```
sum=0
while s=gets
  sum += s.to_i
end
print "sum=#{sum}\n"
```

を見ると、これは

1. sum を 0 にし
2. 標準入力から 1 行読み、結果が nil なら 4 にいく、そうでなければ
3. 結果を数値に変換し、 sum に加算し、2 に戻る
4. ”sum=値” を出力

ということになるわけです。最初のサンプルでは、

```
a=gets.to_s.to_i
```

と、gets の結果が nil である場合を考慮する必要がありましたが、この繰り返しの例では

```
sum += s.to_i
```

で、余計な to_s がないことに注意して下さい。これは、 while のところで条件を評価しているので、ここでは s が niil でないことが「コンパイラに」わかっていて、 s の型が String になっているからです。

3.3 制御構造を使わないプログラミング

while や if を使うのはどんな言語でも基本的なことではありますが、間違いのもとでもあります。ということで、ここでは、明示的にそういうものを使わないプログラムを作ってみます。これは、高尚ないいかたでは「関数的プログラミング」ということになります。以下のプログラムを考えます。

```
s=gets("")
a=s.to_s.split
aint = a.map{|x| x.to_i}
sum = aint.sum
print "sum=#{sum}\n"
```

実行結果は、 sum.cr の場合と同じなので省略します。つまり、このプログラムでも、ファイルの全行の数値を読んで、その合計をだす、ということはできています。

このプログラムは何をしているかを 1 行づつみていきます。

```
s=gets("")
```

は、おなじみの gets ですが、("") がついているのが今までと違います。関数 gets は delimiter という引数をとることができて、それでどこまで読むか、を指定していて、デフォルトは "\n" になっています。なので、単に

```
gets
```

と書くと、1 行読むのですが、そこで "" と長さ 0 の文字列を指定するとファイル全体を一度に読むことができます。

a=s.to_s.split

は、(また nil かもしれないので文字列にした値)、 split という関数にファイル全体の文字列を渡します。 split は、デフォルト (引数なし) では、空白文字、タブ、改行等で文字列を切り分けて、それらからなる「配列」(Crystal では Array) に変換します。Crystal の配列は、定数で書くと、例えば

```
[1, 2, 3]
```

といったもので、これは Int32 型の配列 (型としては Array(Int32)) となります。

```
a=  [1, 2, 3]
```

とすれば、 a[0], a[1], a[2] がそれぞれ 1, 2, 3 です。C 言語風に配列の添字は 0 からです。例えば、

```
"1 2 3".split
```

の実行結果は

```
["1", "2", "3"]
```

ということになります。プログラムの例では、 a は

```
["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"]
```

となるでしょう。次の

```
a.map{|x| x.to_i}
```

で、map は、配列 a の各要素に {} 内の操作をした新しい配列を作ってそれを返すメソッドです。{} 内では ——の中、この場合では x が、元の配列の要素の値になり、その後の部分で x を使って色々操作をした最後の文の値が新しい配列の対応する要素の値になります。なので、 aint は

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

になるわけです。

sum = aint.sum

は、配列に対して、 sum というメソッドがあらかじめ定義されていて、それは全要素の合計を返す、というものなので、それを単に呼んでいます。

まあその、この例では、考え方はともかくなんかかえってプログラムは長くて複雑ではないか、という気もしますが、では以下ではどうでしょう？

```
sum = gets("").to_s.split.map{|x| x.to_i}.sum
print "sum=#{sum}\n"
```

これでは、ファイルを全体呼んで (gets) 文字列に強制的にして (to_s) 要素毎に配列にして (split)、各要素を整数にして (map{—x— x.to_i}) 合計する (sum) のを、全て文字列全体や配列全体へのメソッドの適用の形で実現していて、余計な変数等もなくて意味も明瞭ではないかと思います。

3.4 もう少し複雑な例: 表の読み込みと処理

以下のような 2 次元の表があるとします。

```
 9 26 14 74  3 82 86 75 82 92
28 14 48 32 90 78 20 53 68 21
53  3 44 92 84 98  0 16 38 98
79 79  5 76 51  2 70 83 14 54
16 46 53 42 64 24 49 99 46 84
36 49 40 68 59  9  6 53 74 13
 4 98  6  7 49 38 18 75 62 66
84  8 25 12 16 39 18 34 51 34
 4 46 98 40 37 39 27 33 93 58
50 78 76 70 30 29 78  8 87 17
24 83 64 83 38 16 15  0 39 18
50 80 98 86 52 40  1 60 13 52
78 80 31 84 39 64 74 30 45 11
11 84 84 49 83 58 17 54 59 26
34  7 50 60 35 11 85 15 89 51
65 11 10 10 98 69 77  9 18 66
50 26 59 57 27 36 80 74 35 68
 3 42 40 84 81 34 30 70 86 76
98 88 67 27 63 88 45 74 65 82
16 59  5 73 58 72  6  5 36 65
```

このようなデータは色々なところで現れます。常微分方程式の数値解なら 1 行がある時刻での解、というファイルかもしれないし、多数の粒子を使ったシミュレーションや格子を使ったシミュレーションの結果ならある時刻での各粒子や格子での値が 1 行にはいっているでしょう。

この表について、以下の処理をするプログラムを作ってみます。

1. 各列の合計を最後に出力
2. 各行の合計を行列毎に出力

3. 4,5,6 列目の合計を行毎に出力した上で、その合計も出力

上でやったように、ファイル全体を読み込んでから処理、という方法を考えてみます。そうすると、まずは

```
gets("").to_s
```

ですね。これを行毎に分割するのは split("\n") でできて、これで 1 行が要素になった配列ができます。さらに、その行毎に、先ほどと同じ split.map{—x— x.to_i} を行えば、各行が整数の配列にかわります。つまり

```
gets("").to_s.split("\n").map{|s| s.split.map{|x| x.to_i}}
```

でよさそうです。ところが、以下を実行してみると

```
a=gets("").to_s.split("\n").map{|s| s.split.map{|x| x.to_i}}
print  a.size, "\n"
```

21 という答になって、余計なものがはいっていることがわかります。これは、ファイルの最後の文字が "\n"で、 split したので、最後に「何もない行」が要素としてできてしまったからです。これを防ぐには

```
a=gets("").to_s.chomp.split("\n").map{|s| s.split.map{|x| x.to_i}}
```

と、文字列にしたあとで chomp というメソッドで、最後の "\n" を取り除きます。

```
a=gets("").to_s.chomp.split("\n").map{|s| s.split.map{|x| x.to_i}}
a.each{|x| p x}
```

を実行してみると

```
[9, 26, 14, 74, 3, 82, 86, 75, 82, 92]
[28, 14, 48, 32, 90, 78, 20, 53, 68, 21]
[53, 3, 44, 92, 84, 98, 0, 16, 38, 98]
[79, 79, 5, 76, 51, 2, 70, 83, 14, 54]
[16, 46, 53, 42, 64, 24, 49, 99, 46, 84]
[36, 49, 40, 68, 59, 9, 6, 53, 74, 13]
[4, 98, 6, 7, 49, 38, 18, 75, 62, 66]
[84, 8, 25, 12, 16, 39, 18, 34, 51, 34]
[4, 46, 98, 40, 37, 39, 27, 33, 93, 58]
[50, 78, 76, 70, 30, 29, 78, 8, 87, 17]
[24, 83, 64, 83, 38, 16, 15, 0, 39, 18]
[50, 80, 98, 86, 52, 40, 1, 60, 13, 52]
[78, 80, 31, 84, 39, 64, 74, 30, 45, 11]
[11, 84, 84, 49, 83, 58, 17, 54, 59, 26]
[34, 7, 50, 60, 35, 11, 85, 15, 89, 51]
[65, 11, 10, 10, 98, 69, 77, 9, 18, 66]
[50, 26, 59, 57, 27, 36, 80, 74, 35, 68]
[3, 42, 40, 84, 81, 34, 30, 70, 86, 76]
[98, 88, 67, 27, 63, 88, 45, 74, 65, 82]
[16, 59, 5, 73, 58, 72, 6, 5, 36, 65]
```

という感じの出力になるはずです。ここで each は map に似ていますが、単に {} の中を実行するだけで新しい配列を作らないものです。 p は、適当なフォーマットで出力する、という割合便利な関数で、上のように配列だと [] の中で各要素を 「,」で区切って出力してくれます。

さて、このようにして配列ができてしまうと、後は割合簡単ですが、まずは 2 番めの「各行の合計を行列毎に出力」をやってみましょう。プログラムは

```
a=gets("").to_s.chomp.split("\n").map{|s| s.split.map{|x| x.to_i}}
a.each{|x| print x.sum,"\n"}
```

となりますね。もちろん、

```
gets("").to_s.chomp.split("\n").map{|s| s.split.map{|x| x.to_i}}.each{|x| print x.sum,"\n"}
```

でも同じです。 「.] の前や後で改行して

```
gets("").to_s.chomp.split("\n")
  .map{|s| s.split.map{|x| x.to_i}}.each{|x| print x.sum,"\n"}
```

でも大丈夫です。結果は

```
|gravity> crystal print_line_sum.cr < 20x10table.in
543
452
526
513
523
407
423
321
475
523
380
532
536
525
437
433
512
546
697
395
```

となるはずです。

各列の合計は色々な考え方があります。普通の考え方はまず、要素の数だけ 0 が並んだ配列をつくって、繰り返しで各行の値を足す、となるでしょう。繰り返しに each を使うと

```
a=gets("").to_s.chomp.split("\n").map{|s| s.split.map{|x| x.to_i}}
sum=Array.new(a[0].size,0)
a.each{|x|  x.each_index{|i| sum[i]+=x[i]}}
p sum
```

こんな感じです。実行結果は

```
|gravity> crystal print_column_sum.cr < 20x10table.in
[792, 1007, 917, 1126, 1057, 926, 802, 920, 1100, 1052]
```

です。ここで、 Array.new(size, value) は、 全要素の値が value で要素数が size の配列を作ります。配列 a に対して a.size は要素数です。この new は、インスタンスメソッドではありません。 Array はクラスそのものであって、その型の変数ではないからです。なので、new は「クラスメソッド」の例になり、そのクラスの変数を新しく作るメソッド、ということになります。ないところから新しく作るので、インスタンスメソッドではできないわけです。

さて、1 行の数値の合計だと a.sum ですんだわけですが、各要素毎の和、となるとそうはいきません。Array クラスに対して + 演算子は定義されてますが、それは単に二つの配列を連結するものだからです。ここでは、 sum を一般化した reduce メソッドを使ってみます。

```
a=gets("").to_s.chomp.split("\n").map{|s| s.split.map{|x| x.to_i}}
p a.reduce{|sum,x|  sum=sum.map_with_index{|val,i| val+x[i]}}
```

ここで a.reduce{—sum,x— 何か } は、

1. sum の値を a[0] にする
2. a[1] から最後の要素までについて「何か」の部分を実行する

となって、これの値は最後に実行された何かの値です。

```
sum.map_with_index{|val,i| val+x[i]}}
```

のほうは、 sum の各要素について、その対応する添字も使って新しい値を計算し、それがはいった配列を作ります。なので、この場合は、sum の各要素が sum[i]+x[i] で置き換えることになり、これを各行について実行することで合計が求まる、ということになります。

最後に、「4,5,6 列目の合計を行毎に出力した上で、その合計も出力」です。こちらは普通に、 each で sum に加算していくなら

```
sum=0
gets("").to_s.chomp.split("\n").map{|s| s.split.map{|x| x.to_i}}
  .each{|x| localsum=x[3..5].sum
  print localsum,"\n"
  sum+= localsum}
print "Total=", sum, "\n"
```

ですが、合計に sum を使うなら、

```
sum=gets("").to_s.chomp.split("\n").map{|s| s.split.map{|x| x.to_i}}
  .map{|x| localsum=x[3..5].sum
  print localsum,"\n"
  localsum}.sum
print "Total=", sum, "\n"
```

です。これではあまり簡単になってないですが、各行での和は書かないなら

```
sum=gets("").to_s.chomp.split("\n").map{|s| s.split.map{|x| x.to_i}}
  .map{|x| x[3..5].sum}.sum
print "Total=", sum, "\n"
```

と簡単になります。ここで x[3..5] は配列 x の (最初を 0 として 3 番目から 5 番目の要素からなる配列です。もちろん、x[3..5].sum の代わりに x[3]+x[4]+x[5] でも同じです。

3.5 まとめ

本章では、Crystal の文法と機能について、以下を学んだ。

- gets: 文字列入力関数
- to_s: 何かを文字列に変換するメソッド
- to_i: 文字列を整数値に変換するメソッド
- String 型: 文字列型
- nil: Nil 型、「何もない」ことを表す
- 複合型: String—Nil で、そのどちらかの形であることを表す
- 制御構造: while, if, unless, until
- Bool 型、論理演算
- 数値型の大小比較
- += の形式の演算
- Array (配列) 型
- String のメソッド split、chomp
- Array のメソッド map、each、each_with_index、sum、reduce
- 型のクラスメソッド new

3.6 課題

1. キーボードから

   ```
   1  3
   ```

   といった形で、1 行で 2 つの数を入力する時、その合計を出力するプログラムを作成して下さい。

2. 入力が 1 行に 2 つの整数値であればその合計を出力してまた入力を要求し、数値が 1 つとか 0 であれば終了するプログラムを作成して下さい。

3. 以下の形式の入力ファイル

	科目 1	科目 2	科目 3
太郎	90	80	70
花子	95	80	60
次郎	80	80	50

   があったとして、各人の平均点 (浮動小数点で)、各科目の平均点、全員、全科目の平均点を計算するプログラムを作成して下さい。科目の数・人数がこの例とは違っても実行できるようにして下さい。。

4. 以下のデータは、N 体問題の時間積分をするプログラム (*FDPS*[1] のサンプルプログラム) の出力例です。

```
9.000000e+00
16
0   0.0625   0.46109   -0.00650077   0.522333   0.259502   0.0840111   -0.148161
1   0.0625   1.26554   0.396113   0.0961424   0.876689   0.547619   -0.421987
2   0.0625   1.03187   0.488032   0.770863   0.0883532   0.29302   -0.181856
3   0.0625   -0.41411   -0.721194   -2.06982   0.0677951   -0.219516   -0.556578
4   0.0625   -1.78952   -0.427667   0.611626   -0.884486   -0.159703   -0.0363011
5   0.0625   -1.33761   -0.470841   -0.157951   -0.27971   -0.220079   -0.0240035
6   0.0625   -2.36992   -0.802938   -0.312251   -0.855476   -0.107062   0.054993
7   0.0625   1.04302   0.739473   0.0177287   0.276359   0.116819   0.366697
8   0.0625   0.821259   0.489359   -0.438738   0.759536   -0.628465   0.193383
9   0.0625   -0.183086   -0.638546   0.202722   0.25997   0.0713873   -0.420833
10   0.0625   1.3905   0.219563   -0.0124254   0.0101251   0.459834   -0.266435
11   0.0625   -0.182858   -0.686118   0.0301138   -0.733042   -0.498407   0.477569
12   0.0625   0.240498   0.472115   -0.288607   -0.141425   -0.0349425   0.229265
13   0.0625   -0.992641   0.553993   0.973543   -0.221324   -0.211653   0.422114
14   0.0625   1.18694   0.857442   -0.042799   0.423532   0.136452   0.210578
15   0.0625   -0.170962   -0.462286   0.097523   0.0936016   0.370684   0.101557
```

1 行目は時刻、2 行目は粒子数、その後に 1 行に 1 粒子で

粒子番号 質量 位置 (x,y,z 成分) 速度 (x,y,z 成分)

と 8 個の数字が並んでいます。このデータを読み込み、

[1] http://fdps.jmlab.jp/

* 重心位置
* 重心速度
* 重心速度からの相対速度の絶対値の 2 乗の平均 (恒星系力学ではこれを速度分散と呼ぶ)

を計算するプログラムを作成して下さい。

2 行目の粒子数は使わなくてもかまいませんが、使うなら、n が整数であるとして

```
n.times{|i| 処理}
```

で「処理」を n 回繰り返す機能を利用して下さい。 i には、 0, 1, 2 ... n-1 が順番にはいります。

3.7 参考

Int https://crystal-lang.org/api/0.32.1/Int.html

Float https://crystal-lang.org/api/0.32.1/Float.html

gets https://crystal-lang.org/api/0.32.1/IO.html#gets(delimiter:Char,limit:Int,chomp=false):String?-instance-method

Bool https://crystal-lang.org/api/0.32.1/Bool.html

Chapter 4

文法3 クラスとメソッド (2020/1/24)

Crystal による数値計算入門 4 回目です。前章では、配列や、配列全体に対する操作 (each や map) を使って、入力データに対て色々な処理をすることを学びました。これで、割合色々なことができるようになっていて、エクセルのデータを CSV 形式で出力したものがあれば、それから新しい表を作るとか集計するとかもここまでの知識の応用でできます。

本章では、本格的な数値計算、特に常微分方程式の数値計算に入る前に、いくつかの言語の機能、特に class と struct をみておくことにします。

4.1 class の例:基本的な 3 次元ベクトル

多くの「オブジェクト指向」の概念を取り入れた言語と同じように、Crystal ではプログラムの中で新しい型を定義することができます。数値計算でよく使う、3 次元ベクトルを表す型を作ることを考えます。そうすると、数学で普通に使うような演算ができると便利です。

- ベクトルの加減算
- ベクトルとスカラー (浮動小数点数) の乗除算
- ベクトル同士の内積・外積

つまり、 a と b が、新しく作ったベクトル型の変数だとして、

```
a+b
```

みたいに書きたいわけです。「+」をメソッドにできると、

```
a.+(b)
```

とは書けるわけですが、まだ a+b とは書けません。Crystal では (というか、大抵の言語で) a+b と書けるように、「演算子」になる文字ないし文字列を決めています。一覧は文法[1]をみていただくとして、 +,-,*,/といったところを「定義」することができます。これは a の型に対してメソッド + を定義すると、 a+b を a.+(b) というふうに言語側で解釈します、ということです。

以下は、完全な機能を与えてはいませんが加算だけ定義された 3 次元ベクトル型と、それのテスト計算の例です。

[1] https://crystal-lang.org/reference/syntax_and_semantics/operators.html

```
class Vector3
  property :x, :y, :z
  def initialize(x : Float64,  y : Float64,  z : Float64)
    @x=x; @y=y; @z=z
  end
  def +(a)
    Vector3.new(@x+a.x, @y+a.y, @z+a.z)
  end
end
x=Vector3.new(1,2,3)
p x
y=Vector3.new(1,1,1)
p y
z=x+y
p z
```

以下は実行結果です。

```
|gravity> crystal minimalvector3.cr
#<Vector3:0x7f8e3baa6f60 @x=1.0, @y=2.0, @z=3.0>
#<Vector3:0x7f8e3baa6f30 @x=1.0, @y=1.0, @z=1.0>
#<Vector3:0x7f8e3baa6f00 @x=2.0, @y=3.0, @z=4.0>
```

(1,2,3) というベクトルを変数 x に、 (1,1,1) を y にいれて、 z=x+y を計算し、それぞれを p で出力しています。

実行結果ですが、ベクトルが配列の時のように [1,2,3] とでるのではなくて、

```
#<Vector3:0x7fd5b93e1f60 @x=1.0, @y=2.0, @z=3.0>
```

とちょっと煩雑な形式で出力されているのがわかります。これは、型名である Vector3、そのアドレスのあと、このベクトルクラスが中でもっている変数名とその値が @x=1.0 といった形で出力されます。

さて、コードのほうをみていきましょう。クラスの定義は

```
class Foo
  色々
end
```

ですが、まずそれを使うところをみます。最初は

```
x=Vector3.new(1,2,3)
```

です。これは、Vector3 というクラスそのものの new というメソッドを呼んでいます。Vector3 クラスの変数に対するメソッドではないことに注意して下さい。これは前にも書いた (と思います、、、) クラスメソッドで、一番よく使うのがこの new です。 new は、こちらで定義する必要はないのですが、呼ばれると、その中でインスタンスメソッドである initialize が呼ばれます。ここで呼ばれる initialize メソッドが

```
  def initialize(x : Float64,  y : Float64,  z : Float64)
    @x=x; @y=y; @z=z
  end
```

で定義されているものです。メソッド (関数) 定義は

```
 def 名前 (引数リスト)
   色々
 end
```

という形です。インデントはみやすくする以上の意味はありません。改行は実行文の終わり等を示すのに必要な場合があります。逆に、1 行に複数の実行文を書くには

```
    @x=x; @y=y; @z=z
```

のようにセミコロンで区切ります。引数リストは

```
  (x : Float64,  y : Float64,  z : Float64)
```

のように、(名前 [: 型名], ...) と、名前のあとオプショナルに型名をつけたものを、複数ならコンマで区切って並べたものです。これを括弧の中に書きます。

この initiallize の場合は、 x, y, z の 3 つの数字を受け取り、それらはいずれも 64 ビット浮動小数点である、と宣言しているわけです。次の代入で @x といった 「@」がついた名前がでてきますが、これは「インスタンス変数」と呼ばれるもので、あるクラスの変数がその内部にもっている変数、ということになります。この場合、

```
  x=Vector3.new(1,2,3)
```

を実行すると、Vector3 クラスの変数 x が作られて、それに対して

```
  x.initialize(1,2,3)
```

が呼ばれて、内部変数 @x, @y, @z にそれぞれ引数からわたってきた 1, 2, 3 が代入されることになります。

少し細かいことですが、メソッドのほうでは x: Float64 と浮動小数点数がくるとされているのに整数が渡されますが、このような場合には、演算の場合と同じように整数を勝手に浮動小数点数に変換します。その結果、

```
 p x
```

で出力すると

```
 #<Vector3:0x7fd5b93e1f60 @x=1.0, @y=2.0, @z=3.0>
```

とでることでわかるように、 x は内部に@x=1.0, @y=2.0, @z=3.0 という値を持つことになります。

y も同様に、 代入の結果 (1,0, 1.0, 1.0) という値をもちます。

```
 z=x+y
```

は、既に書いたように x.+(y) が呼ばれ、それを定義しているのが

```
  def +(a)
    Vector3.new(@x+a.x, @y+a.y, @z+a.z)
  end
```

です。これは、自分については @x, @y, @z で値を取り出し、メソッドの引数である a については a.x, a.y, a.z という形で値を取り出して、それぞれの和を使って新しいベクトルを作ります。Crystal では、メソッドの最後の文の値がその関数の値になるので、この新しいベクトルが + 演算の結果ということになります。そういうわけで、

z には各要素の和がはいって、 (2.0,3.0,4.0) となります。

なお、class Vector3 の下にある

```
  property :x, :y, :z
```

は、上の a.x, a.y, a.z といった形で、ある型の変数の内部の変数 (インスタンス変数) を、@x といった形でメソッドの中でだけでなく、「外から」アクセスする t ことを許す、という宣言です。 :x という形は「シンボル」というもので、Crystal では色々なところで変数や関数の名前そのものではなく ":" がついた形を使います。Ruby ではこれは実装の効率の観点で意味があったのですが、Crystal では本当はいらなくて文字列でよいのでは、という気もしますが、文法としてはシンボルがあります。これは実際に Symbol というクラスがある、ということになります。

4.2 struct と本格的なベクトル型

以下は、普通に使う演算が一通り定義された Vector3 クラスです。

```
struct Vector3
  property :x, :y, :z
  def initialize(x : Float64 =0,  y : Float64 =0,  z : Float64 =0)
    @x=x; @y=y; @z=z
  end
  def +(a) Vector3.new(@x+a.x, @y+a.y, @z+a.z) end
  def -(a) Vector3.new(@x-a.x, @y-a.y, @z-a.z) end
  def -()  Vector3.new(-@x, -@y, -@z)  end
  def +()  self  end
  def *(a : Vector3) @x*a.x+ @y*a.y+ @z*a.z end    # inner product
  def *(a : Float) Vector3.new(@x*a, @y*a, @z*a)  end
  def /(a : Float) Vector3.new(@x/a, @y/a, @z/a)  end
  def cross(other)                  # outer product
    Vector3.new(@y*other.z - @z*other.y,
                @z*other.x - @x*other.z,
                @x*other.y - @y*other.x)
  end
  def sqr() self*self end
  def to_a() [@x, @y, @z] end
  macro method_missing(call)
    to_a.{{call}}
```

```
  end
end

class Array
  def to_v() Vector3.new(self[0],self[1],self[2])  end
end

struct Float
  def *(a : Vector3) a*self end
end
```

---

class ではなく struct にしているのは、Crystal の Ruby との違いの 1 つです。struct は class と全く同じように使えるのですが、この Vector3 のような、メソッドの中身が単純な計算が中心である場合にはより効率的なプログラムになります。以下、変更・追加されたメソッドをみていきます。まず、initialize ですが、引数の宣言が変わっています。

```
  def initialize(x : Float64 =0,  y : Float64 =0,  z : Float64 =0)
```

このように、 =0 といった形で値を与えることで、デフォルト値、つまり、省略した時の値を決めておくことができます。なので、 Vector.new は Vector.new(0,0,0) と、また、 Vector.new(1,1) は Vector.new(1,1,0) と同じです。引数の数が足りないと後ろから順番に省略されているとみなされます。

さて、z だけに値をいれて、x, y はデフォルト値のままにしたい、と思ったらどうすればいいでしょうか? 引数の名前を指定して値を設定する文法があり、例えば Vector.new(z:1) は Vector.new(0,0,1) と同じです。

```
  def +(a) Vector3.new(@x+a.x, @y+a.y, @z+a.z) end
```

は、

```
  def +(a)
    Vector3.new(@x+a.x, @y+a.y, @z+a.z)
  end
```

を 1 行にしただけです。Crystal ではどこに改行が必要でどこにはなくていいか、は結構ややこしいですが、メソッド定義の本体が関数呼び出し 1 つとか、引数リストの括弧があればこんなふうに 1 行にもできます。

```
   def -()  Vector3.new(-@x, -@y, -@z)  end
```

は、「単項演算子」である「-」、つまり、 a= -b といった式で現れる「-」を定義します。

```
   def +()  self  end
```

も同様です。ここでででてくる self は「自分自身」です。単項演算子 + は何もしないで自分自身を値として返すわけです。なお、これらは

```
   def +
     self
   end
```

と書くこともでき、改行があれば引数がないことを示す () を省略できます。

Vector3 同士の * は内積、Float との積はスカラー倍、除算は各要素を割る、とし、時々使うので外積を cross という名前で定義します。sqr は 2 乗で、これは * を使って定義しています。

なお、1 行の中で 「#」からあとはコメントになって、コンパイラからは無視されます。

ここで、 +(a) 等では a の型が指定されていないことに注意して下さい。中身で a.x 等を使うので、a は x,y,z が property にあるかあるいはそういうメソッドがある型である必要があります。逆にいうと、そうであれば Vector3 でなくてもかまいません。

C++ でも同じようなことはできるのですが、クラステンプレート、関数テンプレートといったものを使うかなり複雑な記法が必要になります。その辺をより簡潔にするような改良が導入されていますが、どうしても屋上屋を架す感はあり、今までよりはよいが他の言語に比べるとわかりづらいものになっているように思います。

Fortran では現在のところテンプレート自体が導入されておらず、現代的なプログラムを書く上での大きな制約になっています。

次の to_a は、Vector3 型を Array 型に変換します。a が Vector3 だとして、a.to_a とすると Array になるので、 Array に対して定義されたあらゆるメソッドが使えるようになります。

その次の

```
macro method_missing(call)
  to_a.{{call}}
end
```

は、特別な Hook と呼ばれるものの 1 つで、例えば a.map{—x— ...} というふうに、 Vector3 に定義されてないメソッドを使おうとした時のコンパイラの動作を書きます。これが

```
  to_a.{{call}}
```

になっている、ということは、「to_a で Array にしてからそのメソッドを適用せよ」ということになり、この場合は a.to_a.map{—x— ...} というコードをコンパイルすることになってめでたしめでたしとなるわけです。もちろん、Array にもないメソッドであればコンパイルエラーになります。

次の

```
class Array
  def to_v() Vector3.new(self[0],self[1],self[2])  end
end
```

では、元々 Crystal にある Array クラスに to_v という Vector3 に変換するメソッドを追加します。

最後の

```
struct Float
  def *(a : Vector3) a*self end
end
```

は、浮動小数点数 * Vector3 の演算を定義しています。 * が交換法則を、なんてことはコンパイラは知らないので、スカラー*ベクトルとベクトル*スカラーは別に (といっても前者が後者を呼ぶだけですが) 書いておく必要があります。これらのように、すでにあるクラスに自分で定義したメソッドを追加できることは、わかりやすいプログラムを書くために非常に有用です。

さて、このプログラムを例えば vector3.cr という名前でもっていたとして、色々なプログラムでベクトル型を使いたい、ということがあります。それには、もちろんそれぞれのプログラムの中でこの型の定義をすればいいですが、そうすると同じもののコピーが大量に発生します。また、他の人が使う、という時にコピペでは、修正とか改良した時に全ての人が自分のそれを使っている全てのプログラムを修正しないといけなくなります。ければならない。そのような無駄を防ぐのが、プログラムの中で他のプログラムを読込み機能です。

```
require "./vector3.cr"
Vector=Vector3
a=Vector.new(1,2,3)
b=Vector.new(1,1,1)
c=Vector.new(2,1)
d=Vector.new(y:1)

p a+b+c+d, a*b, c*d
p! a+b+c+d, a*b, c*d
```

の最初の行のように、

```
require "./vector3.cr"
```

と書くことで、そこで vector3.cr の中身を読み込んでコンパイラに渡すことができます。これを実行すると

```
|gravity> crystal testvector.cr
Vector3(@x=4.0, @y=5.0, @z=4.0)
6.0
1.0
((a + b) + c) + d # => Vector3(@x=4.0, @y=5.0, @z=4.0)
a * b             # => 6.0
c * d             # => 1.0
```

です。 p の他、 p! も便利な機能で、こちらは値だけでなくて元のプログラムの式自体も出力してくれます。

4.3 まとめ

本章では、Crystal の文法と機能について、以下を学びました。

- class, sturuct の定義のしかた
- メソッドの定義のしかた
- あるクラスの変数を新しく作って返すクラスメソッド new と、その時に使われる initialize の関係
- メソッド定義での引数の書き方、デフォルト値

- 演算子 (+とか) として使えるメソッドの定義
- クラス変数の「中の」変数へのアクセス方法
- コメント
- 引数の型を決めていない関数の書き方
- 「メソッドがない」時の処理の定義
- 既存のクラスへのメソッドの追加

4.4 課題

1. 上の、一応色々な定義したベクトルクラスについて、その全ての機能をテストして結果が正しいことを確認するプログラムを作って下さい。

4.5 参考

Struct https://crystal-lang.org/reference/syntax_and_semantics/structs.html

Chapter 5

運動方程式1 (2020/1/24)

Crystal による数値計算入門 5 回目です。

説明する文体飽きたのでここで対話形式に。

5.1 背景説明的導入

赤木 ：赤木でございます。15 年ぶりくらいの登場かしら？

学生 A： 赤木さん誰に向かって喋ってるんですか？私何故ここにいるんでしたっけ？

赤木 ：いわゆる人柱かしら。

学生 A：あの、、、それはどういう？

赤木 ：数値解析というか、運動方程式とかの数値積分入門+プログラミング入門向けのテキスト、20 年くらい前に C++ 想定で書いたんだけど、C++ 言語もこの 20 年間で随分変わったし、他にも色々な言語がでてきたし、ということで新しくしようと思ったわけ。で、書き始めたのね。だけど何か飽きてきたから、、、

学生 A：はい？

赤木 ：うん、飽きたから、後は君に書いてもらおうかと思って呼んだの。

学生 A：え、前のは紙の本で、ほら、印税半分とか書いてなかったでしたっけ？これそんなのじゃなくて単に赤木さんの趣味っていうか、ウェブページに載せるだけだからお金はいらないですよね？

赤木 ：そこはほらゴニョゴニョをゴニョゴニョして少しは、、、

学生 A：えー、最近研究費不正使用とかで色々ありますが、そういうの大丈夫ですか？

赤木 ：これはちゃんと目的にあった使用だから問題ないわ。

学生 A：じゃあ、そういうことで。

赤木 ：でね、最初にいったみたいに前の本では C++ だったんだけど、これでは Crystal にしてみようと思って。

学生 A：Crystal ってなんでしたっけ？最近よく人工知能とかで聞くのは Python とか、あと Julia とか聞いたことがありますが、、、

赤木 ：Ruby は知ってる？

学生 A：はい、Python と似てるとか、日本で開発されたとかくらいですが。

赤木 : Ruby は、名前からもわかるように Perl を念頭において、でも割合ちゃんとしたオブジェクト指向と動的なインタプリタ言語のよいところをあわせて、本格的に使えることを目指して開発された言語ね。日本ではユーザーそこそこいるんだけど、海外だと同じようなことがまあできなくはない Python のほうが普及したかな？

学生 A : はあ、で、話はなんでしたっけ？ Crystal では？

赤木 : そう、Crystal は、 Ruby とすごく近い文法なんだけど、ちゃんとコンパイラがあって実行が速いの。ほとんど同じプログラムが Ruby で実行するのに比べて 10 倍とか場合によっては 100 倍とか。

学生 A : それはすごそうですが、なんかどマイナーで世界で誰も使ってないとかでは、そんなの勉強しても就職に使えないとかないですか？

赤木 : まあ Ruby も一緒に憶えれば日本では困らないと思うわ。

学生 A : 本当ですか？

赤木 : 多分。

学生 A : (うーん、、、、大丈夫かなあ、、、という気もするけど) まあお金のはいるしやってみますが、、、

赤木 : じゃあ、まずは、コンパイラとかのインストール 1 からクラスとかの解説 4 までやってみて。で、わかんないとことかあったら教えて。

学生 A : じゃあちょっとやってみます。

5.2 一次元調和振動子

学生 A : 4 までやってみました。

赤木 : どうだった？難しい？

学生 A : 難しくないといえば嘘になりますというか、、、なんというか、だからなに？みたいな、、、

赤木 : そうよね。なので、この辺からもうちょっと楽しいことをしましょうよ、というわけ。

学生 A : 楽しいんですか？

赤木 : まあ人によっては。

学生 A : うーん、、、

赤木 : まあしばらく付き合って。というわけで、これやってみて:

一次元調和振動子

$$\frac{d^2x}{dt^2} = -kx \tag{5.1}$$

について、以下の問題に答えよ。

1 初期条件

$$x(0) = 1, \quad \frac{dx}{dt}|_{t=0} = 0 \tag{5.2}$$

からの、 $t = 2\pi$ までの数値解を、前進オイラー法、2 次ルンゲクッタ公式 (色々あるからどれでも)、リープフロッグ法のそれぞれで時間積分するプログラムを作成せよ。時間刻み、数値積分法をコマンドラインオプションとして入力できるようにせよ。

2 それぞれの方法で、 $t = 2\pi$ での誤差を、時間刻みの関数としてグラフ化し、時間の何次になっているか、何故そうなるか議論せよ。

なお、ここでは、数値計算パッケージとかソフトウェアを使うのではなく、自分でプログラムを書くこと。

赤木 ：前進オイラー法とかはそれぞれどういうのか知ってる？学生 A：多分、、、まず前進オイラー法ですね。常微分方程式の初期値問題ですから、方程式が

$$\frac{dx}{dt} = f(x,t) \tag{5.3}$$

で、 $t = 0$ で初期値 $x = x_0$ があって、あと時間刻みが、 h とします。 $t = h$ での数値解が

$$x(h) = x_0 + hf(x_0, 0) \tag{5.4}$$

もうちょっと一般に、時刻 $t = nh$ での数値解を x_n と書くことにすれば

$$x_{n+1} = x_n + hf(x_n, hn) \tag{5.5}$$

で、、、大丈夫でしょうか？

赤木 ：うーんと、、、多分。2 次ルンゲクッタは？

学生 A：同じ微分方程式として、

$$\begin{aligned} k_1 = &\quad f(x_n, hn) \\ k2 = &\quad f(x_n + k_1 h, h(n+1)) \\ x_{n+1} = &\quad x_n + \frac{k_1 + k_2}{2} h \end{aligned} \tag{5.6}$$

でしたっけ？

赤木 ：まあプログラム書いてみて上手くいけばあってるはずよね。前進オイラー法の原理は？

学生 A：教科書的には、もとの常微分方程式の解が存在していてそのテイラー展開も存在しているとすると、

$$x(t+h) = x(t) + \frac{dx}{dt}h + \frac{d^2x}{dt^2}\frac{h^2}{2!} + \frac{d^3x}{dt^3}\frac{h^3}{3!}\cdots \tag{5.7}$$

なわけで、 $f(x,t) = dx/dt$ ですから、テイラー展開の h まで、つまり 1 次の項をとったものが前進オイラー法です。

赤木 ：じゃあ、それで計算したものが、この問題の、 $t = 2\pi$ まで計算した時に近似解になるのはどうして?

学生 A：え、どうしてって、その、数値解だから近似解なのでは？

赤木 ：うーん、、、そうじゃなくて、、、じゃあまずはオイラー法だけでいいから実際にプログラム作って計算してみて？

学生 A：はあ、、、やってみます。

(一週間後)

学生 A：こんな感じです。

```
#!/usr/bin/env crystal
include Math
x=1.0; v=0.0; k=1.0
h=ARGV[0].to_f*PI
p! h
t=0
while t< PI*2 - h/2
  dv = -x*k*h
  x+= v*h
  v+= dv
  t+= h
end
p! [x, v, t]
p! [x-cos(t), v-sin(t)]
```

実行結果をいくつか:

```
|gravity> ./harmonic-euler.cr 1e-3
h # => 0.0031415926535897933
[x, v, t] # => [1.0099184201712226, 2.0875749689053084e-5, 6.283185307179335]
[x - (cos(t)), v - (sin(t))] # => [0.009918420171222575, 2.0875749940652505e-5]
|gravity> ./harmonic-euler.cr 1e-4
h # => 0.0003141592653589793
[x, v, t] # => [1.0009874475970644, 2.069126097899751e-7, 6.283185307177473]
[x - (cos(t)), v - (sin(t))] # => [0.0009874475970643726, 2.0691472301136493e-7]
|gravity> ./harmonic-euler.cr 1e-5
h # => 3.1415926535897935e-5
[x, v, t] # => [1.000098700914577, 2.0672973102662693e-9, 6.283185307165095]
[x - (cos(t)), v - (sin(t))] # => [9.870091457697683e-5, 2.0817890742914563e-9]
|gravity> ./harmonic-euler.cr 1e-6
h # => 3.141592653589793e-6
[x, v, t] # => [1.0000098696530915, 2.062700127729346e-11, 6.283185307154872]
[x - (cos(t)), v - (sin(t))] # => [9.86965309146548e-6, 4.5341698913228974e-11]
```

赤木 ：そうねえ、、まず、プログラムの解説してみて。

学生 A：では最初から。最初の

```
#!/usr/bin/env crystal
```

は、シェルスクリプトとか Ruby のスクリプトで、実行するコマンドを指定するやり方を使っています。「#!」のあとにコマンドを書きます。コマンドはパスみてくれないので、 /usr/bin/env のあとにコマンドを書くのが習慣みたいです。これ別にあってもなくてもいいんですが、実行の時に crystal と書かなくてもよくなります。

赤木 ：うーん、そうねえ、でも、繰り返して実行するなら、 スクリプトじゃないからちゃんとコンパイルしたほうがいいわ。

```
crystal build harmonic-euler.cr
```

で harmonic-euler というコンパイルした実行プログラムができるし、

```
crystal build --release harmonic-euler.cr
```

とすると最適化した速いのができるの。

学生 A：あ、そういえばそうでした。では最初の行は今後なしで。

赤木 ：はい。次の行は？

学生 A：

```
include Math
```

ですね。これは、数学関数とかを、これがないと Math.cos(x) みたいに書かないといけないみたいでちょっと面倒なのでいれてます。

赤木 ：Math は言語というか文法的にはどういうもの？

学生 A：module ですね。class と module がどう違うのかよくわかってないですが、module の中で定義したメソッドは、 class の中で include するとそのクラスのメソッドになるし、 class の中じゃなくて外だと普通の関数になるようです。

赤木 ：そうね。次は x, v, k の値ね。それはいいとしてその次は？

学生 A： ARGV は、コマンドラインで与えた引数が順番に入る配列です。これは C 言語の argv ですが、0 から引数なのは Ruby と同じです。 PI はモジュール Math で定義されている円周率で、2 π まで積分するので入力した数値に円周率かけたものを実際の h にしました。

赤木 ：了解。あと残りは？学生 A：数値積分の本体は

```
while t< PI*2
  dv = -x*k*h
  x+= v*h
  v+= dv
  t+= h
end
```

です。運動方程式と前進オイラー法をそのまま書いてます。

```
  x+= v*h
  v-= x*k*h
```

でもよさそうですが、これだと v を更新する時に、x は既に更新した値になっちゃってるのでオイラー法とは違うものになるので、あらかじめ dv を計算してます。

時刻も 0 から加算していって、 2*PI まできたら止めるとしました。あとは

```
p! [x, v, t]
p! [x-cos(t), v-sin(t)]
```

で、配列にして p! で、とすることで変数名とかも一緒にだしてみてます。cos(t), sin(t) は解析解です。周期 2π で振幅 1 の単振動なのでこれでいいはずです。これとの差で誤差をだしています。

最後に実行結果ですが、h を 10 倍づつ変えて、誤差をだすと大体 1 桁づつ小さくなるので、ちゃんとできてるんではないかとおもいが、どうでしょう？

赤木 ：えーと、そうね、一見いいように見えるんだけど、、、t の値は大丈夫？

学生 A：大丈夫というと？

赤木 ：ちゃんと 2π になってる？

学生 A：え、なってるでしょ？値が 6.2863268998329245、、、あれ？π が 3.141592 として 2 倍は 6.283184、、、あれれ？4 桁めから違いますね。ちゃんと計算すると、、、折角だから電卓に crystal 使ってみます。

```
|gravity> crystal eval "p 6.2863268998329245/(Math::PI)/2"
1.00049999999996
```

赤木 ：それ 1 ひいて逆数とったら？学生 A：えーと。

```
|gravity> crystal eval "p 1.0/(6.2863268998329245/(Math::PI)/2-1)"
2000.0000001600924
```

ですね。2000 ですね。

赤木 ：そう。だから、1 ステップ余計に計算しちゃってるの。どうしてかわかる？

学生 A：本当は 2000 ステップちょうどで終わらないといけないんですが、2000 ステップでは 2π にならないということですね。浮動小数点の足し算で誤差があるから、、、

赤木 ：そう。どう直すのがいい？

学生 A： 比較する値をちょっと小さいめに、例えば h/2 をひいておくとか、、、

赤木 ： それでもいいわね。もうひとつは、はじめから、何ステップで積分するかのほうをいれて、h のほうをあとで決めるのね。そっちでやってみて。

学生 A： times を使ってみました。n.times{ 色々 } で 「色々」を n 回繰り返します。

```
include Math
x=1.0; v=0.0; k=1.0
n=ARGV[0].to_i
h = 2*PI/n
p! h
t=0
n.times{
  dv = -x*k*h
  x+= v*h
  v+= dv
  t+= h
}
p! [x, v, t]
p! [x-cos(t), v-sin(t)]
```

実行結果をいくつか:

```
|gravity> crystal build harmonic-euler-v3.cr
|gravity> ./harmonic-euler-v3 1000
h # => 0.006283185307179587
[x, v, t] # => [1.0199349143076457, 8.432969374211775e-5, 6.283185307179473]
[x - (cos(t)), v - (sin(t))] # => [0.019934914307645712, 8.432969385516134e-5]
```

```
|gravity> ./harmonic-euler-v3 10000
h # => 0.0006283185307179586
[x, v, t] # => [1.0019758699537742, 8.284675641517022e-7, 6.28318530718043]
[x - (cos(t)), v - (sin(t))] # => [0.0019758699537741897, 8.284667206271328e-7]
|gravity> ./harmonic-euler-v3 100000
h # => 6.283185307179586e-5
[x, v, t] # => [1.0001974115707355, 8.269974924947398e-9, 6.283185307175863]
[x - (cos(t)), v - (sin(t))] # => [0.0001974115707354951, 8.273698413812141e-9]
|gravity> ./harmonic-euler-v3 1000000
h # => 6.283185307179587e-6
[x, v, t] # => [1.0000197394036099, 8.271237919989742e-11, 6.283185307155417]
[x - (cos(t)), v - (sin(t))] # => [1.9739403609886352e-5, 1.0688173528613706e-10]
```

赤木 ：t がちゃんと 2π になったようね。はい、今日のところはこの辺かしら。

5.3 課題

1. 修正版のほうのプログラムを実際に実行してみて、同じ答になることを確認して下さい。

5.4 まとめ

- crystal build でコンパイルした実行ファイル、crystal build –release で最適化もしたものができる。
- include Math で数学定数や数学関数が使えるようになる。Math は module で、他にも色々な module がある。自分で作ることもできる。
- ARGF でコマンドライン引数を参照できる。
- 浮動小数点数の演算には誤差があるので、 0.001 を 1000 回加算しても 1 ぴったりにはならないので注意。
- n.times{} で繰り返しができる。

5.5 参考資料

Math https://crystal-lang.org/api/0.32.1/Math.html

module https://crystal-lang.org/reference/syntax_and_semantics/modules.html

Chapter 6

グラフ作成とオイラー法の結果の可視化

赤木 ：前回、数字だけで、ちゃんと数値解がもっともらしい軌道になっているかどうかみてなかったから、今回グラフ書いてみましょう。

学生 A：いいですけど、何で書きます？ gnuplot とか？

赤木 ：それでもいいんだけど、プログラムの中から直接グラフ書けるほうが便利よね。

学生 A：うーん、まあ、そうですかね。データファイルからグラフでも、、、

赤木 ：まあ、今回、折角だから *GR Framework*[1] ってのを使ってみましょう。これ、C で書いてあって C からも Fortran からも使えてあと Python とか Julia からも使えるっぽいので、 Crystal からも使って見れるかなと。

学生 A： その辺よくわかってないですが、、、

赤木 ：まあ一緒にやってみましょう。GR のインストールはできたとして、環境変数 GRDIR がインストールディレクトリをさしてるとすると、

```
#include <stdio.h>
#include <gr.h>

int main(void) {
    double x[] = {0, 0.2, 0.4, 0.6, 0.8, 1.0};
    double y[] = {0.3, 0.5, 0.4, 0.2, 0.6, 0.7};
    //    gr_polyline(6, x, y);
    gr_axes(0.25, 0.25, 0, 0, 1, 1, -0.01);
    // Press any key to exit
    getc(stdin);
    return 0;
}
```

このファイルが grsample.c という名前だとして、

```
cc -I $GRDIR/include grsample.c -o grsample -L $GRDIR/lib -Wl,-rpath,$GRDIR/lib -lGR
```

[1] https://gr-framework.org/index.html

でコンパイル、これだと

```
./grsample
```

で実行できるはず。環境変数 GKS_WSTYPE を x11 にすればウィンドウがでて、pdf にすれば gks.pdf っていうファイルができるはずよ。

インストールは *C* ライブラリのインストールページ[2]から、Ubuntu なら *gr-latest-Ubuntu-x86_64.tar.gz*[3] を落としてきて、それをどっか適当なディレクトリで tar xzf とかで展開するだけ。そうすると、grってディレクトリと、その下に lib とか include とかのサブディレクトリができるの。そのディレクトリの名前を GRDIR に設定して。bash なら

```
export GRDIR=/foo/var/gr
```

ね。csh 系なら

```
setenv  GRDIR /foo/var/gr
```

になるの。それで、あとは

```
export GKS_WSTYPE=x11
```

もやってから *Getting Started*[4] にあるようになんかしてみて。

学生 A : とりあえずできました。

赤木 : 大丈夫そうね。じゃあ、同じようなことを Crystal でやってみて。

学生 A : やってみて、と言われても、、、赤木 : そうねえ、今回とりあえず、作者氏が作った Crystal から GR の関数を呼ぶためのライブラリ *gr-crystal*[5] でごまかしてみて。

学生 A : どうするんですかね？

赤木 : そこに書いてあるから読んでやってみて。

学生 A : えーと、 shard.yml になんか追加しろと。そこにある分だけでいいですかね？

赤木 : あ、最初になんかいるはず。以下でやってみて:

```
name: shards
version: 0.1.0

dependencies:
   grlib:
    github: jmakino/gr-crystal
    branch: master
```

学生 A : shards install と、、、

[2] https://gr-framework.org/c.html#installation
[3] https://gr-framework.org/downloads/gr-latest-Ubuntu-x86_64.tar.gz
[4] https://gr-framework.org/c.html#getting-started
[5] https://github.com/jmakino/gr-crystal

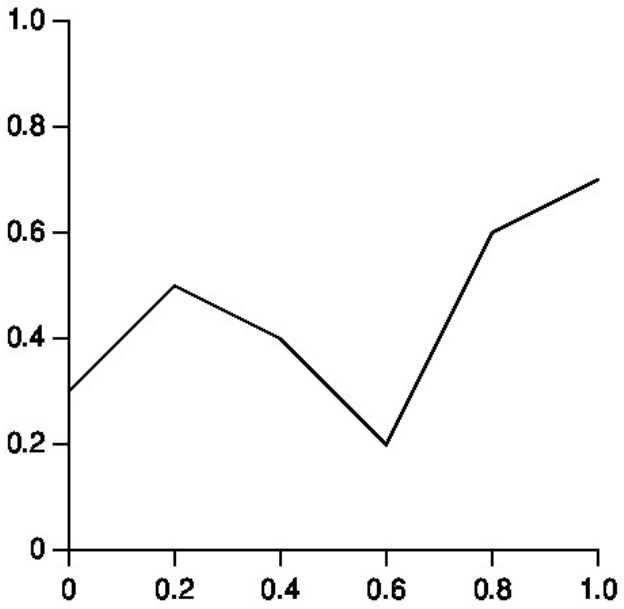

Figure 6.1: grsample (C 版) の出力結果

```
> shards install
Fetching https://github.com/jmakino/gr-crystal.git
Installing grlib (0.1.0 at master)
```

なんかしたっぽいです。

赤木 ：じゃあ、そこの README.md にある通り、

```
crystal lib/grlib/examples/grsample.cr
```

してみて。

学生 A： あ、なんかでました。

(以下心の声) 実際にやってみると色々問題が起こると思います。ありがちなのを以下に。

- エラーはでないが、なにも表示されない。リターンキー押すと終わる。
  環境変数 GKS_WSTYPE が設定されていないとこうなることがあります。

  ```
  env | grep GKS_WSTYPE
  ```

  で値を確認して、設定されてなかったら設定して下さい。

- グラフがでるけど軸の目盛りがない。

  ```
  GKS: file open error (/foo/var/baz/fonts/gksfont.dat)
  open: No such file or directory
  ```

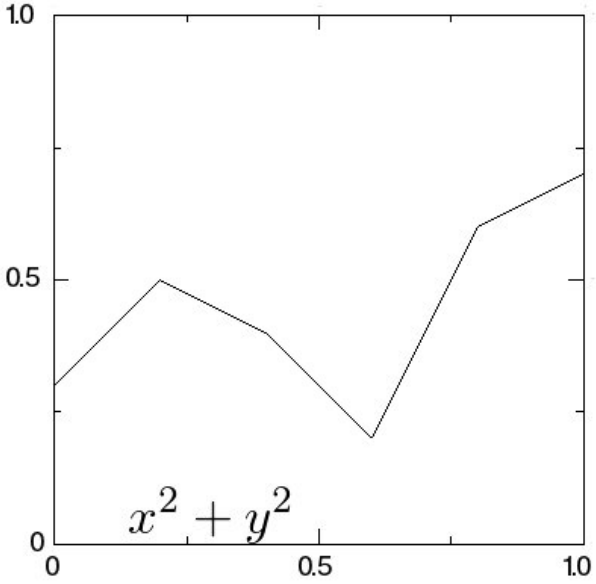

Figure 6.2: grsample (Crystal 版) の出力結果

みたいなメッセージがでる。

環境変数 GRDIR が正しく設定されてないとこうなることがあります。

```
echo $GRDIR
ls   $GRDIR/fonts/gksfont.dat
```

とかやってちゃんと正しい値かどうか確認して下さい。

- そもそも実行されない。

```
/usr/bin/ld: -lGR が見つかりません
collect2: error: ld returned 1 exit status
Error: execution of command failed with code: 1: ‘cc "${@}" -o ...
```

みたいなエラーになる。

これも環境変数 GRDIR が正しく設定されてないとこうなることがあります。

```
ls   $GRDIR/lib
```

を実行して、 libGR.so があるかどうか確認して下さい。

- グラフに $x^2 + y^2$ がない。 dvipng がなんたらというエラーメッセージがでる。

数式の処理に LaTeX 処理系、特に dvipng というコマンドを使っています。インストールされてなければ (Ubuntu の場合)

```
sudo apt install dvipng
```

でインストールして下さい。

赤木 ：じゃあ、まず、サンプルの、 grsample.cr の解説をするわね。

```
require "grlib"
GR.setwindow(0,1,0,1)
GR.box
GR.polyline([0.0, 0.2, 0.4, 0.6, 0.8, 1.0],  [0.3, 0.5, 0.4, 0.2, 0.6, 0.7])
GR.setcharheight(0.05)
GR.mathtex(0.3,0.2,"x^2+y^2")
c=gets
```

最初の

```
require "grlib"
```

は、grlib.cr を読み込んでくるやつね。 vector クラスのところでやったわね？ただ、ライブラリできているので、 ./grlib.cr ではなくて単に grlib で、コンパイラが捜すようになるわ。

その次からの GR. で始まるのが全部、 GR の機能を Crystal から呼ぶためのライブラリ関数を使ってるの。関数の形や引数の意味は GR の *Python API*[6] と同じだから、そっちみて？

学生 A：ええと、、、 gr.setwindow ってのがありますが、これですか？

赤木 ：そう。

学生 A： えーと、すみません、全然分からないんですが、、、

```
setwindow establishes a window, or rectangular subspace, of world
coordinates to be plotted. If you desire log scaling or
mirror-imaging of axes, use the SETSCALE function.
```

world coordinates ってなんですか？

赤木 ：あー、それ、 GR のベースになってる GKS っていう、コンピュータグラフィックスシステムの規格の用語ね。ISO 標準にもなってるのよ。と、そんなのはどうでもいいわね。この場合、2 次元のグラフで、横軸と縦軸の範囲を指定するの。この場合、 x 軸の範囲も y 軸の範囲も 0 から 1 まで、ということね。

次の box は、C の gr_axes を呼んでるんだけど、ちょっと工夫してあって GR のデフォルトでは下と左にしか軸を書かないのを、上と右にも書くようにしているの。あと、7 個引数があって面倒くさいから、適当にデフォルト値を設定して、それでよければそれで、としちゃってるの。

polyline は「折れ線」を引くの。x 座標の配列と y 座標の配列を引数にとるのね。C の gr_polyline は最初に点の数をいれたけど、Crystal の配列は自分の要素の数を .size でもってるから、そっちにを使うようにしたのね。

学生 A： x と y で要素数違ったらどうするんですか？

赤木 ： 小さいほう使うんじゃないかしら？やってみて。

[6] https://gr-framework.org/python-gr.html

setcharheight(0.05) は文字の高さを決めるもので、これはグラフを表示しているウィンドウとかの全体が左下 (0,0) で右下が (1,1) になる座標系での高さのはず。なので、0.05 とか小さな値なの。この座標系を規格化装置座標とかいったような気が。normalized device coordinates ね。

学生 A：最初はどうなってるんですか？

赤木 ：GR のドキュメントにどっかに書いてあるから捜して。そういうの見つけられるようになるのも大事だから。

学生 A：はい、、、

赤木 ：最後に、 mathtex(0.3,0.2,"x^2+y^2") だけど、これは、最初の 2 つが位置で、これもさっきの規格化座標のほう。あと、3 個目の文字列だけど、この関数だと LaTeX の数式モードの中にいるとしてコンパイルされて表示されるわ。

学生 A：もっと普通なのはないんですか？

赤木 ：捜してみて。なんとか text みたいなのがあるはず。

学生 A：はあ。

赤木 ：でね、 やってみて欲しいのは、前回の単振動の数値解で

- 解の軌跡を x-v 平面に表示する
- 1 周じゃなくて 5 周とか 10 周書いてみる
- ステップサイズもいくつかかえてみる

というような。

学生 A： えーと、今日ですか？

赤木 ： それはおまかせ。学生 A： それでは、、、、

(しばらくあと)

作ってみました。プログラムは

```
require "grlib"
include Math
include GR
x=1.0; v=0.0; k=1.0
n=ARGV[0].to_i
norb=ARGV[1].to_i
wsize=ARGV[2].to_f
h = 2*PI/n
p! h
t=0
setwindow(-wsize, wsize,-wsize, wsize)
box
setcharheight(0.05)
mathtex(0.5, 0.06, "x")
mathtex(0.06, 0.5, "v")

(n*norb).times{
  xp=x
  vp=v
  dv = -x*k*h
```

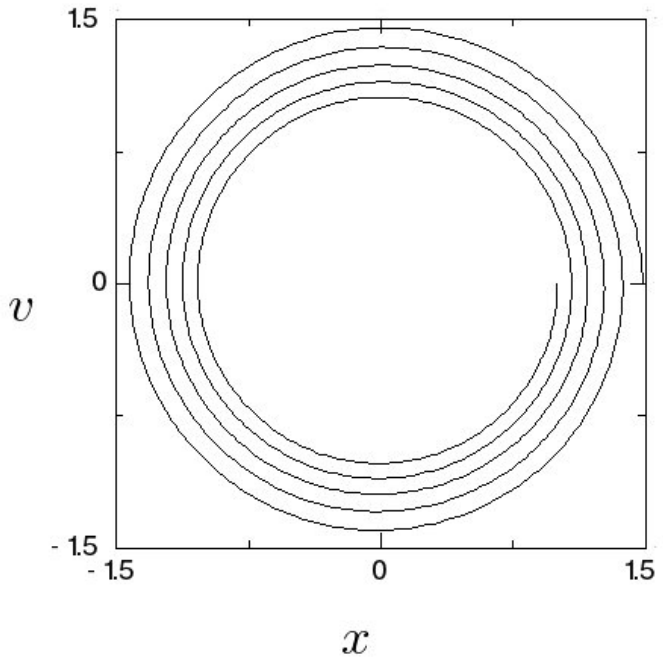

Figure 6.3: Euler 法での軌道。周期あたり 250 ステップ

```
  x+= v*h
  v+= dv
  t+= h
  GR.polyline([xp,x], [vp,v])
}
p! [x, v, t]
p! [x-cos(t), v-sin(t)]
gets
```

で、実行結果を図につけます。

赤木 ：なかなかよい感じね。

学生 A：これ、図にしてみると分かりますが円というからせんですね。一応、ステップ数増やすと段々円に近くなりますが、、、

赤木 ：そうね。何故そうなるかわかる？本当は円になるはずよね？

学生 A：えーと、そういう方法だから、とか、、、

赤木 ：それはそうなんだけど、君が何故そうなるかわかってる感じがあんまりなかったり。今の時刻の値を x_i, v_i、次の時刻の値を x_{+1}, v_{i+1}、 $k = 1$ とすると、次の時刻の値はどうなるかしら？

学生 A：えーと、

$$\begin{aligned} x_{i+1} &= x_i + hv_i \\ v_{i+1} &= v_i - hx_i \end{aligned} \tag{6.1}$$

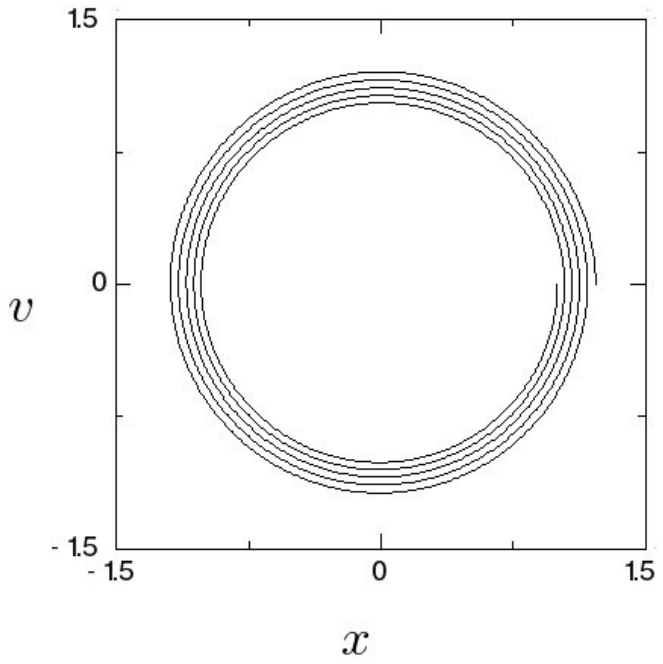

Figure 6.4: Euler 法での軌道。周期あたり 500 ステップ

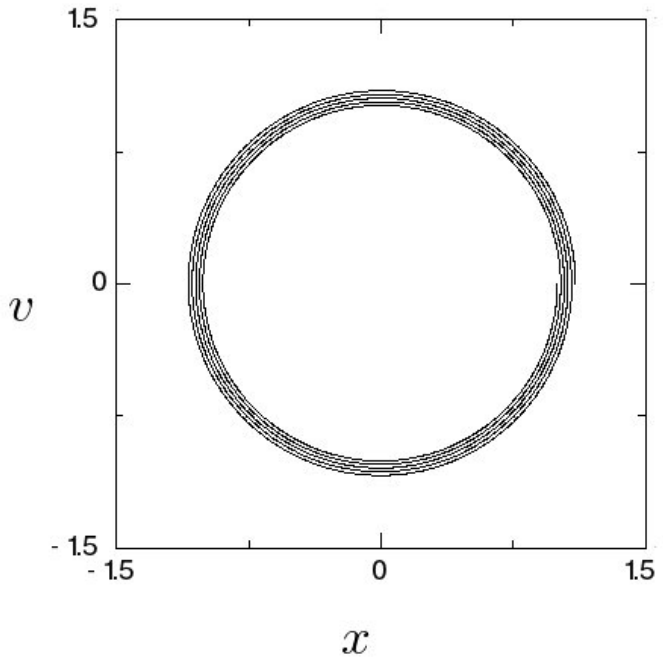

Figure 6.5: Euler 法での軌道。周期あたり 1000 ステップ

ではないかと。

赤木 ：そうね。で、これから、 i=0 の値からの一般解を計算できるでしょ？

学生 A :???

赤木 ： (x_i, v_i) をベクトル $\mathbf{x}_i$ とすれば

$$\mathbf{x}_{i+1} = \begin{pmatrix} 1 & h \\ -h & 1 \end{pmatrix} \mathbf{x}_i \tag{6.2}$$

はいいわね？

学生 A : えーと、上の式と比べて、、、多分。赤木 ：そうすると、ここにでてくる行列を A とすれば、

$$\mathbf{x}_{i+1} = A^n \mathbf{x}_0 \tag{6.3}$$

でしょ？なので、あとは A^n を計算できればよくて、これは A が $X^{-1}DX$ の形で対角化できれば、A^n が $X^{-1}D^nX$ になる、ってのは線型代数でやったわね？

学生 A : そういわれるとそういう気もします。

赤木 ：まあだから、そういうふうに機械的に手を動かせば答がでる、ってのが線型代数の偉大さなんだけど、ここはちょっと格好つけて複素数でやってみて。 $w_i = x_i + iv_i$ としたら w_{i+1} は？

学生 A : (n*10 分ほど計算) $x_i - hv_i + i(v_i + hx_i)$ だから、$(1 + ih)w_i$ です。

赤木 ：複素数を掛けることは回転+拡大・縮小と考えられる、ってのは聞いたことくらいはある?

学生 A : あるかもしれませんが、、、

赤木 ：まあやってみれば、、、 $1 + ih$ を、 a, θ が実数として、$ae^{i\theta}$ の形にしてみて。

学生 A： $a\cos\theta + ia\sin\theta = 1 + ih$ ということですね。

$$\begin{aligned} a &= \sqrt{1 + h^2} \\ \theta &= \tan^{-1} h \end{aligned} \tag{6.4}$$

でしょうか。

赤木 ：はい、よくできました。 $1 + ih$ の絶対値が $\sqrt{1 + h^2}$ で、回転角は $h = \tan\theta$ からでるわね。

学生 A : はい、そうです。

赤木 ：じゃあ、初期値を w_0 とすれば w_n は？

学生 A： $(ae^{i\theta})^n w_0 = a^n e^{in\theta} w_0$ で、回転しながら絶対値が大きくなるということですね。

赤木 ：そう。 h が小さくなれば、1 ステップでの絶対値の増えかたは $\sqrt{1 + h^2} \simeq 1 + h^2/2$ だから h^2 に比例して小さくなるけど、同じ時刻まで積分するのに必要なステップ数は $1/h$ に比例して増えるので、結局どれだけ本当の解からずれるかは h に比例するわけね。

学生 A : 角度のほうもずれませんか？

赤木 ：もちろんずれるけど、じゃあそれは h の何乗か、計算してみて。

学生 A : じゃあこれは読者の皆様の練習問題ということで、、、、

赤木 ：あらあら。まあいいわ。これから実際に計算していこうという問題はもちろん、調和振動じゃなくてもっとややこしい非線型な問題なわけだけど、数値計算法の振舞いは基本的に線型な問題に対して定義されてるの。

ここまでの話で、調和振動に前進オイラー法を適用すると、ある時刻 T まで積分した時の誤差が h に比例する、と示せたわけじゃない？

学生 A：えーと、ちゃんと証明になってましたっけ？

赤木 ：そこはちゃんと証明にしてみて。練習問題ね。

学生 A：はあ、、、

6.1 課題

1. GR、 gr-crystal をインストールして、サンプルのグラフを作ってみて下さい。
2. オイラー法のプログラムを実行して、テキストにあるような図ができることを確認して下さい。
3. 1 周期あたりのステップ数を、 100 から 1000000 くらいまで、例えば 2 倍づつ変化させて、横軸に時間刻み、縦軸に誤差をとったグラフを作って下さい。グラフは対数目盛りにするか、対数にした値をプロットして下さい。

6.2 まとめ

- GR と gr-crystal を使って、基本的なグラフを作成できる。

6.3 参考資料

GR https://gr-framework.org/index.html

gr-crystal https://github.com/jmakino/gr-crystal—gr-crystal

Chapter 7

2次と4次のルンゲクッタ法

赤木 ：オイラー法もちょっと飽きたから、もう少し賢い方法をやってみて。2 次と 4 次のルンゲクッタね。

学生 A：4 次ってどんなのですか？

赤木 ：これ。

$$\begin{aligned} x_{n+1} &= x_n + h(k_1/6 + k_2/3 + k_3/3 + k_4/6) \\ k_1 &= f(x_n, t_n) \\ k_2 &= f(x_n + hk_1/2, t_n + h/2) \\ k_3 &= f(x_n + hk_2/2, t_n + h/2) \\ k_4 &= f(x_n + hk_3, t_n + h) \end{aligned} \tag{7.1}$$

2 次ルンゲクッタも同じように書くと

$$\begin{aligned} k_1 = &\quad f(x_n, t_n) \\ k2 = &\quad f(x_n + k_1 h, t_{n+1}) \\ x_{n+1} = &\quad x_n + \frac{k_1 + k_2}{2} h \end{aligned} \tag{7.2}$$

まず 2 次のほうを、ルンゲクッタ法を関数にして、さらにそれが微分方程式を受け取ることができるように作ってみて。

学生 A： 微分方程式を受け取るってどういうことですか？赤木 ： 微分方程式を関数で表現して、その関数をルンゲクッタ法の関数に渡すわけ。学生 A： すみません、何いわれてるのかわかりません。赤木 ： じゃあちょっと順番に、微分方程式を関数にしてみて。オイラー法で。

学生 A：前の

```
1:include Math
2:x=1.0; v=0.0; k=1.0
3:n=ARGV[0].to_i
4:h = 2*PI/n
5:p! h
```

```
 6:t=0
 7:n.times{
 8:  dv = -x*k*h
 9:  x+= v*h
10:  v+= dv
11:  t+= h
12:}
13:p! [x, v, t]
14:p! [x-cos(t), v-sin(t)]
```

で、微分方程式は $dx/dt = v, \quad dv/dt = -kx$ だから、 x, v, k が入力で時間導関数が出力な関数ですね。えーと、出力が 2 つの関数って書けるんですか？

赤木 ： できるかできないかというとできるんだけど、でも、上のルンゲクッタの数式自体が、 x とか f とか k はスカラーではなくてベクトルじゃない？だから、ベクトル使うほうが自然でしょ？

学生 A : でも、今まで 3 次元ベクトルしかやってないし、ここで必要なのは 2 次元だし、2 次元問題になったら 4 次元ベクトルがいりますねよね？

赤木 ：そうね、なので、一般の次元のベクトルを、あんまり実行効率よくないかもだけど作っておいて。

学生 A : 作っておいてといわれましても、、、赤木 ：と、そうね。ここでは、Array から新しい型を作ることを考えるわね。オブジェクト指向言語でいう「継承」ってやつ。なんか難しいそうなんだけど、要するに、Array と同じ機能をもつ新しいクラスを作って、それに演算とかメソッドを追加するの。加算だけ書いてみるとこんな感じ:

```
class MathVector(T) < Array(T)
  def +(a)
    self.map_with_index{|x,k| x+a[k]}.to_mathv
  end
end

class Array(T)
  def to_mathv
    x=MathVector(T).new
    self.each{|v| x.push v}
    x
  end
end
```

最初の

```
class MathVector(T) < Array(T)
```

は、Array を継承して MathVector というクラスを作ります、というものね。Ruby だと

```
class MathVector < Array
```

なんだけど、ここは Crystal ではちょっと違うの。ここで (T) は、Array はさらに何かクラスをパラメータとしてもつ、ということで、つまり、整数の Array とか実数の Array とかを Array(Int32) と

か Array(Float64) とかで作れるわけ。逆に、必ず Array がどういうクラスの要素をもつか、を決めておかないといけないから、Ruby みたいに何もない Array を [] とかで作れなくて、Array(Int32).new とかしないといけないの。

学生 A： なんか面倒そうですね。Ruby のほうがよくないですか？

赤木 ： まあこれについては空集合をなんかうまいことやってくれないの？という気もするわね。でも、そのかわり 100 倍とかそれ以上実行速度が速いから、、、

学生 A：はあ、、、

赤木 ：でも実行速度は重要よ。1 分で終わるものが 1 時間かかったらやる気にならないもの。

学生 A：そうなんですが、、、

赤木 ：で、3 行目の

```
self.map_with_index{|x,k| x+a[k]}.to_mathv
```

が + メソッドの本体ね。map_with_index は Array のメソッドで、各要素とその添字から計算した値を要素とする Array を返すのね。なので、 Array になっちゃうから、 Array を MathVector に変換するメソッド to_mathv を Array のほうに作っておくの。これ、もうちょっと上手く書ける気がするけど、私よくわかってないから、上の話でできたの要素がない MathVector をまず作って、で、それに Array のほうの要素を 1 つづつ追加して、としてるの。

学生 A：なんかややこしいですね、、、実行遅いんじゃないですか？

赤木 ：まあ素晴らしく速いわけではないわね。速くする話はもうちょっと先でするわ。とりあえず、これで、 - と、単項の -,+ と、あとスカラーとの掛け算 *(a) を作っておいて。

あと、Float のほうにも、 MyVector との掛け算がないと掛け算に順序ができちゃうから、 vector3.cr の時と同じように Float との乗算も作っておいてね。

学生 A：はあ、、、こんなんですかね。

```
 1:class MathVector(T) < Array(T)
 2:  def +(a) self.map_with_index{|x,k| x+a[k]}.to_mathv end
 3:  def -(a) self.map_with_index{|x,k| x-a[k]}.to_mathv end
 4:  def -() self.map{|x| -x}.to_mathv  end
 5:  def +() self end
 6:  def *(a) self.map{|x| x*a}.to_mathv end
 7:end
 8:
 9:struct Float
10:  def *(a : MathVector(T))  a*self end
11:end
12:
13:class Array(T)
14:  def to_mathv
15:    x=MathVector(T).new
16:    self.each{|v| x.push v}
17:    x
18:  end
19:end
```

赤木 ：で、それ使って、まず微分方程式書いて。

学生 A：x が 長さ 2 で x, v がはいったベクトル、 k は係数のスカラーとして

```
def harmonic(x,k)
  [x[1], -k*x[0]].to_mathv
end
```

ですか？ベクトル返す必要があるから 2 要素の配列にしてから .to_mathv つけてみてます。

赤木 ：式はあってる気がするからコンパイルできれば、、、

学生 A：

```
require "./mathvector.cr"
def harmonic(x,k)
  [x[1], -k*x[0]].to_mathv
end
p! harmonic([1.0,0.5].to_mathv, 1.0)
```

でそれっぽい [0.5, -1.0] がでたから大丈夫ではないかと、、、

赤木 ：テストしておくのはよいことね。じゃあ、次はこの関数使ってオイラー法のプログラムを書き直してみて。あ、あと、刻み幅を、 1/100 から $1/10^6$ まで、プログラムの中でループ回して、刻み幅と誤差をだすようにしてみて。繰り返しは、前もでてきたけど、

```
n.times{ ...}
```

で n 回繰り返す、ってのを使いましょう。

学生 A：うーん、、、

```
 1:include Math
 2:require "./mathvector.cr"
 3:
 4:def harmonic(x,k)
 5:  [x[1], -k*x[0]].to_mathv
 6:end
 7:n=100
 8:5.times{
 9:  h=1.0/n*PI
10:  t=0.0
11:  x=[1.0,0.0].to_mathv
12:  k=1.0
13:  while t< PI*2 - h/2
14:    x += harmonic(x,k)*h
15:    t+= h
16:  end
17:  print "h= ", h, "  errors= ",x[0]-cos(t),  " ", x[1]-sin(t), "\n"
18:  n*=10
19:}
```

こんな感じでしょうか。実行すると

```
|gravity> crystal harmonic-euler-with-function.cr
h= 0.031415926535897934  errors= 0.10367468781049172 0.0022800427262427655
h= 0.0031415926535897933  errors= 0.009918420171222575 2.0875749940652505e-5
h= 0.0003141592653589793  errors= 0.0009874475970643726 2.0691472301136493e-7
h= 3.1415926535897935e-5  errors= 9.870091457697683e-5 2.0817890742914563e-9
h= 3.141592653589793e-6  errors= 9.86965309146548e-6 4.5341698913228974e-11
```

赤木 ： なかなかよい感じね。オイラー法が

```
x += harmonic(x,k)*h
```

で書けてて、もとの数学的定義に近いから、わかりやすくない？

学生 A：まあ、それはそうですね。

赤木 ：じゃあ次はこれを 2 次ルンゲクッタにしてみて。

学生 A：えーと、なので、

```
x += harmonic(x,k)*h
```

のところも、数式と同様

```
k1 = harmonic(x,k)
k2 = harmonic(x+k1*h, k)
x += h/2*(k1+k2)
```

とすればいいわけですよね？なので、

```
 1:include Math
 2:require "./mathvector.cr"
 3:
 4:def harmonic(x,k)
 5:  [x[1], -k*x[0]].to_mathv
 6:end
 7:n=100
 8:5.times{
 9:  h=1.0/n*PI
10:  t=0.0
11:  x=[1.0,0.0].to_mathv
12:  k=1.0
13:  while t< PI*2 - h/2
14:    k1 = harmonic(x,k)
15:    k2 = harmonic(x+k1*h, k)
16:    x += h/2*(k1+k2)
17:    t+= h
18:  end
19:  print "h= ", h, "  errors= ",x[0]-cos(t),  " ", x[1]-sin(t), "\n"
20:  n*=10
21:}
```

こんな感じでしょうか。実行すると

```
|gravity> crystal harmonic-rk2-with-function.cr
h= 0.031415926535897934  errors= 2.3818764601557518e-5 -0.0010332614065876578
h= 0.0031415926535897933  errors= 2.429886425403538e-8 -1.0335394959216679e-5
h= 0.0003141592653589793  errors= 2.4347190930029683e-11 -1.0335214253967046e-7
h= 3.1415926535897935e-5  errors= 7.549516567451064e-15 -1.0190453008868347e-9
h= 3.141592653589793e-6  errors= 2.6645352591003757e-15 1.4413227284901166e-11
```

おお、確かにすごく精度あがってますね。

赤木 ：そうね。で、あと、考えて欲しいのは、このオイラー法やルンゲクッタ法自体を関数にする、この関数は微分方程式の関数を引数として受け取る、ということなの。そうできると、微分方程式と数値積分法が独立に定義できるから、それぞれをベクトル型の定義と同じように別々に作っておいておけるし、一つのプログラムで数値積分法を切換える、といったこともできるじゃない。

このあとで色々そういうことをやってもらうから、この辺から入門みたいなね。

学生 A：はあ、、、えーと、関数を引数でもらうというと、Fortran ではいきなり名前で渡せましたが Crystal ではどうするんですか？

赤木 ：そうね。これ割合ややこしいところで、 Proc クラスというのを使うの。Ruby だと、コンパイルしないで実行時に実行できればいいから、関数の引数の型とか気にしないで適当に渡す文法なんだけど、Crystal では渡す関数の引数の型を指定して、他の関数とかに渡せる形にする文法があるの。例えば

```
f = -> (xx : MathVector(Float64), k : Float64){ harmonic(xx,k)}
```

とすると、 f が、ベクトル x とスカラー t を引数にとって、関数 harmonic にそれを渡してその結果を返すもの、ということになるの。但し、 f そのものは普通の関数ではないので、 f(x,t) じゃなくて f.call(x,t) というふうに使うの。

あ、だから、これ、 {} の中は別に関数 1 つしか書けないわけじゃなくて、なんでも書けるのね。それから、

```
k=1.0
f = -> (xx : MathVector(Float64) ){ harmonic(xx,k)}
```

みたいなこともできるの。

学生 A：これあとで k の値が変わったらどうなるんですか？

赤木 ：良いところに気が付いたわね。この f ができた時に k は 1 だったわけで、それを憶えてるの。なので、あとで k 変えても f は k=1 のままで動くわ。だって、変わって欲しいならちゃんと k 渡せばいいんだもの。

学生 A：なるほど。

赤木 ：まずはオイラー法を関数にしてみて。 x, t, h, それから微分方程式として f を受け取るのね。で,f は t も引数で f.call(x.t) となるとして。

学生 A：そしたら、

```
def euler(x,t,h,f)
  x+= h*f.call
  x
end
```

とかですか？

赤木 ：そうね、それはそれでいいんだけど、 t もアップデートして返すようにできない？

学生 A : 2 つ返す方法ってあるんでしたっけ？

赤木 ：Ruby だと配列にして返すとなんかできたんだけど、Crystal なのでタプル使ってみて。

```
[x.t]
```

の代わりに

```
{x,t}
```

と書くの。これは Ruby にはなくて Crystal にあるのは、タプルは作ったあとで要素に代入とかできなくて、型推論とか簡単だからかしらね。配列で書いてもできないわけじゃないんだけど、例えば

```
1:def test
2:  {[1,2,3], "abc"}
3:end
4:a,b =test
5:p! a
6:p! b
```

で、

```
|gravity> crystal testtupple.cr
a # => [1, 2, 3]
b # => "abc"
```

みたいなことができるわけ。

学生 A : そうすると、こんな感じでしょうか？

```
def euler(x,t,h,f)
  x+= h*f.call
  t+=h
  {x,t}
end
```

赤木 ：そうね、これでまったく正しくて全然問題ないんだけど、x とか t って別に代入しなくても、返す値計算すればよくて、タプルの中に式書けるから、もうちょっと簡単にならないかしら？

学生 A： 式を直接、ということですね、えーと、、、、

```
 def euler(x,t,h,f)
   {x+h*f.call, t+h}
 end
```

ですか？

赤木 ：それでコンパイル通って実行できれば多分。2 次ルンゲクッタは？

学生 A：えーと、同じようにするなら、、、

```
def rk2(x,t,h,f)
  k1 = f.call(x,t)
  k2 = f.call(x+k1*h, t+h)
  {x + h/2*(k1+k2), t+h}
end
```

でしょうか？

赤木 ：多分。じゃあ、さっきのプログラムで、積分スキームについても 2 通りやるようなプログラムにしてみて。

学生 A：(注文多いな、、、) はい。

```
 1:include Math
 2:require "./mathvector.cr"
 3:
 4:def harmonic(x,k)
 5:  [x[1], -k*x[0]].to_mathv
 6:end
 7:def euler(x,t,h,f)
 8:  x+=h*f.call(x,t)
 9:  t+=h
10:  {x,t}
11:end
12:def rk2(x,t,h,f)
13:  k1 = f.call(x,t)
14:  k2 = f.call(x+k1*h, t+h)
15:  {x + h/2*(k1+k2), t+h}
16:end
17:
18:["Euler", "RK2"].each{|name|
19:  n=100
20:  print "Result for ", name, " method\n"
21:  5.times{
22:    h=1.0/n*PI
23:    t=0.0
24:    x=[1.0,0.0].to_mathv
25:    k=1.0
26:    f = -> (xx : MathVector(Float64), t : Float64){ harmonic(xx,k)}
27:    while t < PI*2 - h/2
28:      if name == "Euler"
29:        x, t = euler(x,t,h,f)
30:      else
```

```
 31:        x, t = rk2(x,t,h,f)
 32:      end
 33:    end
 34:    print "h= ", h, "  errors= ",x[0]-cos(t),  " ", x[1]-sin(t), "\n"
 35:    n*=10
 36:  }
 37:}
```

一応動いたので大丈夫ではないかと、、、

```
|gravity> crystal harmonic-multischemes.cr
Result for Euler method
h= 0.031415926535897934  errors= 0.10367468781049172 0.0022800427262427655
h= 0.0031415926535897933  errors= 0.009918420171222575 2.0875749940652505e-5
h= 0.0003141592653589793  errors= 0.0009874475970643726 2.0691472301136493e-7
h= 3.1415926535897935e-5  errors= 9.870091457697683e-5 2.0817890742914563e-9
h= 3.141592653589793e-6  errors= 9.86965309146548e-6 4.5341698913228974e-11
Result for RK2 method
h= 0.031415926535897934  errors= 2.3818764601557518e-5 -0.0010332614065876578
h= 0.0031415926535897933  errors= 2.429886425403538e-8 -1.0335394959216679e-5
h= 0.0003141592653589793  errors= 2.4347190930029683e-11 -1.0335214253967046e-7
h= 3.1415926535897935e-5  errors= 7.549516567451064e-15 -1.0190453008868347e-9
h= 3.141592653589793e-6  errors= 2.6645352591003757e-15 1.4413227284901166e-11
```

赤木 ：よさそうね。結果をは前のと同じになってるわね。あとは、4 次のルンゲクッタと、数値だけみてても気分がでないので、グラフよね。

学生 A：4 次は、、、あ、式もらってますね。

```
  1:include Math
  2:require "./mathvector.cr"
  3:require "grlib"
  4:include GR
  5:
  6:def harmonic(x,k)
  7:  [x[1], -k*x[0]].to_mathv
  8:end
  9:def euler(x,t,h,f)
 10:  x+=h*f.call(x,t)
 11:  t+=h
 12:  {x,t}
 13:end
 14:def rk2(x,t,h,f)
 15:  k1 = f.call(x,t)
 16:  k2 = f.call(x+k1*h, t+h)
 17:  {x + h/2*(k1+k2), t+h}
 18:end
 19:
 20:def rk4(x,t,h,f)
 21:  hhalf=h/2
```

```
22:  k1 = f.call(x,t)
23:  k2 = f.call(x+k1*hhalf, t+hhalf)
24:  k3 = f.call(x+k2*hhalf, t+hhalf)
25:  k4 = f.call(x+k3*h, t+h)
26:  {x + (h/6)*(k1+k2*2+k3*2+k4), t+h}
27:end
28:
29:setwindow(3e-5, 7e-2, 1e-14, 1e-1)
30:setscale(3)
31:box(x_tick:100,y_tick:100,major_y:2,xlog: true, ylog: true)
32:
33:["Euler", "RK2", "RK4"].each{|name|
34:  n=100
35:  print "Result for ", name, " method\n"
36:  ha=Array(Float64).new
37:  ea=Array(Float64).new
38:  10.times{
39:    h=1.0/n*PI
40:    t=0.0
41:    x=[1.0,0.0].to_mathv
42:    k=1.0
43:    f = -> (xx : MathVector(Float64), t : Float64){ harmonic(xx,k)}
44:    while t < PI*2 - h/2
45:      if name == "Euler"
46:        x, t = euler(x,t,h,f)
47:      elsif name == "RK2"
48:        x, t = rk2(x,t,h,f)
49:      else
50:        x, t = rk4(x,t,h,f)
51:      end
52:    end
53:    ex = x[0]-cos(t)
54:    ey = x[1]-sin(t)
55:    print "h= ", h, "  errors= ",ex,  " ", ey, "\n"
56:#    ha.push log(h)/log(10)
57:#    ea.push log((x[0]-cos(t)).abs)/log(10)
58:    ha.push h
59:    ea.push sqrt(ex*ex+ey*ey)
60:    n*=2
61:  }
62:  p! ha
63:  p! ea
64:  polyline(ha, ea)
65:}
66:setcharheight(0.04)
67:mathtex(0.5,0.07,"\\Delta t")
68:mathtex(0.02,0.5,"\\Delta x")
69:mathtex(0.4,0.72,"\\rm Euler")
70:mathtex(0.5,0.55,"\\rm RK2")
71:mathtex(0.6,0.25,"\\rm RK4")
72:
73:c=gets
```

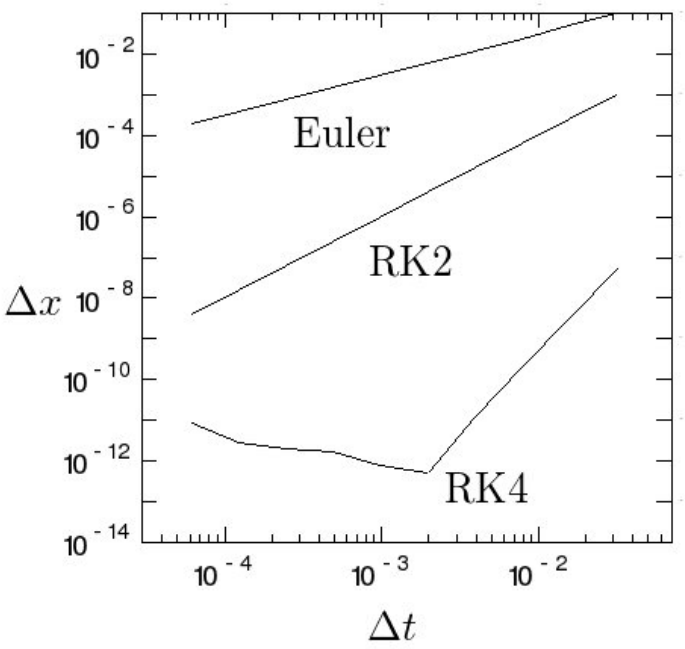

Figure 7.1: オイラー法及び 2 次、4 次のルンゲクッタの、時間刻みと精度の関係。

グラフだしてみました。

赤木 ：プログラム解説してみて。読者の皆様 (いるのかそんなの？) 向けに。

学生 A : はい。最初の 2 行は今までと同じで数学関数とベクトル型使います宣言で、次の 2 行は GR の同じような使います宣言です。

あと、6-15 行は前と同じなので省略します。20-27 が 4 次ルンゲクッタですね。2 次と同じように、ほぼ数式の通りです。

29-31 はグラフ書く準備で、setwindow(xmin, xmax, ymin, ymax) は x 軸、y 軸の範囲、 setscale は引数の 1 ビット目が x 軸、2 ビット目が y 軸を対数にするかどうかです。 box は作者が適当に作った関数で、 GR のいくつかの関数呼んで枠を書くものですね。 x_tick:100 みたいに 名前:値 という形式なのは、デフォルトの値がある引数に、呼ぶ時の順番とは無関係に値を与えるやり方みたいです。

赤木 ：あら、よく気が付いたわね。

学生 A : 著者のサンプルがそうなってたので。どんどんいくと、33 行は RK4 がはいっただけで前と同じですね。36, 37 行で、プロットするデータをいれる配列を長さ 0 で作ります。そのあと 52 行目までは前のと同じで、53, 54 行で誤差の x, y 成分を計算して、 58, 59 で時間刻みと誤差の絶対値を配列にいれます。

で、64 行で線ひいて、グラフはできたことになります。

最後に 66 行以降で軸のラベルや線がどの積分公式かのラベルつけます。これは LaTeX の数式モードになるらしいので、 \Delta t とか使えますがそのかわりなんでもイタリックになるので、ならないように \Euler としてます。LaTeX なら \rm ですが、\ となるのは文字列渡す時に \ でないと \

が実際には渡らないからです。

あと、結果ですが、傾きがオイラー、ルンゲクッタ 2、4 次でそれぞれちゃんと 1, 2, 4 くらいなので、多分ちゃんとできてるのではないかと、、、

赤木 ：テキスト出すのは別の関数もあったはずね。でもまあ、大変結構。

これ、4 次ルンゲクッタで、h 小さくしても途中から精度上がらなくなるのどうしてかわかる？

学生 A：えーと、丸め誤差でしたっけ？

赤木 ：そう。今これ、Float64 で計算しているのね。これは IEEE-754 という規格に従っていて、実数を符号 1 ビット、指数 11 ビット、仮数 52 ビットで表すの。仮数は 1 と 2 の間に 52 ビット使うから、1 の時に大体 10^{-16} の精度があるわけ。

それより精度が落ちて10^{-12} くらいになってるのは、数千ステップ計算しているからね。

これ、何故そんなに落ちるのかとかもうちょっと落ちないようにできないのかとか色々わりと大事な話があるんだけど、その辺またあとでね。

次回は、ちょっと違うタイプの時間積分法ね。お疲れさま。あ、あと、オイラー法とかルンゲクッタ法とかがどういうものかはあんまり説明してないわね。ちょっと古いけどこの辺そんなに変わってないので、著者の昔の講義資料[1]の 3 回目くらいまでみておいてね。

7.1 課題

1. 図 7.1 をだすプログラムを自分でも実行してみて下さい。コピペでもいいですが、意味を考えながら実際にプログラムを手で入力してみると意味がわかってきます。
2. 適当なパラメータで、誤差が時間のどのような関数になるか、10^4周期くらいまで積分してグラフにしてみて下さい。

7.2 まとめ

- 本章では、一般の次元のベクトル型を作成した。これは、配列を継承して、演算するメソッドを追加した。
- オイラー法の他、2 次と 4 次のルンゲクッタ法をプログラム化した。
- これらの数値積分法のメソッドに、微分方程式をあらわす関数を引数として渡す方法があることをみた。このやり方により、、微分方程式系がまったく違うものでも同じ数値積分法のメソッドを使うことができる。
- GR-Crystal で両対数のグラフを書くやり方をみた。

7.3 参考資料

システム数理 IV 講義資料 http://jun-makino.sakura.ne.jp/kougi/system_suuri4_1999/overall.html

gr-crystal https://github.com/jmakino/gr-crystal—gr-crystal

[1] http://jun-makino.sakura.ne.jp/kougi/system_suuri4_1999/overall.html

Chapter 8

シンプレクティック法

赤木 ： ここまで、線形の方程式ばっかりで、解析答があるのでそもそもなぜ数値計算するのかしらという疑問もあったと思うんだけど、もうちょっと我慢してね。線形の方程式のメリットはまさにその答がわかっている、ということで、数値積分法の特徴がわかりやすいから。

もちろん、数値積分できることの意義は、解析解がない方程式でも答を求められる、ということなんだけど、そうなるとじゃあ答は正しいのか、ということになって、その話はあとでもうちょっと詳しくするのでもうちょっと線型方程式でね。

ちょっとオイラー法に話を戻すわね。前の、オイラー法で軌道プロットする

```
require "grlib"
include Math
include GR
x=1.0; v=0.0; k=1.0
n=ARGV[0].to_i
norb=ARGV[1].to_i
wsize=ARGV[2].to_f
h = 2*PI/n
p! h
t=0
setwindow(-wsize, wsize,-wsize, wsize)
box
setcharheight(0.05)
mathtex(0.5, 0.06, "x")
mathtex(0.06, 0.5, "v")

(n*norb).times{
  xp=x
  vp=v
  dv = -x*k*h
  x+= v*h
  v+= dv
  t+= h
  polyline([xp,x], [vp,v])
}
p! [x, v, t]
```

```
p! [x-cos(t), v-sin(t)]
gets
```

だけど、

```
xp=x
vp=v
dv = -x*k*h
x+= v*h
v+= dv
t+= h
```

のところ、

```
xp = x
vp = v
x+= v*h
v+=  -x*k*h
t+= h
```

にして、n=100, 10 回転とかでグラフ書いてみて。

学生 A : えーと、これどういう意味でしょうか？ x を更新して、その更新した x を使って v を更新するわけですね？精度は同じじゃないんですか？

赤木 ：まあとりあえずやってみて。

学生 A : はい、、、え、なんかこれ間違ってないですか？図 8.1 ですが、10 回転のはずなのに線が一本で、、、

オイラー法のだと図 8.2 で、ちゃんと広がってますね。あれ？

赤木 ： これ、実はこれであってるの。 6 でやったみたいに、 行列で書いてなんかしてみて？

学生 A : えーと、

$$\begin{aligned} x_{i+1} &= x_i + hv_i \\ v_{i+1} &= v_i - hx_{i+1} \end{aligned} \tag{8.1}$$

ですが、これだと右辺にまだ x_{i+1} があるから、これを $x_i + hv_i$ で置き換えると

$$\begin{aligned} x_{i+1} &= x_i + hv_i \\ v_{i+1} &= (1-h^2)v_i - hx_i \end{aligned} \tag{8.2}$$

なので、

$$\mathbf{x}_{i+1} = \begin{pmatrix} 1 & h \\ -h & 1-h^2 \end{pmatrix} \mathbf{x}_i \tag{8.3}$$

ですね。えーと、ここからどうしましょうか？

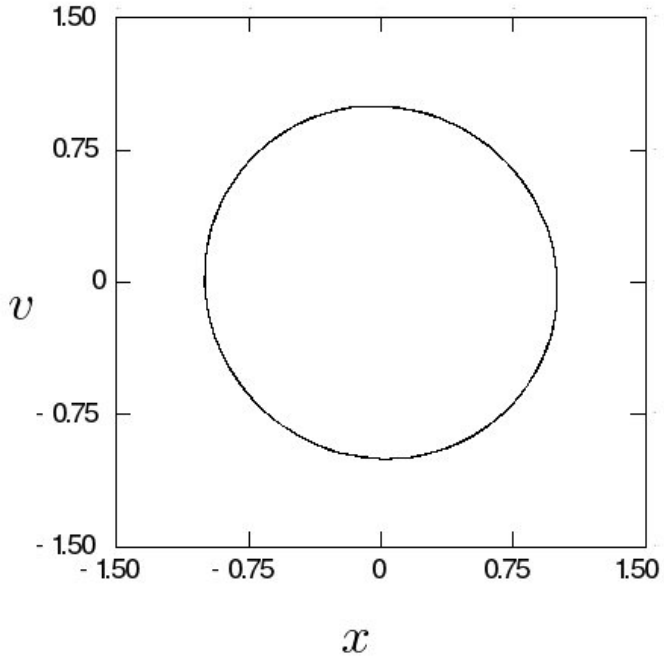

Figure 8.1: 修正版 Euler 法での軌道。周期あたり 100 ステップ

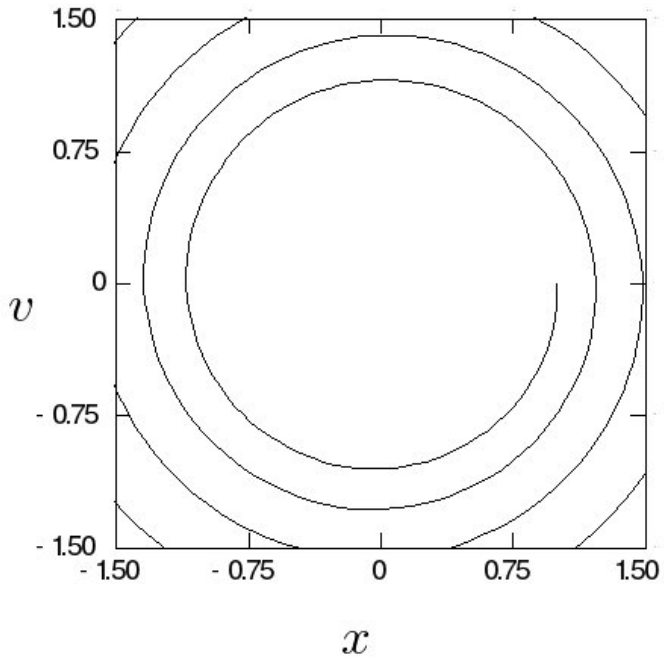

Figure 8.2: Euler 法での軌道。周期あたり 100 ステップ

赤木 ：ここからは線型代数の力に頼ることにして、まず固有方程式ね。固有方程式は？

学生 A：えーと、 $A - \lambda I$ の行列式でよかったでしたっけ？

$$(1-\lambda)(1-h^2-\lambda)+h^2=0 \tag{8.4}$$

だから、

$$\lambda^2-(2-h^2)\lambda+1=0 \tag{8.5}$$

で、

$$\lambda=(1-h^2/2)\pm\sqrt{-h^2+h^4/4} \tag{8.6}$$

赤木 ：これルートの中は負なのわかる？

学生 A：えーと、 h が 1 より小さいからですか？

赤木 ：そう。なので、

$$\lambda=(1-h^2/2)\pm i\sqrt{h^2-h^4/4} \tag{8.7}$$

となるわけね。固有値の絶対値は？

学生 A：

$$1-h^2+h^4/4+h^2-h-4/4=1 \tag{8.8}$$

なので、1 ですね。

赤木 ：そう。だから、固有ベクトルの空間で考えると、この数値積分法で調和振動子を積分することはそれぞれの要素に絶対値が 1 の共役な複素数を掛けることになってるわけ。

なので、固有ベクトルの空間では、2 つの要素がそれぞれ逆方向に円運動するわけね。元の空間ではどうなるかわかるかしら？

学生 A：楕円とかですか？

赤木 ：なんかあてずっぽう感があるけど、そうね。実数の解がいるけど、これは複素共役な値が解になってるから、2 つを足せばいいの。

この辺、実際に頑張って計算すれば確認できるけど、線型空間とか対角化とかの性質から、実際に要素とか計算しなくてもわかる、というのが大事ね。

学生 A：でもどういう楕円なんですか？

赤木 ：固有ベクトルちゃんと求めてその座標系での円が元の座標系でどうなるか出すのでも求まるけど、楕円の方程式は $ax^2+bxv+cv^2=1$ だからそれが時間積分で不変になるような a, b, c を求めてみればいいわけ。

学生 A：それは読者の演習問題ということですね？

赤木 ：まあじゃあそういうことで。ところで、この方法での解なんだけど、確かにオイラー法みたいにどんどんずれるわけではないけど、もちろんちゃんと円軌道になるわけでもないのね。そのことは、上の楕円が楕円であって円ではないことと、計算すればわかるけど楕円の円からのずれがこの方法では h に比例することからわかるわけ。

なので、もっと正確に軌道を求められる、精度のよい方法はないか、というのがここからの話ね。

学生 A：なんか 2 次とか 4 次のルンゲクッタみたいなのがないか、ということですよね？

赤木 ：そう。で、もちろん、ルンゲクッタでは駄目なのね。上の方法は

$$\begin{aligned} x_{i+1} &= x_i + hv_i \\ v_{i+1} &= v_i - hx_{i+1} \end{aligned} \tag{8.9}$$

だったわけで、 x と v で違う式、というのがわりと本質的なの。これ調和振動子で書いてるけど、一般の運動方程式だと $dv/dt = f(x,t)$ で、以下面倒なので自律系でいいのかな、 t がないやつね、での話とすると、

$$\begin{aligned} x_{i+1} &= x_i + hv_i \\ v_{i+1} &= v_i + hf(x_{i+1}) \end{aligned} \tag{8.10}$$

なわけね。で、これは x を先にアップデートして v を次にしているから、その逆の

$$\begin{aligned} v_{i+1} &= v_i + hf(x_i) \\ x_{i+1} &= x_i + hv_{i+1} \end{aligned} \tag{8.11}$$

というのも考えられるわね。

学生 A： はあ、そうですけど、、、

赤木 ：これね、誤差のでる向きが逆になるの。ちょっとちゃんと調べてみて？

学生 A：ちょっと雑ですが作ってみました:

```
require "grlib"
include Math
include GR
n=ARGV[0].to_i
norb=ARGV[1].to_i
wsize=ARGV[2].to_f
h = 2*PI/n
p! h
setwindow(0, norb,-wsize, wsize)
box
setcharheight(0.05)
mathtex(0.5, 0.06, "t/2\\pi")
mathtex(0.06, 0.5, "err")

def symplectic1(x,v,t,k,h)
  x+= v*h
  v+=  -x*k*h
  t+= h
  {x,v,t}
end
def symplectic2(x,v,t,k,h)
  v+=  -x*k*h
```

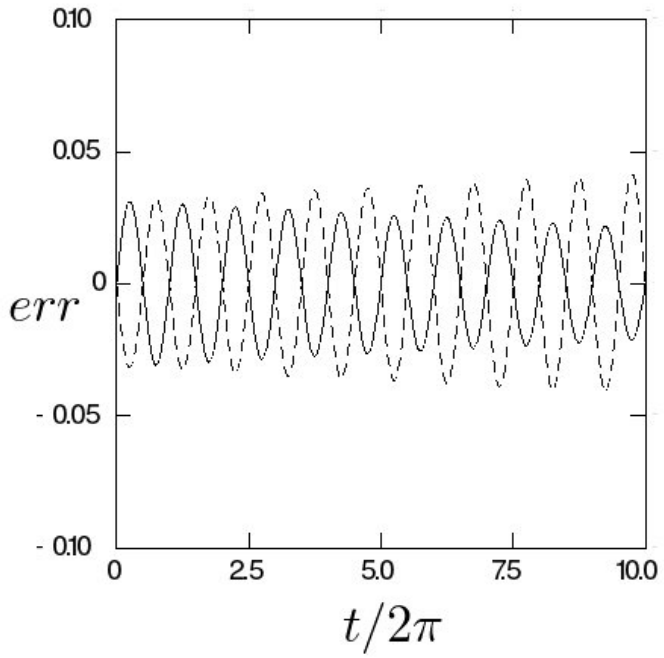

Figure 8.3: 2 つの修正版 Euler 法での x の誤差。周期あたり 100 ステップ。

```
  x+= v*h
  t+= h
  {x,v,t}
end

2.times{|i|
  x=1.0; v=0.0; k=1.0; t=0.0
  xdata=[0.0]
  ydata= [x-cos(t)]
  (n*norb).times{
    if i==0
      x,v,t=symplectic1(x,v,t,k,h)
    else
      x,v,t=symplectic2(x,v,t,k,h)
    end
    xdata.push t/(2*PI)
    ydata.push x-cos(t)
  }
  polyline(xdata, ydata)
  setlinetype(2)
}
gets
```

赤木 ：まあ段々ずれるけど、最初は綺麗に逆でしょ？だから、毎回この 2 つを繰り返す、ってのが

考えられるじゃない。どういう計算になるかしら？

学生 A：えーと、例えば

$$\begin{aligned} v_{i+1} &= v_i + hf(x_i) \\ x_{i+1} &= x_i + hv_{i+1} \end{aligned} \tag{8.12}$$

の後に

$$\begin{aligned} x_{i+2} &= x_{i+1} + hv_{i+1} \\ v_{i+2} &= v_{i+1} + hf(x_{i2}) \end{aligned} \tag{8.13}$$

なわけですね？あれ、

$$\begin{aligned} v_{i+1} &= v_i + hf(x_i) \\ x_{i+2} &= x_i + 2hv_{i+1} \\ v_{i+2} &= v_{i+1} + hf(x_{i+2}) \end{aligned} \tag{8.14}$$

になって、 x_{i+1} はでてこなくなりますね。

赤木 ： v_{i+1} も消してみて？

学生 A：

$$\begin{aligned} x_{i+2} &= x_i + 2hv_i + h^2 f(x_i) \\ v_{i+2} &= v_i + h[f(x_i) + f(x_{i+2})] \end{aligned} \tag{8.15}$$

であってます？

赤木 ：そうね。これもう i+1 なくなってるから、今 i+2 となってるのを i+1 と思うことにして、$H = 2h$ として書換えて、書換えたあとで H を h に戻してみて？

学生 A：えーと、、、、

$$\begin{aligned} x_{i+1} &= x_i + hv_i + (h^2/2) f(x_i) \\ v_{i+1} &= v_i + (h/2)[f(x_i) + f(x_{i+1})] \end{aligned} \tag{8.16}$$

であってますか？

赤木 ：これ、 x のほうは x_i のところでのテイラー展開の 2 次までとってて、v は台形公式ね。なので、あってるはず。これもさっきのグラフに追加してみて？あ、ちなみに、この積分方法はリープフロッグ (蛙飛び)って名前なの。

学生 A：適当ですが

```
require "grlib"
include Math
```

```
include GR
n=ARGV[0].to_i
norb=ARGV[1].to_i
wsize=ARGV[2].to_f
h = 2*PI/n
p! h
setwindow(0, norb,-wsize, wsize)
box
setcharheight(0.05)
mathtex(0.5, 0.06, "t/2\\pi")
mathtex(0.06, 0.5, "err")

def symplectic1(x,v,t,k,h)
  x+= v*h
  v+=  -x*k*h
  t+= h
  {x,v,t}
end
def symplectic2(x,v,t,k,h)
  v+=  -x*k*h
  x+= v*h
  t+= h
  {x,v,t}
end

def symplectic2(x,v,t,k,h)
  v+=  -x*k*h
  x+= v*h
  t+= h
  {x,v,t}
end

def leapfrog(x,v,t,k,h)
  f0 = -x*k
  x+= v*h + f0*(h*h/2)
  f1 = -x*k
  v+= (f0+f1)*(h/2)
  t+= h
  {x,v,t}
end

3.times{|i|
  x=1.0; v=0.0; k=1.0; t=0.0
  xdata=[0.0]
  ydata= [x-cos(t)]
  (n*norb).times{
    if i==0
      x,v,t=symplectic1(x,v,t,k,h)
    elsif i==1
      x,v,t=symplectic2(x,v,t,k,h)
    else
      x,v,t=leapfrog(x,v,t,k,h)
```

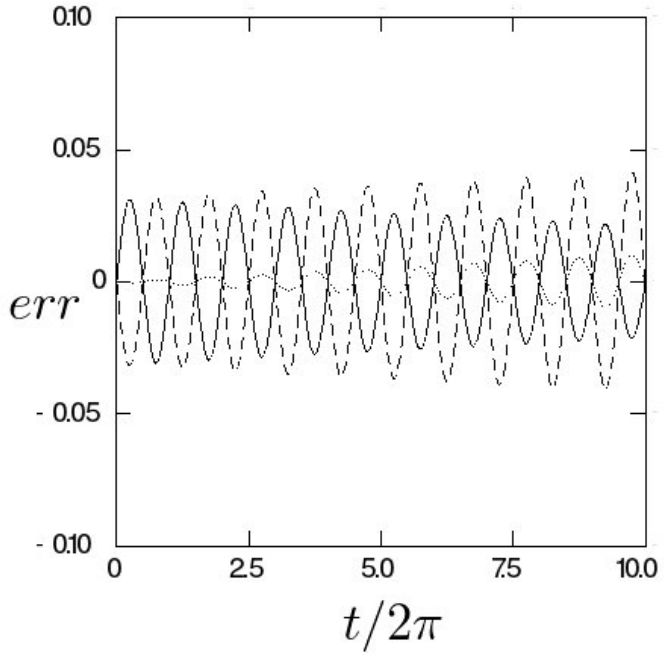

Figure 8.4: リープフロッグでの x の誤差。周期あたり 100 ステップ。

```
    end
    xdata.push t/(2*PI)
    ydata.push x-cos(t)
  }
  polyline(xdata, ydata)
  setlinetype(i+2)
}
gets
```

---

setlinetype に数字入れるとなんか線変わるみたいなので、適当な数字いれてます。点線がリードフロッグですね。時間刻みを 1/10 に小さくすると、前の 2 つの公式は誤差が大体 1/10 になってますが、リープフロッグはそれよりもずっと小さくなってるので、多分ちゃんと時間刻みの 2 乗になってるのではと。

赤木 ：そうね。これ、あとの都合もあるから、3 つの積分法ちゃんと関数にしてみて。自律型方程式用として x,v,h,f を受け取って x,v 返すものにしましょう。で、前の章と同じように誤差をだしてみて。誤差は、積分している間の x, v の誤差をベクトルとして絶対値にして、その最大値にしてみて。

学生 A：何か注文が多いですが、、、えーと、

```
require "grlib"
include Math
include GR
def symplectic1a(x,v,h,f)
  x+= v*h
```

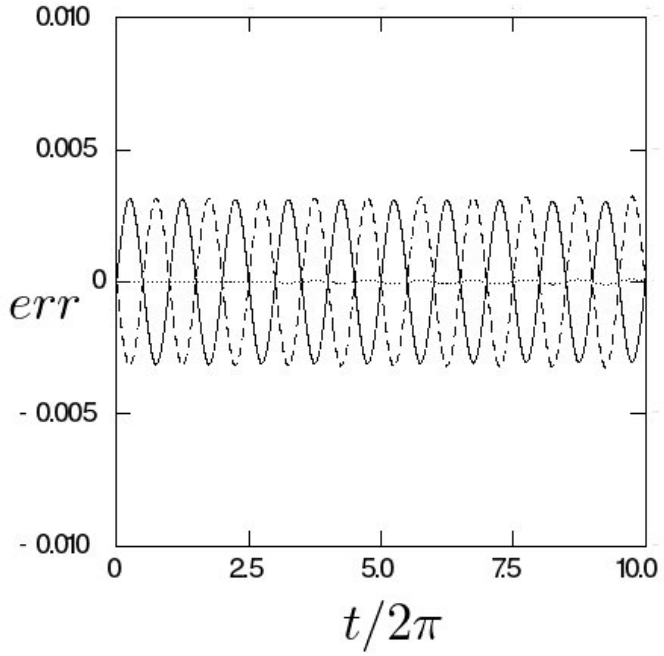

Figure 8.5: リープフロッグでの x の誤差。周期あたり 1000 ステップ。

```
  v+=  f.call(x)*h
  {x,v}
end
def symplectic1b(x,v,h,f)
  v+=  f.call(x)*h
  x+= v*h
  {x,v}
end


def leapfrog(x,v,h,f)
  f0 = f.call(x)
  x+= v*h + f0*(h*h/2)
  f1 = f.call(x)
  v+= (f0+f1)*(h/2)
  {x,v}
end

def harmonic(x,k)
  -k*x
end


setwindow(3e-5, 7e-2, 1e-8, 1e-1)
```

```
setscale(3)
box(x_tick:100,y_tick:100,major_y:2,xlog: true, ylog: true)

k=1.0
ff = -> (xx : Float64){ harmonic(xx,k)}

lt=1
["S1A", "S1B", "LF"].each{|name|
  n=100
  print "Result for ", name, " method\n"
  ha=Array(Float64).new
  ea=Array(Float64).new
  10.times{
    h=1.0/n*PI
    t=0.0
    x=1.0
    v=0.0
    emax = 0.0
    h = 2*PI/n
    p! h
    while t < 10*PI*2 - h/2
      if name == "S1A"
        x, v = symplectic1a(x,v,h,ff)
      elsif name == "S1B"
        x, v = symplectic1b(x,v,h,ff)
      else
        x, v = leapfrog(x,v,h,ff)
      end
      t+= h
      ex = x-cos(t)
      ev = v+sin(t)
      eabs=sqrt(ex*ex+ev*ev)
      emax = eabs if eabs > emax
    end
    ha.push h
    ea.push emax
    n*=2
  }
  p! ha
  p! ea
  setlinetype(lt)
  polyline(ha, ea)
  lt+=1
}
setcharheight(0.04)
mathtex(0.5,0.07,"\\Delta t")
mathtex(0.02,0.5,"\\rm Err ")
mathtex(0.4,0.55,"\\rm S1A")
mathtex(0.5,0.75,"\\rm S1B")
mathtex(0.4,0.25,"\\rm Leapfrog")

c=gets
```

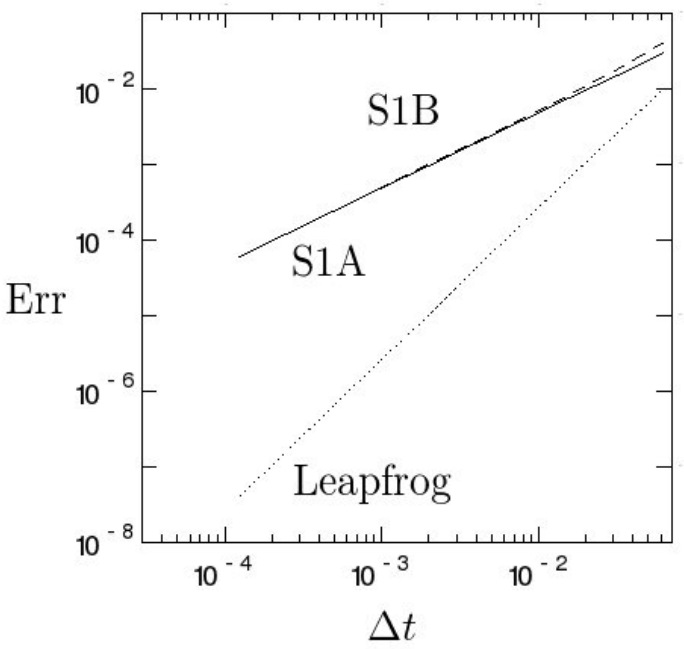

Figure 8.6: 3 つの公式での時間刻みと最大誤差。

こんなふうになりました。

赤木 ：あ、いい感じね。じゃあついでにもっと高次にいきましょう。4 次の公式を書くと、こんな感じなの。

$$\begin{aligned} S_4(h) &= L(d_1 h)L(d_2 h)L(d_1 h), \\ d_1 &= 1/(2-2^{1/3}), \quad d_2 = 1-2d_1 = -2^{1/3}/(2-2^{1/3}) \end{aligned} \tag{8.17}$$

学生 A : あのー、意味がわからないんですけど、、、、。

赤木 ：:あ、これはね、 $L(h)$ がステップ h のリープフロッグで 1 ステップだけ積分するってこと。だから、リープフロッグでまずちょっと行き過ぎて、次に逆に戻って、最後に最初と同じだけ進んで次の時刻に行くというふうになってるわけ。

これ、戻る量を進む量の $2^{1/3}$ 倍にしているのね。なぜそうするかというと、リープフロッグは x も v も時間刻みの 2 次までとってることになるから、1 ステップの誤差は 3 次なのね。だから、2 回目で戻る時の誤差は進む時の 2 倍になるから、進むのは 2 回でうちけしてくれないか、ということなわけ。こんなので上手くいくのかと思うけど、うまくいくのよ。

で、さらに 6 次の公式ってのもあって、国立天文台におられた吉田春夫先生が導いた、

$$d_1 = d_7 = 0.784513610477560$$

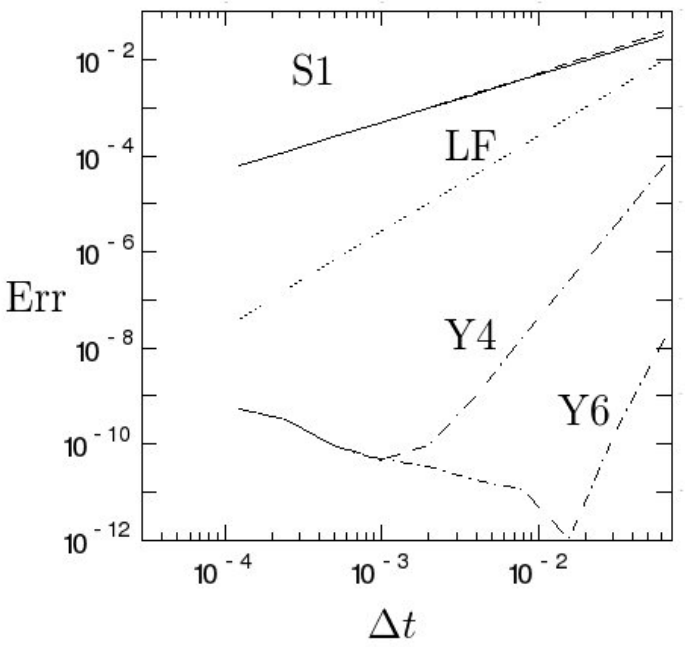

Figure 8.7: 6 次までの公式での時間刻みと最大誤差。

$$\begin{aligned} d_2 = d_6 &= 0.235573213359357 \\ d_3 = d_5 &= -1.17767998417887 \\ d_4 &= 1.31518632068391 \end{aligned} \tag{8.18}$$

で 7 回リープフロッグ呼ぶと 6 次になる公式が有名ね。4 次の関数作ってみて。

学生 A：3 回呼ぶんですね。こんなのですかね？

```
def yoshida4(x,v,h,f)
  d1 = 1.0 / (2-exp(log(2.0)/3))
  d2 = 1 - 2*d1
  x,v= leapfrog(x,v,h*d1,f)
  x,v= leapfrog(x,v,h*d2,f)
  leapfrog(x,v,h*d1,f)
end
```

もうちょっと賢く書ける気がするのと、d1 とか d2 毎回計算しているのはよくない気がしますが、、、あと 6 次も作りました。結果は 8.7 で、大丈夫な感じがします。

赤木 ：そうね、ちゃんとできた気がするので、今回この辺で。ちなみ、今回やったようなのを「シンプレクティック積分法」というのね。シンプレクティックってのはどういう意味かというと、これ解析力学やったら「正準変換」というのを習ったはずなんだけど、数値積分法が「正準変換」になっている、ということ。正準変換って、大雑把な意味としては、その変換によって物理系が変わらない、つまり、変換してから時間積分して元に戻しても、もとの系を時間積分しても、同じ解になる、ということで、それがそれが無限に時間たってもなりたつような数値積分法なわけ。まあそうするとなんかいいことがありそうでしょ？で、今回みたように実際にある、というのがわかってるわけ。

8.1 課題

1. 調和振動に対して 1 次のシンプレクティック法が保存する楕円の方程式を実際に求めて下さい。
2. 楕円の方程式が実際に数値積分でみたされていることを確認して下さい。
3. 4 次と 6 次の公式をプログラムにいれて、 8.7 と同様なグラフを自分で作って下さい。

8.2 まとめ

1. 調和振動に対して、解がずれていかないような積分公式がある。
2. 1 次の公式では、x を計算して、その x を使って v を計算とか、その逆とかをする
3. 2 次の公式は 2 つの 1 次の公式を順番に適用して構成できる
4. 4 次以上の公式は、2 次の公式の組合せで実現できる。

8.3 参考資料

システム数理 IV 講義資料 http://jun-makino.sakura.ne.jp/kougi/system_suuri4_1999/overall.html

Recent Progress in the Theory and Application of Symplectic Integrators, H. Yoshida 1993, https://link.springer.com/chapter/10.1007/978-94-011-2030-2_3

Chapter 9

ケプラー問題

赤木 ：ここまででわりと色々やったことになるわけで、

- Crystal 言語の基本。あと配列ととかベクトルクラスとか、関数を引数にとる関数とかのそこそこ高度な機能
- GR-Crystal でのお絵かき
- 数値計算法、というか常微分方程式の数値解法の基本
  - オイラー法、ルンゲクッタ法
  - シンプレクティック法

あたりをやったけど、まだ物理系としては調和振動子しか使ってなかったので色々あんまり気分がでなかったと思うの。なので、今回は、調和振動子じゃなくて、私たちの業界ではある意味基本の、ケプラー問題をやってみて。

学生 A：ケプラー問題ってなんでしょう？

赤木 ：太陽の周りを惑星が 1 個だけ回ってる、というやつ。

学生 A：これでも解析解あるんですよね？

赤木 ：そうだけど、惑星 2 つになると解析解があるとはいいがたくなるのね。で、惑星の軌道とか、太陽系が将来どうなるかとかは、ニュートン以降今でも研究対象なの。コンピュータが使えるようになってからは、ここでやってるような数値計算が研究の大事な方法になっていて、もう 30 年くらい前だけど、1980 年代に数値計算で冥王星の軌道はカオス的だと示されたとか、色々あるのよ。

学生 A：カオス的ってなんですか？

赤木 ：それはまたそれだけで 1 冊本を読まないといけないような話なので今日は詳しくはしないけど、大雑把には、すごく近い 2 つの軌道、つまり、太陽系が 2 つあって、ほとんど同じ初期条件から出版したとして、2 つの間の差が指数関数的にどんどん成長するかどうかで定義されてるわね。

学生 A：はあ。

赤木 ：まあ、まずはケプラー問題やってみましょう。前回書いた色々な積分公式を使い回せるようにして、 require して使えるようにして、それでケプラー問題解いてみて。運動方程式はわかるわね？重力定数 G は 1 で、太陽の質量は一応設定できる形だけど値は 1 でね。そうして、例えば半径 1 の円軌道だと速度 1、周期 2π で簡単だから。

学生 A： grlib.cr みたいに module とかにするんでしょうか？

赤木 ： そのほうがあとで変なことがおきないわね。そうしましょう。

学生 A：そうしたら、 integratorlib.cr という名前で

```
 1:#
 2:# integrator library
 3:#
 4:module Integrators
 5:  extend self
 6:  def euler(x,t,h,f)
 7:    x+=h*f.call(x,t)
 8:    t+=h
 9:    {x,t}
10:  end
11:  def rk2(x,t,h,f)
12:    k1 = f.call(x,t)
13:    k2 = f.call(x+k1*h, t+h)
14:    {x + h/2*(k1+k2), t+h}
15:  end
16:
17:  def rk4(x,t,h,f)
18:    hhalf=h/2
19:    k1 = f.call(x,t)
20:    k2 = f.call(x+k1*hhalf, t+hhalf)
21:    k3 = f.call(x+k2*hhalf, t+hhalf)
22:    k4 = f.call(x+k3*h, t+h)
23:    {x + (h/6)*(k1+k2*2+k3*2+k4), t+h}
24:  end
25:
26:  def symplectic1a(x,v,h,f)
27:    x+= v*h
28:    v+=  f.call(x)*h
29:    {x,v}
30:  end
31:  def symplectic1b(x,v,h,f)
32:    v+=  f.call(x)*h
33:    x+= v*h
34:    {x,v}
35:  end
36:
37:  def leapfrog(x,v,h,f)
38:    f0 = f.call(x)
39:    x+= v*h + f0*(h*h/2)
40:    f1 = f.call(x)
41:    v+= (f0+f1)*(h/2)
42:    {x,v}
43:  end
44:
45:  D1 = 1.0 / (2-exp(log(2.0)/3))
46:  D2 = 1 - 2*D1
47:  def yoshida4(x,v,h,f)
48:    x,v= leapfrog(x,v,h*D1,f)
49:    x,v= leapfrog(x,v,h*D2,f)
50:    leapfrog(x,v,h*D1,f)
```

```
51:  end
52:
53:  def  yoshida6(x,v,h,f)
54:    d = {0.784513610477560, 0.235573213359357,
55:         -1.17767998417887, 1.31518632068391};
56:    4.times{|i|x,v = leapfrog(x,v,h*d[i],f)}
57:    3.times{|i|x,v = leapfrog(x,v,h*d[2-i],f)}
58:    {x,v}
59:  end
60:end
```

でどうでしょうか？

赤木 ：私もよく知らないので解説お願い。

学生 A : 4 行目はこういう module 作りますです。module の名前は大文字で始まらないといけないらしいので、そうしてます。

赤木 ：次の extend self ってなに？

学生 A : 私もあんまりわかってないんですが、これがないと Integrators.leapfrog みたいな形でモジュールで定義した関数を呼ぶのができないみたいです。それでも include Integrators とすれば leapfrog で呼べるということらしいです。

赤木 ：何故それが extend self なのかわからないけど、やりたいことはわかったわ。

学生 A : その後はずっと関数の定義で、これはモジュールにする前と同じです。1 箇所だけ違うのは、 45-46 行で、これ j は 4 次シンプレクティック公式で使う d1, d2 ですが、モジュールの定数として、関数の外で定義してます。Crystal では大文字で始まるものは定数とのことでした。定数なので多分コンパイルの時か、あるいはプログラムの実行の最初とかで 1 度だけ評価されてくれるのではないかと、、、

赤木 ：なるほど。ケプラー問題解くプログラムのほうは？

学生 A : 作ってみました。

```
 1:require "grlib"
 2:require "./integratorlib.cr"
 3:require "./vector3.cr"
 4:include Math
 5:include GR
 6:def kepler_acceleration(x,m)
 7:  r2 = x*x
 8:  r=sqrt(r2)
 9:  mr3inv = m/(r*r2)
10:  -x*mr3inv
11:end
12:def energy(x,v,m)
13:  m*(-1.0/sqrt(x*x)+v*v/2)
14:end
15:
16:m=1.0
17:ff = -> (xx : Vector3){ kepler_acceleration(xx,m)}
18:integrator = if ARGV[3]=="LF"
19:               STDERR.print "Leap frog will be used\n"
20:               -> (xx : Vector3, vv : Vector3, h : Float64){ Integrators.leapfrog(xx,vv,h,ff)}
```

```
21:             else
22:               STDERR.print "Yoshida4 will be used\n"
23:               -> (xx : Vector3, vv : Vector3, h : Float64){ Integrators.yoshida4(xx,vv,h,ff)}
24:             end
25:
26:n=ARGV[0].to_i
27:norb=ARGV[1].to_i
28:wsize=ARGV[2].to_f
29:h = 2*PI/n
30:t=0.0
31:x= Vector3.new(1.0,0.0,0.0)
32:v= Vector3.new(0.0,1.0,0.0)
33:setwindow(-wsize, wsize,-wsize, wsize)
34:box
35:setcharheight(0.05)
36:mathtex(0.5, 0.06, "x")
37:mathtex(0.06, 0.5, "y")
38:e0 = energy(x,v,m)
39:emax = 0.0
40:while t < norb*PI*2 - h/2
41:  xp=x
42:  x, v = integrator.call(x,v,h)
43:  polyline([xp[0], x[0]], [xp[1], x[1]])
44:  t+=h
45:  emax = {(energy(x,v,m)-e0).abs, emax}.max
46:end
47:p! -emax/e0
48:c=gets
```

赤木 ：じゃあまた解説お願いできるかしら？

学生 A：はい。最初の 5 行はいいですね？r equire と include だけなので。6-11 行がケプラー問題の運動方程式というか加速度項を計算する関数です。基本的に x は前に定義した Vector3 型を想定してますが、別に 2 次元ベクトルでも内積とスカラーとの掛け算があるクラスならこの関数で計算できます。計算している式は運動方程式

$$\frac{d^2x}{dt^2} = -GM\frac{\mathbf{x}}{x^3} \tag{9.1}$$

の右辺です。 $G = 1$ でいいとのことだったのでそこはさぼってます。

赤木 ：16 行は m を 1 にして、17 行は前の調和振動子の時と同じで関数を関数に渡せるようにしてるのね。

えーと、その次の 18 から 24 行目はなにこれ？

学生 A：すみません、ちょっと変わったことしてみたくて。やってるのは、integrator に、その上の ff と同じように時間積分公式のほうをいれるのですが、ARGV[3]、つまりコマンドラインで与える 4 個目のパラメータでその値を変えるので if ...else ... end になってます。if 文は真になるほうの中身の最後の式が値になるそうなので、こんなふうに画面に文字とか書いてから式書いたらその最後の式が値になって integrator に入るわけです。

赤木 ：ちょっと気持ち悪いけど、これでコンパイルも実行もできたのね？

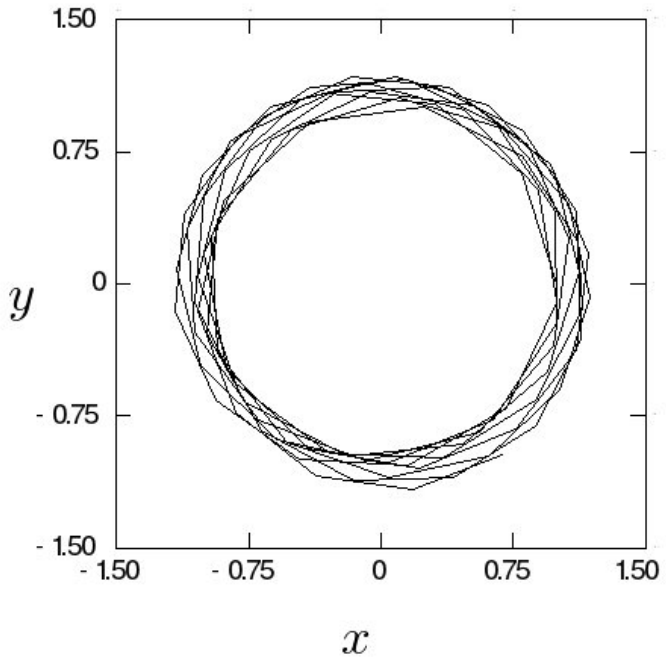

Figure 9.1: リープフロッグ、1 周期 10 ステップ、10 周期のケプラー問題円軌道の解

学生 A : はい。どんどんいくと、 26-30 行は前にオイラー法で軌道書いたプログラムと同じで、周期あたりのステップとか何周期かとか、グラフの軸の範囲とかです。

31、32 は初期条件の設定ですね。ここで初めてが Vector3 ということになります。33-37 はグラフの枠とかラベルで、前と同じです。

38-39 は、積分精度がよくわからなかったので、エネルギーを計算するようにして初期のエネルギーをだしてあと最大誤差を初期化してます。

40-46 行が実際の時間積分で、ほぼ integrator を call するだけですね。あとは前の x の座標から線ひいて、初期からのエネルギーの変化を更新して、というくらいです。

赤木 ：なるほどね、積分公式を選択する if 文がこっちにはないから、わかりやすいといえばわかりやすいわね。

プログラムって、 while とか if とかが何重にも重なる (ネストする、というのね) と、それだけで何やってるかわからなくなるので、なるべくそうならないほうがいいの。

と、うんちくはいいとして結果は？

学生 A : 1 周期 30 ステップ、10 周期、リープフロッグでだしたのが図 9.1 です。ケプラー問題でも、リープフロッグでは誤差が一方的にたまったりしないことがわかります。エネルギー誤差は 2.5% とでてました。

赤木 ：積分公式変えると？

学生 A : 4 次シンプレクティックだと図 9.2 です。エネルギー誤差は 0.9% で、すごくよくなってはいないですがだいぶよいです。ちなみに、100 ステップにするとリープフロッグで 3.9e-6、4 次シンプレクティックでは 2.6e-10 となって、なんか精度よすぎるんですが、、、リープフロッグでは 2 桁のはずなのが 4 桁、4 次シンプレクティックでも 4 桁のはずが 7 桁近くあがってます。

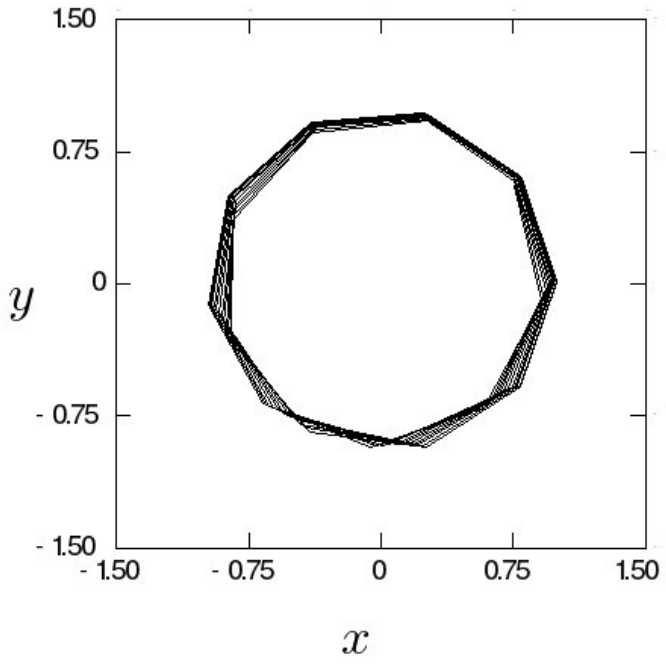

Figure 9.2: 4 次シンプレクティック、1 周期 10 ステップ、10 周期のケプラー問題円軌道の解

赤木 ：ああ、これはそうなるのよ。円軌道だけ特別なの。これ昔から知られていると思うんだけどちゃんと解説してあるのはあんまりみたことない気がするわ。楕円軌道もできるようにしてみて。最後のコマンドラインパラメータに離心率追加して、遠点から計算始めてみて。

学生 A：軌道長半径は 1 のままとすると、遠点では太陽からの距離は 1+e ということですね。速度は距離に直交して、エネルギーは -0.5 なので、速度を v として

$$-\frac{1}{1+e}+\frac{v^2}{2}=-\frac{1}{2} \tag{9.2}$$

なので

$$v^2=-1+\frac{2}{1+e}=-\frac{1-e}{1+e} \tag{9.3}$$

となって、

```
ecc=ARGV[4].to_f
x= Vector3.new(1.0+ecc,0.0,0.0)
v= Vector3.new(0.0,sqrt((1-ecc)/(1+ecc)),0.0)
```

でいいんじゃないかと。例えば $e=0.5$ でやってみると、、、あ、1 周 10 ステップでは破綻しているので 20 でやるとリープフロッグで 17%、200 にすると 0.26%ですね。離心率が 0 でなければちゃんと積分公式の次数でエネルギー保存するんですね。4 次公式もちゃんと 4 桁よくなってます。

```
|gravity> keplerplot2 20 10 2 LF 0.5 </dev/null
(-emax) / e0 # => 0.17515575170856668
Leap frog will be used
|gravity> keplerplot2 200 10 2 LF 0.5 </dev/null
(-emax) / e0 # => 0.002618881439332199
Leap frog will be used
|gravity> keplerplot2 20 10 2 Y4 0.5 </dev/null
(-emax) / e0 # => 0.2662227787182232
Yoshida4 will be used
|gravity> keplerplot2 200 10 2 Y4 0.5 </dev/null
(-emax) / e0 # => 2.01715287211357e-5
Yoshida4 will be used
```

赤木 ：離心率 0.9 とか 0.99 とか 0.999 にしたらどうなるかしら？

学生 A：えーと、、離心率とステップでループ回すようにプログラム変えますか？

赤木 ：それでもいいんだけど、今まであんまりこういう話をしてないけど、1 つのパラメータを入力して積分するプログラムと、ループ回すのと、時間積分するところは同じなのに違うものができるのはあんまりよくないじゃない。

学生 A：そうしたらどうするのがいいですかね？

赤木 ：UNIX 風の考え方だと、個々のプログラムはなるべく機能が少ないものとか単一機能にして、それを組合せて、みたいな感じね。

学生 A：でも今のプログラム画面にグラフとかでますし、、、

赤木 ：じゃあまずはその辺まで実行時に設定できるようにしておいて。

学生 A：そうすると 5 個コマンドラインオプションつけるわけですね。自分でもどれがなにか段々わからなくなってきてて、、、

赤木 ：そうねえ、、、ちゃんとドキュメント書くとか、ヘルプメッセージをだすようにするとかかしら。

学生 A：ヘルプメッセージってどうやってだすんですか？

赤木 ：C とかの普通のプログラムだと、単にプログラムの中でそのままとかだけど、、、

学生 A：あと、例えば、コマンド実行の時に -n 10 とするとステップ数の変数に 10 いれるとかはうまい関数とかあるのでしょうか？

赤木 ：うーん、うまいかどうか難しいけど、作者氏が作ったの使ってみる？

学生 A：もっと普通ののほうがよくないですか？

赤木 ：まあそうかもだけど、わりと便利よ。一応こっちは Crystal のパッケージ管理ツールでいれられるから、Crystal のプログラムのソースファイルがあるディレクトリに、 shard.yml っていうファイル作って、中身を

```
name: shards
version: 0.1.0

dependencies:
  clop:
    github: jmakino/clop-crystal
    branch: master
dependencies:
```

```
grlib:
 github: jmakino/gr-crystal
 branch: master
```

にして、

```
shards install
```

ってコマンド実行してみて。あ、インターネットにつながってないと駄目よ。github と書いてあることからわかると思うけど、これ github にアクセスするから。

学生 A：えーと、すみません、github ってなんですか？

赤木 ：あ、、知らないか。そうよね。これは、git っていうソフトウェアのバージョン管理ツールがあって、それを使って複数の人が共同開発するようなプロジェクト用に、その git のサーバーを立ててるサイトなの。お金払うと非公開のプロジェクト、ただでも公開のプロジェクトはつくれるから、ソフトウェアを公開するには便利なの。あ、最近はただでも非公開のもつくれたかも。

要するに、そこにソフトウェアあげておくと、他の人がコマンド 1 つでダウンロードとか、最新版への更新とかができるわけ。バージョン管理というのは、データベースにして、昔の状態も残してある、ということね。だから、なんかの理由で古いのを使いたいとか、間違ってバグありのをいれちゃったとかにも対応できると。あと、他の人がこう修正したらと提案するとかこういう問題があったというとかもやりやすくて記録が残るしかけがあるの。

学生 A：それって、すごいソフトウェア書けるプロな人が使うもんじゃないですか？

赤木 ：あ、そうでもないわ。あなたが色々プログラムが書く時にも、あと卒業論文とかも、バージョン管理して、そのデータベースを自分の手元の機械のほかにあちこちに置くのは絶対しないといけないことよ。

まあ学生で 1-2 年の間だと運がよければあんまり困ったことにならないかもしれないけど、特に卒業論文書いているとそのデータがあるノート PC とかが壊れる、ということは結構あるから。

あと、1 箇所だと日本だと地震とか津波で計算機ごとデータが消えるとかだって絶対におこらないとはいえないわ。そうじゃなくても、計算機が壊れることはあるし。私は基本的に全部の文章とかプログラムとかを、

- 普段使ってるノート PC
- 家の計算機
- 大学の計算機

の 3 箇所にはおいていて、家のも大学のも raid1 っていう 2 つのハードディスクに同じものを書く仕掛けにして、さらに git でバージョン管理して git のデータも 3 箇所においてるわ。

学生 A：それちょっと神経質すぎるというか、パラノイアックじゃないですか？

赤木 ：まあ年とるとその間には色々あるのよ。これはこれで重要なんだけどちょっと脱線したわね。そういうわけで shards install してみて。

学生 A： はあ。 shards install でリターン、と。

```
Fetching https://github.com/jmakino/clop-crystal.git
Installing clop (0 at master)
```

こんなのでました。

赤木 ：そしたら、 lib/clop/src の下に clop.cr と clopsample.cr ができてるはずね。サンプルのほう実行してみて。

学生 A：

```
|gravity> crystal lib/clop/src/clopsample.cr
Option -n(--number_of_particles) need some value to be given
Try -h for the list of options
```

なんかたりませんといわれてるみたですが、、

赤木 ： あとヘルプで -h でなんかでると書いてあるわね。それつけてみて。あ、クリスタルコンパイラのオプションと区別しないといけないから、 – を最初につけてね。

学生 A：

```
|gravity> crystal lib/clop/src/clopsample.cr -- -h
Softening length for Plummer softening, where rs2=r**2+eps**2 is used in place of r**2.
Number of particles
Extra diagnostics
Shifts center of mass velocity
Name of the outputfile
Command line option parser sample code
  -s        --softening_length                   0.0   -n     --number_of_particles          int     n
```

あ、なんかでますね。

赤木 ： 見方はなんとなくわかるかしら？ 1 文字の短いオプションと、それを同じことになる長いオプションがあって、データ型 (あれば) とデフォルトの値と説明がちょっとでるのね。

学生 A：はい。 -n になんか値いれたらどうなるんでしょうか？

赤木 ：まあやってみれば。

学生 A：

```
|gravity> crystal lib/clop/src/clopsample.cr -- -n 10
options # => #<CLOP:0x7f3a4d158680
 @eps=0.0,
 @n_particles=10,
 @output_file_name="",
 @vcom=[3.0, 4.0, 5.0],
 @xdiag=false>
a.class # => Float64
a # => 0.0
```

なんかでますね。プログラムは、、、なんか一杯かいてありますがこれ文字列変数に値いれているだけで、最後が

```
clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
options=CLOP.new(optionstr,ARGV)
pp! options

a= options.eps
pp! a.class
pp! a
```

ですね。 option っていう変数の中に n_particle というメンバー変数があって、それでに 10 がはいってますね。

赤木 ：そうね。この clop ライブラリ、クラスを普通に定義するんじゃなくて、テキストで

```
Short name: -n
Long name:--number_of_particles
Value type:int
Default value:none
Variable name:n_particles
Description:Number of particles
Long description:
  Number of particles in an N-body snapshot.
```

とか色々書いてるだけで、 CLOP というクラスができて、その中に n_particles という変数ができて、コマンドラインから設定できるわけ。この記述自体は、このオプションの仕様っていうか、どういうものかを人間にもわかるように書いてあるんだけど、それからクラスとかのコードをコンパイル時に作るのね。

学生 A：どうやってるんですかそれ？

赤木 ：まあその、結構邪道な感じのことしてるわねこれ、、、雑に説明すると、このプログラムのソースコード自体の、この文字列定義が終わるところまでを切り出して、それをその文字列を読んで Crystal のクラス定義のソースコードを生成するして出力する関数とその呼び出しを付け加えて、それを Crystal のプログラムとして実行して、その出力結果をコンパイラに渡すのね。それをやってるのが

```
clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
```

で、これは普通の関数じゃなくて、コンパイル時に呼び出される関数みたいなもので、 Crystal ではマクロ (macro)っていうものなの。C や C++ でいうところのプリプロセッサ指示行みたいなものなんだけど、はるかに高度なことができるわ。

学生 A：なんかもうちょっと簡単にできそうな気も、、、

赤木 ：まあ、なるべく Crystal 自体の機能を使って作ってみたかったんじゃないかしら。 コンパイラがもう一度コンパイラ呼び出して実行までするので、ちょっと遅いのよねこれ、、、それと、作者氏もこんなややこしい macro 書くの初めてだから、多分もっとスマートなやり方あるわね。

学生 A：で、これ使って書くんですか？やってみますが、、、オプション 1 つにつきこの 7 個全部書くんですか？

赤木 ：書かないでも動くものあるかもしれないけど、全部書いて。まあ大した手間じゃないでしょ？まあこの辺はデザインした人の思想で、ちゃんと他人にもわかるように書いてね、ということなのよ。

学生 A：はあ、、、とりあえずやってみます。

```
 1:require "grlib"
 2:require "./integratorlib.cr"
 3:require "./vector3.cr"
 4:require "clop"
 5:include Math
 6:include GR
 7:
 8:optionstr= <<-END
 9:  Description: Test integrator for Kepker problem
10:  Long description:
11:    Test integrator for Kepker problem
12:    (c) 2020, Jun Makino
13:
14:  Short name:           -n
15:  Long name:  --nsteps
16:  Value type:int
17:  Default value: 20
18:  Variable name: n
19:  Description:Number of steps per orbit
20:  Long description:     Number of steps per orbit
21:
22:  Short name: -o
23:  Long name:  --norbits
24:  Value type:int
25:  Default value:1
26:  Variable name:norb
27:  Description:Number of orbits
28:  Long description:     Number of orbits
29:
30:  Short name:-w
31:  Long name:  --window-size
32:  Value type:  float
33:  Variable name:wsize
34:  Default value:1
35:  Description:Window size for plotting
36:  Long description:
37:    Window size for plotting orbit. Window is [-wsize, wsize] for both of
38:    x and y coordinates
39:
40:  Short name:-e
41:  Long name:--ecc
42:  Value type:float
43:  Default value:0.0
44:  Variable name:ecc
45:  Description:Initial eccentricity of the orbit
46:  Long description:     Initial eccentricity of the orbit
47:
48:  Short name:-g
49:  Long name:--graphic-output
50:  Value type:        bool
51:  Variable name:gout
52:  Description:
```

```
 53:    whether or not create graphic output (default:no)
 54:  Long description:
 55:    whether or not create graphic output (default:no)
 56:
 57:  Short name:-t
 58:  Long name:--integrator-type
 59:  Value type:        string
 60:  Variable name:itype
 61:  Default value:LF
 62:  Description:
 63:    integrator scheme. LF:leapflog, Y4:Yosida4
 64:  Long description:
 65:    integrator scheme. LF:leapflog, Y4:Yosida4
 66:END
 67:
 68:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
 69:options=CLOP.new(optionstr,ARGV)
 70:
 71:def kepler_acceleration(x,m)
 72:  r2 = x*x
 73:  r=sqrt(r2)
 74:  mr3inv = m/(r*r2)
 75:  -x*mr3inv
 76:end
 77:def energy(x,v,m)
 78:  m*(-1.0/sqrt(x*x)+v*v/2)
 79:end
 80:
 81:m=1.0
 82:ff = -> (xx : Vector3){ kepler_acceleration(xx,m)}
 83:integrator = if options.itype=="LF"
 84:               STDERR.print "Leap frog will be used\n"
 85:               -> (xx : Vector3, vv : Vector3, h : Float64)
 86:               { Integrators.leapfrog(xx,vv,h,ff)}
 87:             else
 88:               STDERR.print "Yoshida4 will be used\n"
 89:               -> (xx : Vector3, vv : Vector3, h : Float64)
 90:               { Integrators.yoshida4(xx,vv,h,ff)}
 91:             end
 92:
 93:h = 2*PI/options.n
 94:t=0.0
 95:x= Vector3.new(1.0+options.ecc,0.0,0.0)
 96:v= Vector3.new(0.0,sqrt((1-options.ecc)/(1+options.ecc)),0.0)
 97:if options.gout
 98:  wsize=options.wsize
 99:  setwindow(-wsize, wsize,-wsize, wsize)
100:  box
101:  setcharheight(0.05)
102:  mathtex(0.5, 0.06, "x")
103:  mathtex(0.06, 0.5, "y")
104:end
```

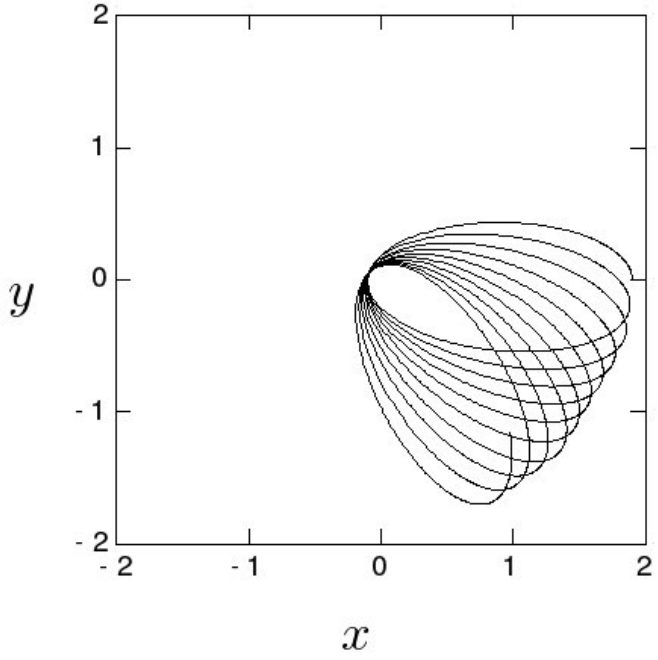

Figure 9.3: 離心率 0.9、10 周期のケプラー問題をリープフロッグで 1 周期 300 ステップで数値積分した結果。

```
105:e0 = energy(x,v,m)
106:emax = 0.0
107:while t < options.norb*PI*2 - h/2
108:  xp=x
109:  x, v = integrator.call(x,v,h)
110:  polyline([xp[0], x[0]], [xp[1], x[1]]) if options.gout
111:  t+=h
112:  emax = {(energy(x,v,m)-e0).abs, emax}.max
113:end
114:p! -emax/e0
115:c=gets if options.gout
116:
```

こんな感じですかね。一応動くことは確認してます。前のプログラム (keplerplot2.cr) と同じパラメータなら同じ答です。離心率がある計算結果をは、例えば離心率 0.9 で図 9.3 とかその次のみたいな感じです。

離心率大きいとすごくステップ数ふやしてもなんか結果怪しいというか、近点が移動しますね、、、

赤木 ：まあそういうものよ。その辺の対応は次回かその次でね。プログラムのほうは、8 行目から 65 行目までがオプションの記述ということね。Description と Long description が同じなところに人間というのはこういうふうにさぼるものかという感慨があるわね。

学生 A：英語苦手なんですよ、、、プログラムはあとほとんど前のと同じで、n とかが options.n に変わるくらいです。あと、 options.gout を使って、グラフだすかどうか区別してます。

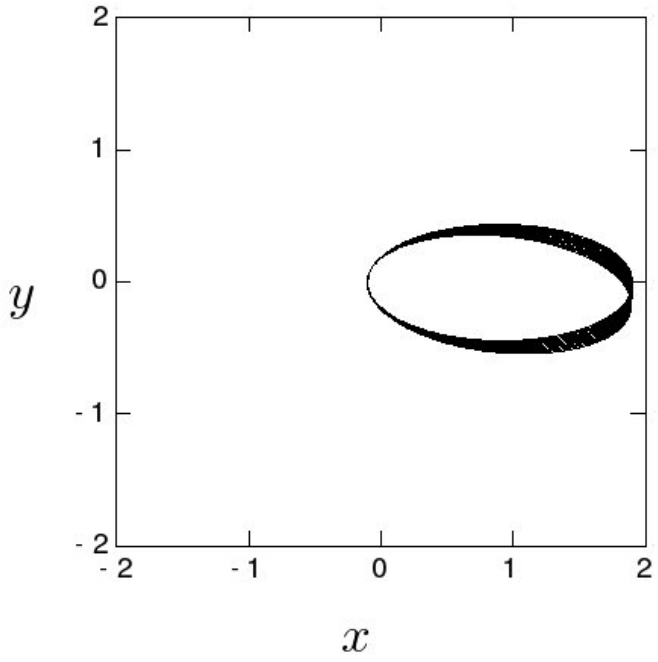

Figure 9.4: 図 9.3 と同じだが 1 周期 1000 ステップ。

赤木 ：そうすると、グラフ書くプログラムはこっちを呼んで、繰り返すようなのを作ってね。あ、何度も実行するわけだから、 crystal build でコンパイルして実行ファイル作ってね。

あ、そういえば、 make って知ってる？

学生 A : えーと、 make なんとか っていれるとそのなんとかをコンパイルしてくれるとかいうものでしたっけ？ Makefile というものを書くらしいと、、、

赤木 ： 使ったことはない？

学生 A : オープンソースのプログラムのインストールとかはしたことありますが、自分のプログラムに使ったことはないです。

赤木 ：じゃあ、とりあえず Makefile を以下の 2 行で作ってみて

```
  %  : %.cr

crystal build  $<
```

あ、これ、2 行目の crystal の前はタブ 1 つじゃないと駄目で、空白文字では駄目なので注意してね。

学生 A : 作りました。

赤木 ： これ、さっきの、CLOP がはいったプログラムの名前は？

学生 A : keplerplot3.cr です。

赤木 ： じゃあ、make -n keplerplot3 といれてみて？

学生 A : はい

```
crystal build  keplerplot3.cr
```

とでます。

赤木 ：これは、 make に -n というオプションつけると、どのコマンドが実行されるかだけがでて実行しないのね。なので、 -n とると

学生 A：あ、 keplerplot3 ができてますね。

赤木 ： もう一度 make してみて

学生 A：

```
make: 'keplerplot3' は更新済みです.
```

とでます。

赤木 ：これ、前にコンパイルした時からソースファイルとか関係するファイルが新しいかとかをチェックしてるの。なので、実行前は make するようにしておけばちゃんと最新のが使われるわけ。

学生 A： なるほど。

赤木 ： まず、離心率 と最初の 1 周期あたりのステップ数を与えて、10 周期まで積分した時の誤差を、ステップ数 2 倍にしながら 10 回繰り返してグラフ書く、ってどうかしら？あ、折角だから、離心率は複数値与えるようにして。 float vector ってのがあったでしょ？

学生 A：じゃあまあ作ってみます、、、

```
 1:require "grlib"
 2:require "clop"
 3:include Math
 4:include GR
 5:
 6:optionstr= <<-END
 7:  Description: Test integrator  driver for Kepler problem
 8:  Long description:
 9:    Test integrator driver for Kepker problem
10:    (c) 2020, Jun Makino
11:
12:  Short name:           -n
13:  Long name:  --nsteps-initial
14:  Value type:int
15:  Default value: 20
16:  Variable name: n
17:  Description:Initial number of steps per orbit
18:  Long description:     Initial number of steps per orbit
19:
20:  Short name: -o
21:  Long name:  --norbits
22:  Value type:int
23:  Default value:10
24:  Variable name:norb
25:  Description:Number of orbits
26:  Long description:     Number of orbits
27:
28:  Short name: -N
```

```
29:  Long name:  --number-of trial-integrations
30:  Value type:int
31:  Default value:10
32:  Variable name:ntry
33:  Description:
34:  Long description:
35:    Number of trial integrations. The timestep is halved at each
36:    iteration
37:
38:  Short name:-e
39:  Long name:--ecc
40:  Value type:float vector
41:  Default value:0.0,0.3
42:  Variable name:ecc
43:  Description:values of the eccentricity of the orbit
44:  Long description:     values of the eccentricity of the orbit
45:
46:  Short name:-y
47:  Long name:--rane-of-y
48:  Value type:float vector
49:  Default value:1e-15,1e-2
50:  Variable name:yrange
51:  Description:range of plot of y axis
52:  Long description:     range of plot of y axis
53:
54:  Short name:-t
55:  Long name:--integrator-type
56:  Value type:        string
57:  Variable name:itype
58:  Default value:LF
59:  Description:
60:    integrator scheme. LF:leapflog, Y4:Yosida4
61:  Long description:
62:    integrator scheme. LF:leapflog, Y4:Yosida4
63:END
64:
65:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
66:options=CLOP.new(optionstr,ARGV)
67:
68:n=options.n
69:system "make keplerplot3"
70:h0 = 2*PI/n
71:setwindow(h0/(1<<options.ntry), h0, options.yrange[0], options.yrange[1])
72:box(10,10, major_x:1, major_y:2, ylog: true, xlog: true)
73:setcharheight(0.04)
74:mathtex(0.5, 0.06, "h")
75:mathtex(0.01, 0.5, "\\Delta E")
76:options.ecc.each{|ecc|
77:  hs=Array(Float64).new
78:  errs=Array(Float64).new
79:  n=options.n
80:  options.ntry.times{
```

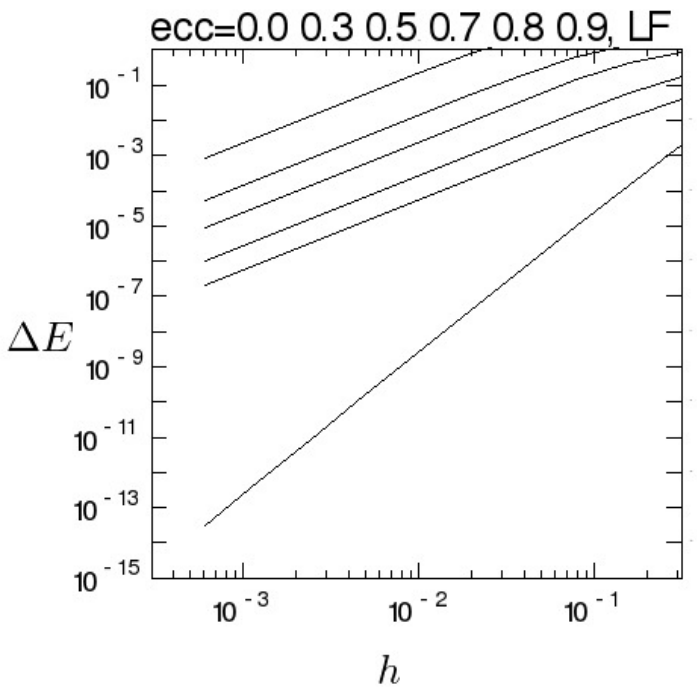

Figure 9.5: リープフロッグ、10 周期のケプラー問題をステップ数を変化させて数値積分して、最大誤差をプロットしたもの。離心率の値はグラフの上にある通る。

```
81:    errs.push `keplerplot3 -n #{n} -e #{ecc} -o #{options.norb} -t #{options.itype}`.
82:                   split.last.to_f.abs
83:    hs.push 2*PI/n
84:    n*=2
85:    pp! hs
86:    pp! errs
87:  }
88:  polyline(hs, errs)
89:}
90:text(0.2,0.91,"ecc="+options.ecc.join(" ")+", "+options.itype)
91:c=gets
92:
```

一応できて動いているんじゃないかと、、、こんなグラフがつくれます。

赤木 ： あ、なんかいい感じね。プログラム解説お願い。

学生 A： 63 行目まではまたオプション定義です。n, o, e, y, t とあって、もうそのテキストに書いてある通りですが、1 周期あたりのステップ数の初期値、積分する軌道周期、離心率 (コンマで区切って複数与える)、グラフ書く時の y 軸の最小、最大値、使う積分公式です。

赤木 ：一杯あるわね。

学生 A：あ、そうかもしれません。ふやすのが簡単なのでつい、、、

赤木 ：まあそういうふうに使うために作ったものだから、それでいいのよ。なるべくプログラムは

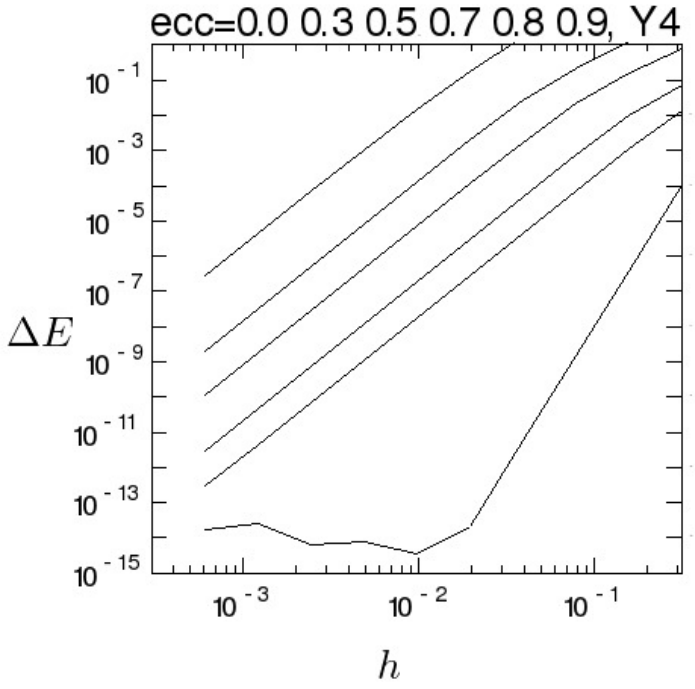

Figure 9.6: 図 9.5 と同じだが 4 次公式。

柔軟に、同じようなことだけどちょっとだけ違う、というのはパラメータで変えるように、とすると間違いも少ないわ。

学生 A：はあ、そうかもしれません。69 行目の system は、文字を OS のコマンドとして実行するものです。ここでは make かけて最新のソースからコンパイルします。本当はコンパイルエラーになったら止めるとかあるべきかもしれないですがさぼってます。

その後はグラフのレンジ決めて枠書いてラベル書くのが 75 行目までですね。76 行目の each で離心率の値毎のその中を実行で、その中は、線引くのに使う配列を 77,78 行列で空にして、80 行目が ntry 回ステップサイズ変えながら繰り返す、で、81 行で keplerplot3 を実行します。実行するのは system と同じですが、ここではバッククォートで文字列例を囲んでいて、これは実行結果の出力を文字列として返すものです。Ruby にもあってもっと古い言語にもあるんですかね？

赤木 ：perl にはあったわね。

学生 A:文字列の中で #{n} とかしているのは、そこが {} の中の式の値、文字型でなければ to_s で文字列にしたもの、に置き換わるものです。これも Ruby と同じですね。で、その文字列を split.last.to_f.abs で、これはスペースで区切りがあるとして配列に変換、配列の最後の要素、それを浮動小数点に変換、絶対値、と順番に関数適用ですね。その結果を配列 errs に追加するという格好です。

赤木 ： ピリオドの後で改行してもいいのね。

学生 A： やったら大丈夫でした。で、その次で時間刻みも配列にいれて、85 行で n を 2 倍にします。その後は計算できてそうかどうか画面にだしてるだけです。で、89 行でデータの線をひきます。これで本体はおしまいですね。

あ、離心率の値と積分公式もグラフにあったほうがいいかなと思って、91 行の text をつけました。

赤木 ： なるほど。あとはグラフのどの線がどの値かがわかるようななにかをつければ、このグラ

フなら論文にあってもおかしくないわね。

学生 A：(赤木さんのいうこと信用できるのかな、、、)

赤木 ：これ、とにかく、離心率ちょっと大きくなると誤差がものすごく大きくなるのがわかるわね？リープフロッグでも、0.3 から 0.9 までで 4 桁、4 次公式だと 6 桁くらい悪いのね。これどうしてかわかる？

学生 A：読者の演習問題ということでどうでしょうか？

赤木 ：そうねえ、、、

学生 A：(逃げる)

9.1 課題

1. 離心率を横軸にして、ステップサイズ毎にエネルギー誤差の線がでるようなグラフを作って下さい。
2. ステップサイズが同じ時、離心率を変えるとエネルギー誤差がどう変わるか、特に、離心率が 1 に近づいた時のどうなるか、それは何故か、を議論しなさい。
3. 1 万軌道くらいまで積分した時のエネルギー誤差を、いくつかのステップサイズと積分公式の組合せで書いてみて下さい。グラフは、両対数 (エネルギー誤差は絶対値) と、リニアプロットの両方を作って下さい。
4. 4 次ルンゲクッタでケプラー問題を積分するプログラムも作って (今あるものに機能追加して)、同じようなグラフを作って下さい。
5. エネルギー誤差の振舞いの積分公式による違い、それが何故起こるかを検討して下さい。

9.2 まとめ

1. Crystal で人が作ったライブラリを使う仕組みに shards というものがある。
2. 作者が作った、コマンドラインオプションを解釈する仕掛け clop-crystal を使ってみた。
3. make コマンドの最小限の使い方を学んだ。
4. ケプラー軌道の数値積分を行った。離心率が 1 に近いと問題が起こることがわかった。

9.3 参考資料

https://github.com/jmakino/clop-crystal

Chapter 10

多体問題

赤木 ： 今回は多体問題ということで。まあその、本当に大規模な数値計算するなら最近だと全部自分でプログラム書くってだいぶ無理で、並列化のところにパッケージ使うとか、初めから全部人が書いたのをもらってきて使うかなんだけど、基本的な仕掛けは自分でわかってて欲しいのね。

学生 A : そういうものですか？

赤木 ：そういうものよ。

学生 A : 赤木 ：学生 A : 赤木 : 学生 A :

10.1 課題

10.2 まとめ

10.3 参考資料

まだ中身決めてないので不明

Chapter 11

ハミルトニアン分割

赤木 :まだできてないの。ちょっと待ってね。