Crystal による数値計算入門

牧野淳一郎

March 24, 2022

PDF 版は こちら[1]
プログラムソースは こちら[2]

[1] ../intro_crystal.pdf
[2] https://github.com/jmakino/numerical-calculation-with-crystal

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
    i/interactive            starts interactive Crystal
    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 1.3.2 [932f193ae] (2022-01-18)

LLVM: 10.0.0
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 クラスのメソッドである to_s が、String を返すと String クラスのメソッドである 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。なお、「Nil クラスのメソッドである to_s」といちいち書く代わりに、Ruby や Crystal のドキュメントでは「Nil#to_s」と書くようです。この文書でも以下この表記を使います。

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。これは、左側から順番に評価して、そこをで値が決まったら右側はもう評価しない、という規則を明文化したものです。

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"
```

これだと、いくつかの関数を呼んでいるだけで、 while も if もありません。つまり、制御構造がないプログラムになっているわけです。

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
#<Vector3:0x7f308b382f90 @x=1.0, @y=2.0, @z=3.0>
#<Vector3:0x7f308b382f60 @x=1.0, @y=1.0, @z=1.0>
#<Vector3:0x7f308b382f30 @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
  include YAML::Serializable
  @x : Float64 = 0.0
  @y : Float64 = 0.0
  @z : Float64 = 0.0
  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
  def self.zero()
    Vector3.new
  end
  def to_a()
    [@x, @y, @z]
  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
[2mShowing last frame. Use --error-trace for full trace.[0m

In [4mvector3.cr:2:11[0m

[2m 2 | [0m[1minclude YAML::Serializable[0m
             [32;1m^-----------------[0m
[33;1mError: undefined constant YAML::Serializable[0m
```

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 C： 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 C：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 C：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。で、前の本は A 君に付き合ってもらったけど、彼はもうなんか偉くなっちゃってこんなのに付き合うほど暇じゃなさそうだから、今回は君にね。

学生 C：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 C：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 C：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 C：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 C : 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 C : 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 C : 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 C : 本当ですか？

赤木 : 多分。

学生 C : (うーん、、、、大丈夫かなあ、、、という気もするけど) まあお金がはいるならやってみます。

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 C : 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 C : 4 までやってみました。

赤木 : どうだった？難しい？

学生 C : 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 C : 楽しいんですか？

赤木 : まあ人によっては。

学生 C : 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 C：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 C：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 C：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、

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

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 C：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 C：はあ、、、やってみます。

(一週間後)

学生 C：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 C：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 C : 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 C :

```
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 C : 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 C : 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 C : 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思いますが、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 C : 大丈夫というと？

赤木 ：ちゃんと 2π になってる？

学生 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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

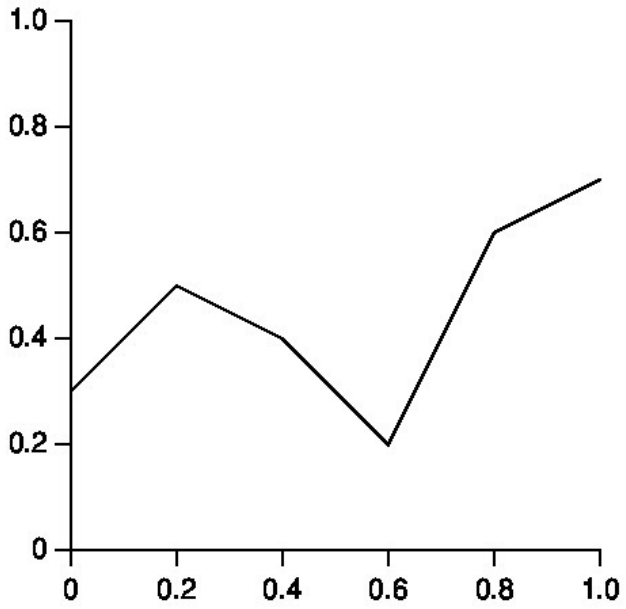

Figure 6.1: grsample (C 版) の出力結果

```
> 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 C： 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
  ```

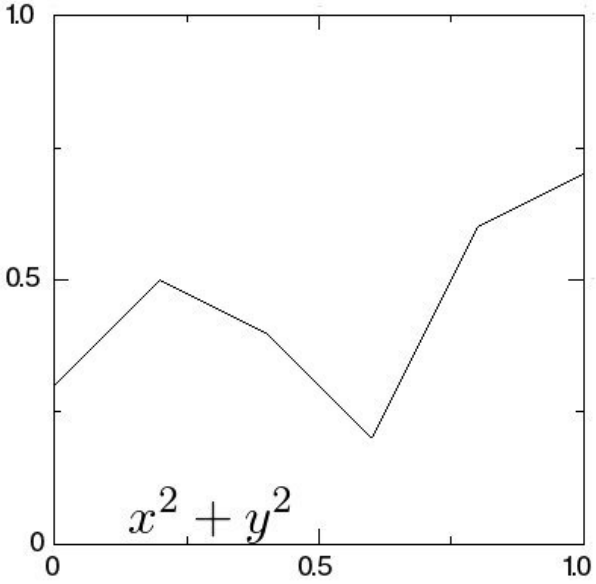

Figure 6.2: grsample (Crystal 版) の出力結果

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 C : 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 C： 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 C： 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 C : 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 C : 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 C : 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 C : 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 C : えーと、今日ですか？

赤木 : それはおまかせ。学生 C : 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
```

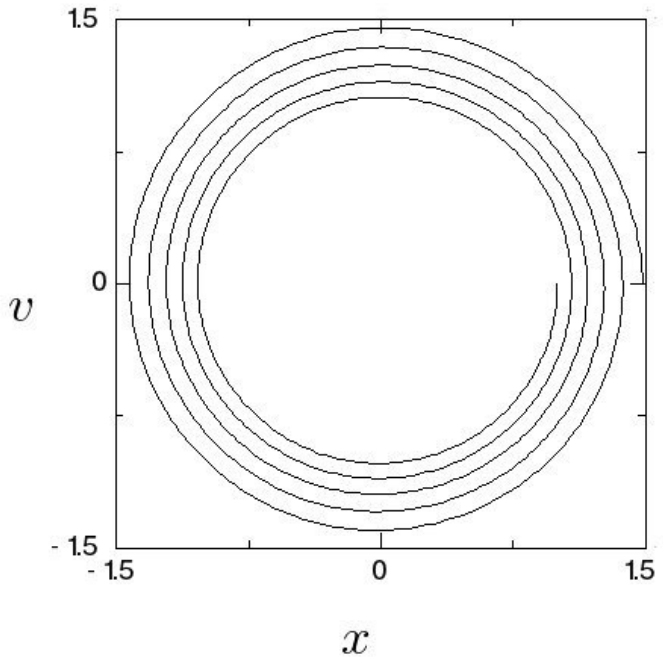

Figure 6.3: Euler 法での軌道。周期あたり 250 ステップ

```
  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 C：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 C：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 C：えーと、

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

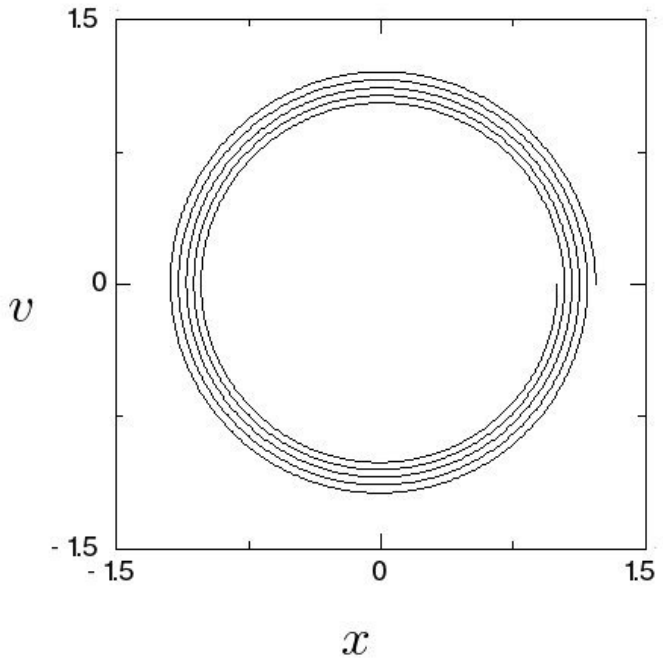

Figure 6.4: Euler 法での軌道。周期あたり 500 ステップ

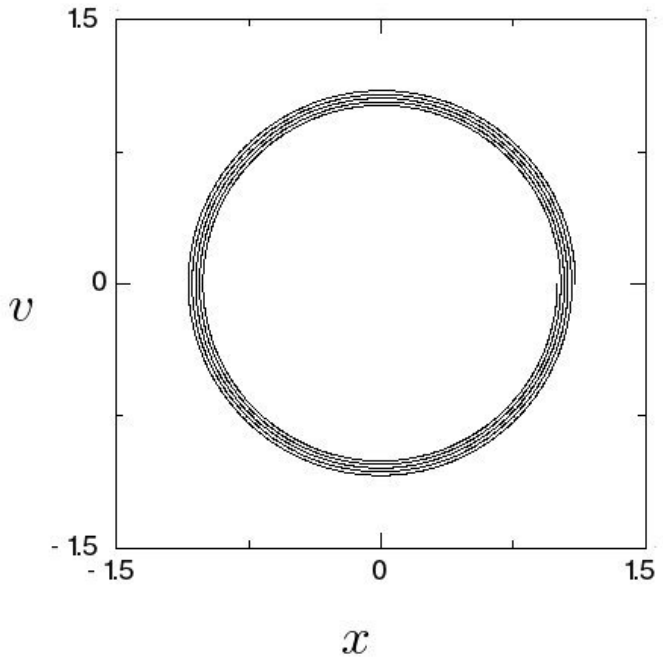

Figure 6.5: Euler 法での軌道。周期あたり 1000 ステップ

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 C :???

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 C : 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 C : 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 C : (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 C : 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 C： $a\cos\theta + ia\sin\theta = 1 + ih$ 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 C : 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 C： $(ae^{i\theta})^n w_0 = a^n e^{in\theta} w_0$ 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 C : 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 C : 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 C : 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 C : 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 C : 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 C： 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 C： 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 C : 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 C : 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 C : 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 C： 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 C：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 C：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 C：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 C：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 C : 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 C :

```
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 C : 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 C：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 C：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 C：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), t : 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 C：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 C：なるほど。t はどこにいっちゃうんですか？

赤木 ：これ、ルンゲクッタ公式としては t があるんだけど、今解く微分方程式には t がないわけね。なので、ルンゲクッタ公式のプログラムは t が引数にある関数がほしいから、そういう定義をして、{} の中では xx と t があるんだけど、実際の関数は harmonic は t 使わないから単に無視されるわけ。

学生 C：harmonic(xx,k) から 引数が (xx,t) である別の関数を作って、それを積分公式に渡す、みたいな感じですね。なんかちょっと複雑な気がしますが、、、

赤木 ：他の言語だと lambda とか使って、もっとわけのわからない書き方だったりするから、 Ruby 由来の書き方の Crystal はわりとわかりやすいかな。C++ だと C++14 になってラムダ式ができて、こんな感じ:

```
auto plus = [](int a, int b) { return a + b; };
int result = plus(2, 3); // result == 5
```

[] が謎だけどあとは Crystal の記法から想像つくでしょ？普通の関数みたいに見えて call とかつかないのが特殊ね。

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 C : 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 C : 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 C : 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 C： 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 C：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 C：(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 C：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
```

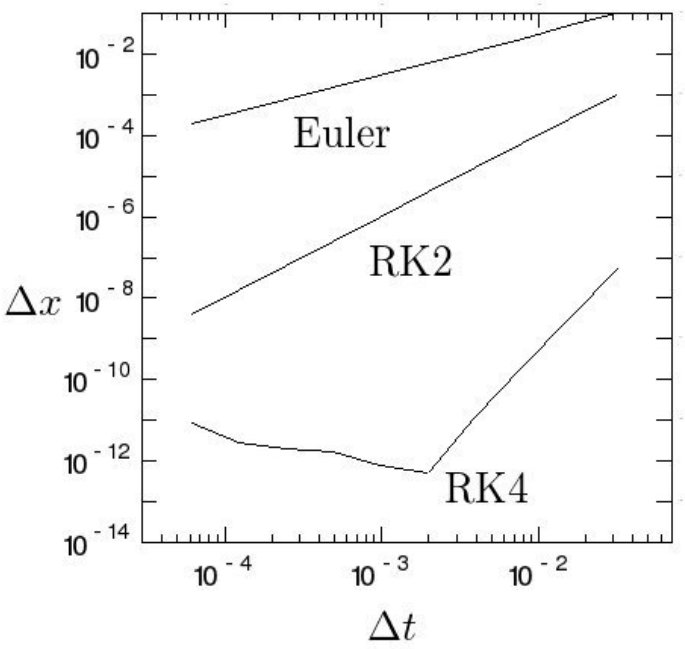

Figure 7.1: オイラー法及び 2 次、4 次のルンゲクッタの、時間刻みと精度の関係。

```
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
```

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 C : 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 C : 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 C : 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。

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

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

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 C : 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 C : 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 C : 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？

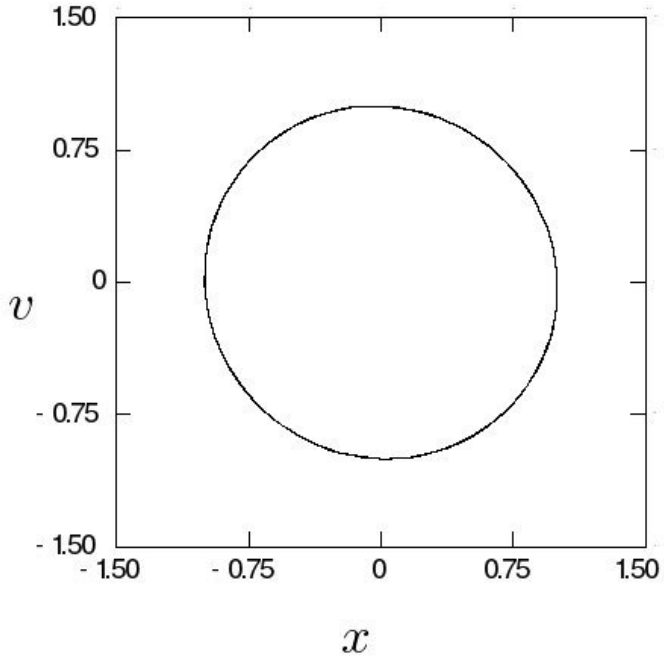

Figure 8.1: 修正版 Euler 法での軌道。周期あたり 100 ステップ

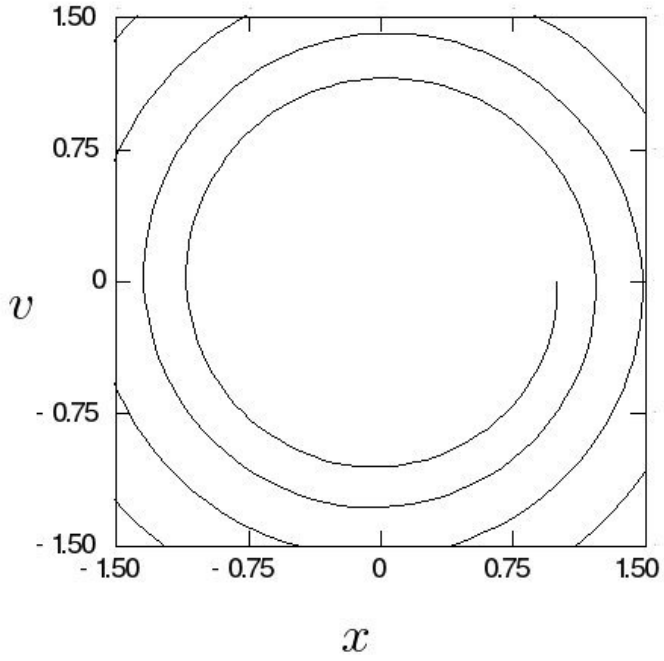

Figure 8.2: Euler 法での軌道。周期あたり 100 ステップ

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 C : 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 C : 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 C :

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

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 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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
```

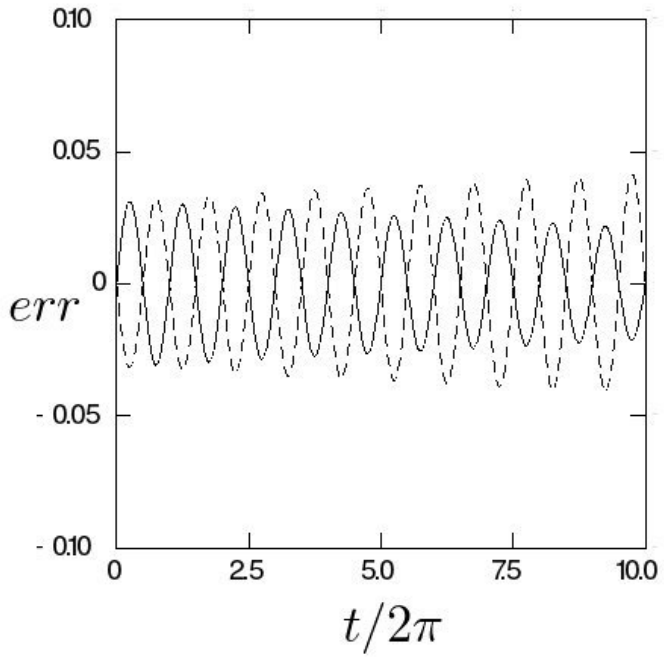

Figure 8.3: 2 つの修正版 Euler 法での x の誤差。周期あたり 100 ステップ。

```
  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 C : 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 C :

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

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 C : 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 C : 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)
```

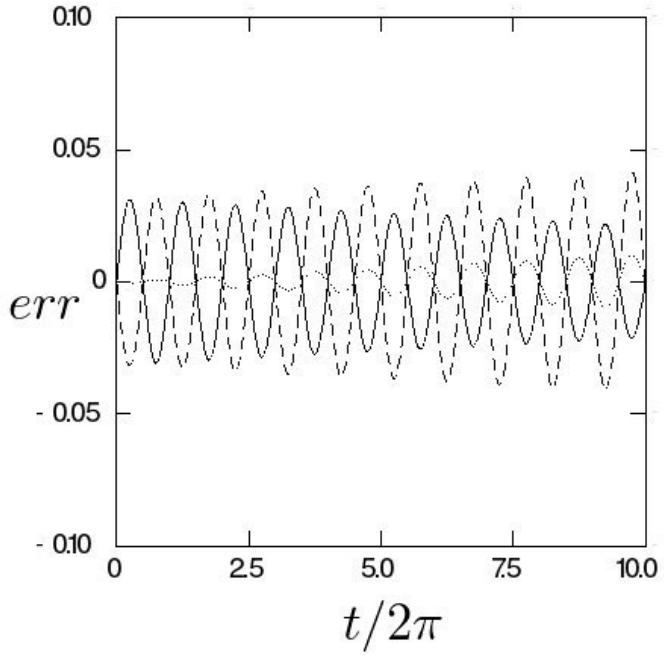

Figure 8.4: リープフロッグでの x の誤差。周期あたり 100 ステップ。

```
    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 C：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
```

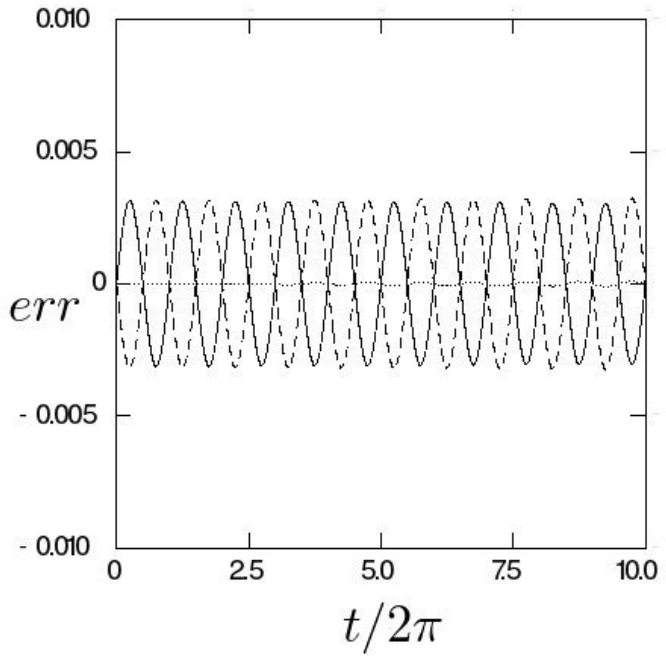

Figure 8.5: リープフロッグでの x の誤差。周期あたり 1000 ステップ。

```
  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
```

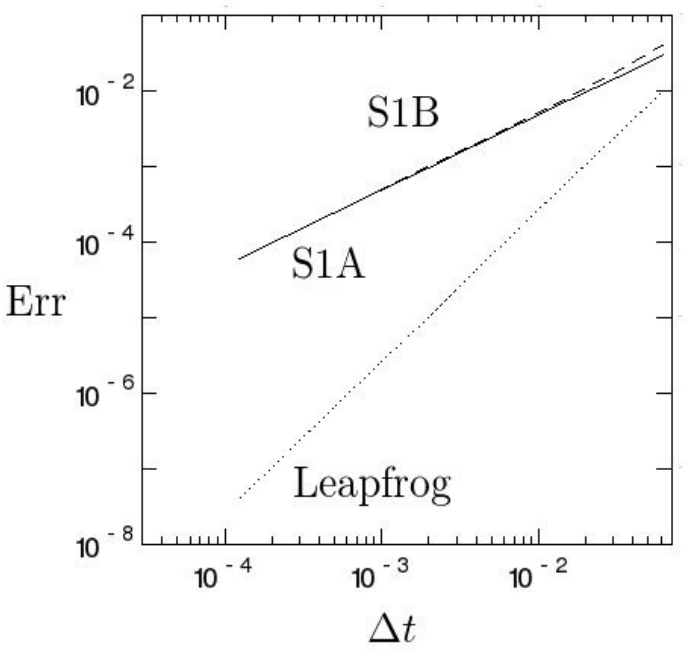

Figure 8.6: 3 つの公式での時間刻みと最大誤差。

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。

$$\begin{aligned} S_4(h) &= L(d_1h)L(d_2h)L(d_1h), \\ d_1 &= 1/(2-2^{1/3}), \quad d_2 = 1-2d_1 = -2^{1/3}/(2-2^{1/3}) \end{aligned} \tag{8.17}$$

学生 C : 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、

$$d_1 = d_7 = 0.784513610477560$$

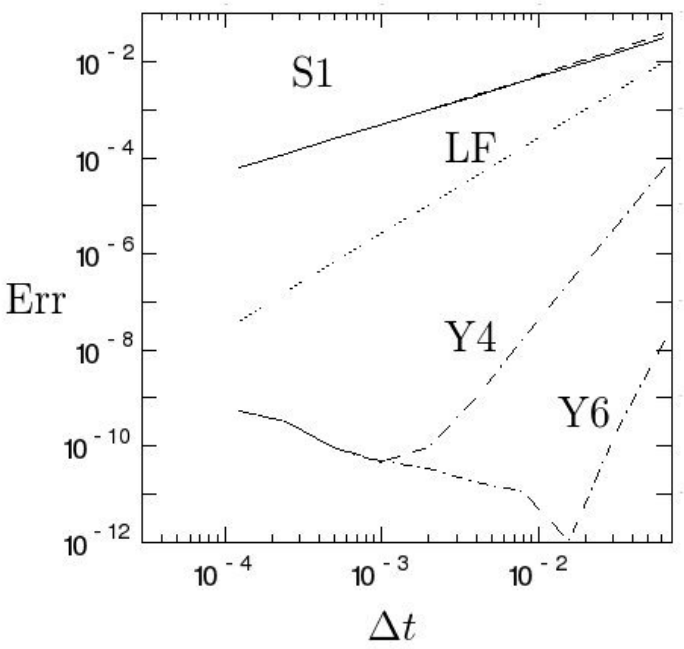

Figure 8.7: 6 次までの公式での時間刻みと最大誤差。

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

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 C : 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

ケプラー問題とコマンドラインオプション

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 C : 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 C : 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 C : 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 C : 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 C： 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 C：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:#    print "rk2 called\n"
13:    k1 = f.call(x,t)
14:    k2 = f.call(x+k1*h, t+h)
15:    {x + (k1+k2)*(h/2), t+h}
16:  end
17:
18:  def rk4(x,t,h,f)
19:#    print "rk4 called\n"
20:    hhalf=h/2
21:    k1 = f.call(x,t)
22:    k2 = f.call(x+k1*hhalf, t+hhalf)
23:    k3 = f.call(x+k2*hhalf, t+hhalf)
24:    k4 = f.call(x+k3*h, t+h)
25:    {x + (k1+k2*2.0+k3*2.0+k4)*(h/6), t+h}
26:  end
27:
28:  NITER = 5
29:  def gauss4(x, t, h, f)
30:    f1 =  f.call(x,t)
31:    f2 = f1
32:    a11 = 0.25
33:    a12 = 0.25 - sqrt(3.0)/6
34:    a21 = 0.25 + sqrt(3.0)/6
35:    a22 = 0.25
36:    t1 = t+(a11+a12)*h
37:    t2 = t+(a21+a22)*h
38:    NITER.times{
39:xg1 = x+(f1*a11+f2*a12)*h
40:xg2 = x+(f1*a21+f2*a22)*h
41:f1 = f.call(xg1,t1)
42:f2 = f.call(xg2,t2)
43:    }
44:    { x+(f1+f2)*0.5*h, t+h}
45:  end
46:
47:  def symplectic1a(x,v,h,f)
```

```
48:    x+= v*h
49:    v+=  f.call(x)*h
50:    {x,v}
51:  end
52:  def symplectic1b(x,v,h,f)
53:    v+=  f.call(x)*h
54:    x+= v*h
55:    {x,v}
56:  end
57:
58:  def leapfrog(x,v,h,f)
59:    f0 = f.call(x)
60:    x+= v*h + f0*(h*h/2)
61:    f1 = f.call(x)
62:    v+= (f0+f1)*(h/2)
63:    {x,v}
64:  end
65:
66:  D1 = 1.0 / (2-exp(log(2.0)/3))
67:  D2 = 1 - 2*D1
68:  def yoshida4(x,v,h,f)
69:    x,v= leapfrog(x,v,h*D1,f)
70:    x,v= leapfrog(x,v,h*D2,f)
71:    leapfrog(x,v,h*D1,f)
72:  end
73:
74:  def  yoshida6(x,v,h,f)
75:    d = {0.784513610477560, 0.235573213359357,
76:         -1.17767998417887, 1.31518632068391};
77:    4.times{|i|x,v = leapfrog(x,v,h*d[i],f)}
78:    3.times{|i|x,v = leapfrog(x,v,h*d[2-i],f)}
79:    {x,v}
80:  end
81:
82:
83:end
84:
85:module SymplecticIntegrators
86:  extend self
87:  def leapfrog(s, h)
88:    s.inc_vel(h*0.5)
89:    s.inc_pos(h)
90:    s.calc_accel
91:    s.inc_vel(h*0.5)
92:  end
93:  D1 = 1.0 / (2-exp(log(2.0)/3))
94:  D2 = 1 - 2*D1
95:  def yoshida4(s,h)
96:    leapfrog(s,h*D1)
97:    leapfrog(s,h*D2)
98:    leapfrog(s,h*D1)
99:  end
```

```
100:
101:  def  yoshida6(x,v,h,f)
102:    d = {0.784513610477560, 0.235573213359357,
103:         -1.17767998417887, 1.31518632068391};
104:    4.times{|i|leapfrog(s,h*d[i])}
105:    3.times{|i|leapfrog(s,h*d[2-i])}
106:  end
107:end
108:
```

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 C : 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 C : 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 C : 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 C : 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 C : 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 C : 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？

学生 C : 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。

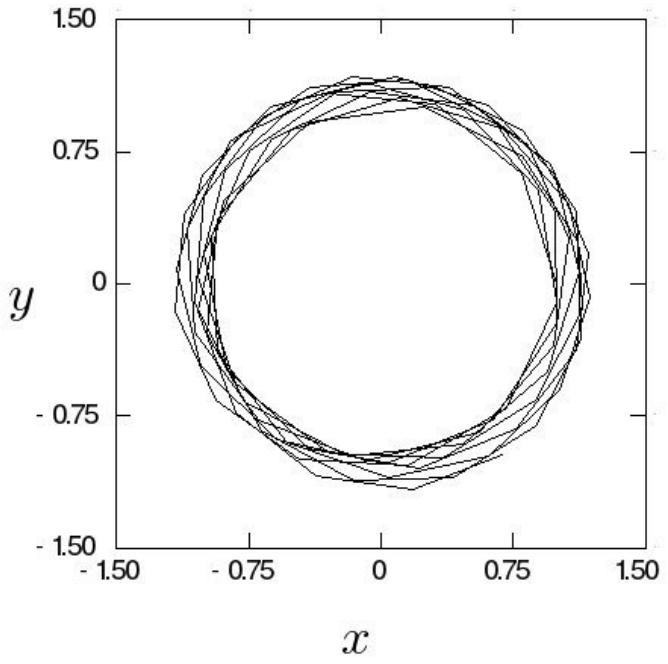

Figure 9.1: リープフロッグ、1 周期 10 ステップ、10 周期のケプラー問題円軌道の解

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 C : 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 C : 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。

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

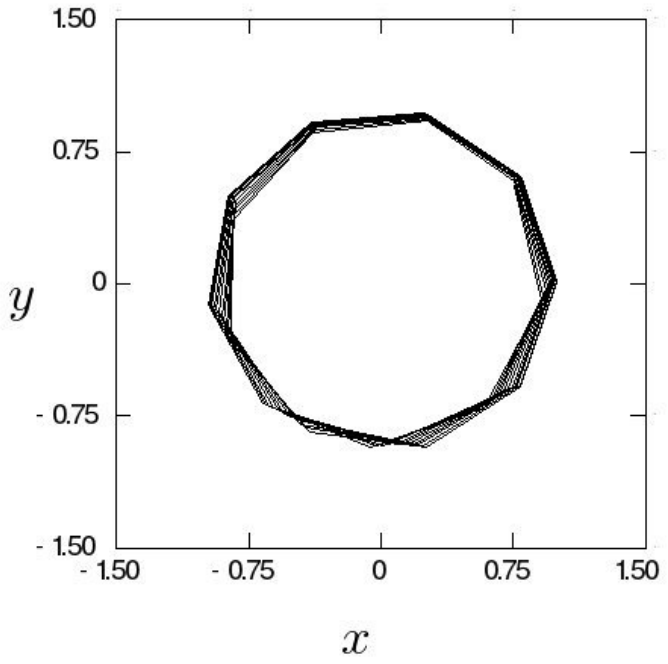

Figure 9.2: 4 次シンプレクティック、1 周期 10 ステップ、10 周期のケプラー問題円軌道の解

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

```
  nacsio:
    github: jmakino/nacsio
    branch: master
  narray:
    github: jmakino/narray
    branch: main
```

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 C : 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 C : 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 C : 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 C： 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 C :

```
gravity> crystal lib/clop/src/clopsample.cr
sh: 0: Can't open ./clop_process.sh
[2mShowing last frame. Use --error-trace for full trace.[0m

In [4mlib/clop/src/clop.cr:21:8[0m

[2m 21 | [0m[1m{{system("sh ./clop_process.sh #{l} #{f} #{d} #{strname}")}}[0m
        [32;1m^-----[0m
[33;1mError: error executing command: sh ./clop_process.sh 78 "/home/makino/papers/intro_crystal/lib/cl
```

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 C :

```
gravity> crystal lib/clop/src/clopsample.cr -- -h
sh: 0: Can't open ./clop_process.sh
[2mShowing last frame. Use --error-trace for full trace.[0m

In [4mlib/clop/src/clop.cr:21:8[0m

[2m 21 | [0m[1m{{system("sh ./clop_process.sh #{l} #{f} #{d} #{strname}")}}[0m
        [32;1m^-----[0m
[33;1mError: error executing command: sh ./clop_process.sh 78 "/home/makino/papers/intro_crystal/lib/cl
```

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 C : 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 C :

```
gravity> crystal lib/clop/src/clopsample.cr -- -n 10
sh: 0: Can't open ./clop_process.sh
[2mShowing last frame. Use --error-trace for full trace.[0m

In [4mlib/clop/src/clop.cr:21:8[0m

[2m 21 | [0m[1m{{system("sh ./clop_process.sh #{l} #{f} #{d} #{strname}")}}[0m
        [32;1m^-----[0m
[33;1mError: error executing command: sh ./clop_process.sh 78 "/home/makino/papers/intro_crystal/lib/cl
```

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 C：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 C：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 C：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 C：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)
```

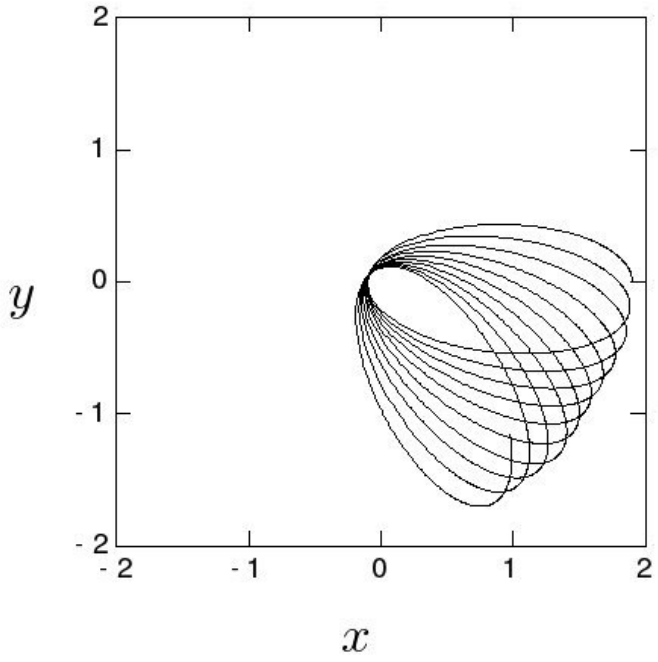

Figure 9.3: 離心率 0.9、10 周期のケプラー問題をリープフロッグで 1 周期 300 ステップで数値積分した結果。

```
100:  box
101:  setcharheight(0.05)
102:  mathtex(0.5, 0.06, "x")
103:  mathtex(0.06, 0.5, "y")
104:end
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、、、

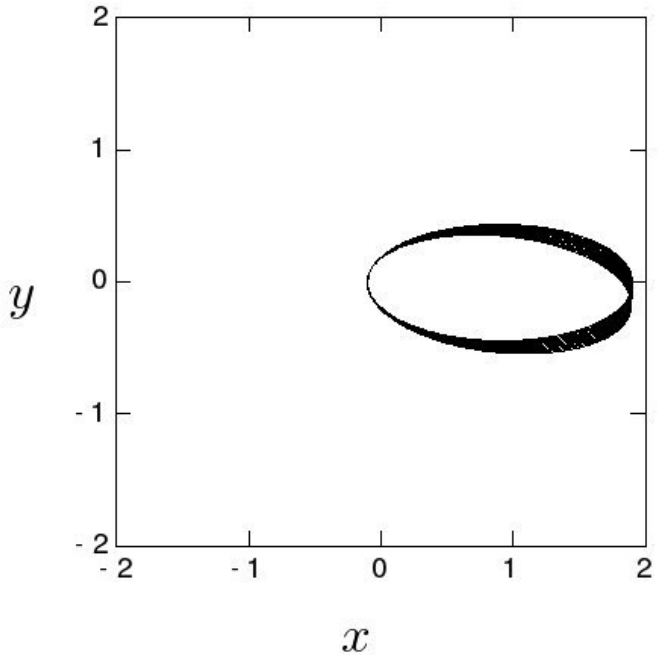

Figure 9.4: 図 9.3 と同じだが 1 周期 1000 ステップ。

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 C : 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。

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 C : 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 C : 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 C : 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 C： keplerplot3.cr です。

赤木 ： じゃあ、make -n keplerplot3 といれてみて？

学生 C： 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 C：あ、 keplerplot3 ができてますね。

赤木 ： もう一度 make してみて

学生 C：

```
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 C： 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 C：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:--range-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")
```

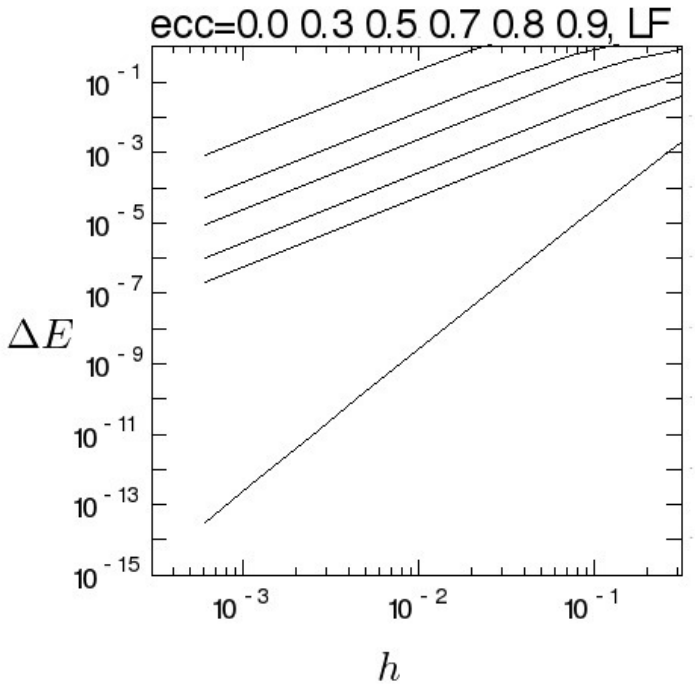

Figure 9.5: リープフロッグ、10 周期のケプラー問題をステップ数を変化させて数値積分して、最大誤差をプロットしたもの。離心率の値はグラフの上にある通る。

```
76:options.ecc.each{|ecc|
77:  hs=Array(Float64).new
78:  errs=Array(Float64).new
79:  n=options.n
80:  options.ntry.times{
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 C : 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

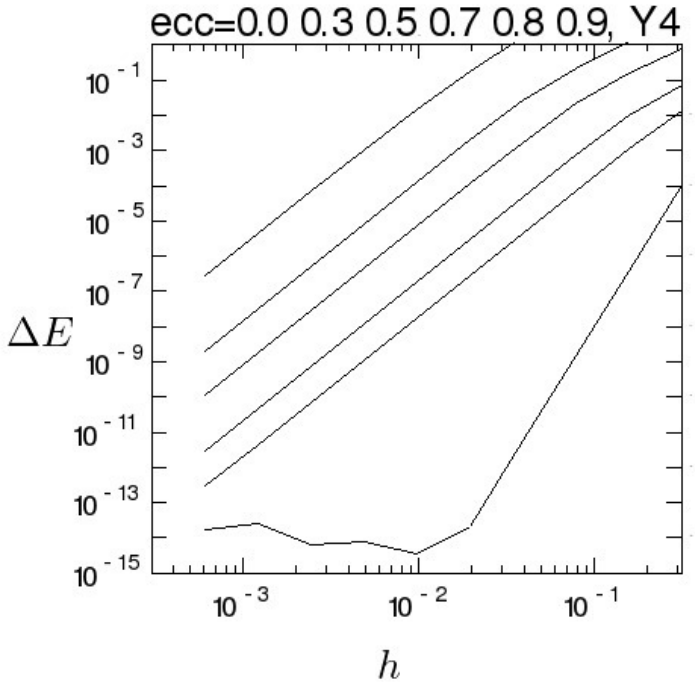

Figure 9.6: 図 9.5 と同じだが 4 次公式。

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 C：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 C：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 C: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 C： 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 C : (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 C : 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 C : (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

10.1　多体問題の設定と単位系

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 C : そういうものですか？

赤木 ：そういうものよ。ここではあんまり科学的な意味とかはを追求しないことにして、初期条件としては沢山星がある、例えば半径 1 の球の中の一様分布とかにしてみて。

学生 C： 質量と速度はどうしますか？あと 1ってどういう単位ですか？光年とかですか？

赤木 ：ケプラー問題では太陽質量を 1、重力定数 G も 1 にしたじゃない？そういう感じで、星団の質量を 1、重力定数 G も 1 としてみて。星の質量は、まずは N 個ならみんな等しく $m = 1/N$ ね。

学生 C : ケプラー問題だと、長さの単位は天文単位で、 2π が 1 年ということだったじゃないですか？

赤木 ：あら、別にそんなことはないわよ。それだと地球の軌道しかし計算できないみたいじゃない？

学生 C : あれ、そうじゃないんですか？木星なら軌道半径を 5.2 にしないと、周期が 12 年とかにならないですよね？

赤木 ：あー、これ割合説明がというか理解が難しい？1 が 1 天文単位だと思うと地球の軌道だけど、1 が 5.2 天文単位だと思うと、半径 1 の軌道が木星の軌道になるの。

学生 C :?

赤木 ：うーん、これ意外に説明難しいわね。運動方程式は

$$\frac{d^2x}{dt^2} = -GM\frac{\mathbf{x}}{x^3} \tag{10.1}$$

だったわけでしょ？まず、単位系をとりなおす、というのがどういう意味か、ということね。長さの単位って、 1m とか 1cm とか、あるいは 1 天文単位とか、あと星団とか銀河だと 1 パーセク、あ、何故か知らないけど天文学ではパーセク使って光年は使わないのね、とか色々あるでしょ？

今、物理学では単位といえば SI 単位のことで、それ以外駄目みたいな話になってるから、単位系を勝手にとるってあんまり馴染みがないかもしれないけど、天文学では系の大きさにみあった単位使わないとすごく計算間違いしやすくなるし、計算機でも上手く計算できなかったりするから、単位系とりなおす話は重要なの。

SI だと、G の値は、1 kg のものが 1m 離れたところにある時にそれから受ける重力加速度を $1\mathrm{m/s}^2$ を単位にして測ったものでしょ？そうすると、例えば $X\mathrm{m}$ を単位としたらどうなるかしら？加速度の単位も $X\mathrm{m/s}^2$ になるわけね。

学生 C： X のところにあるから、 1m の時より $1/X^2$ に小さくなりますね。で、加速度が 1 というのも $X\mathrm{m/s}^2$ を 1 と思うわけですね。だからもうひとつ X がはいって G は $1/X^3$ に小さくなります。

赤木 ：多分あってるんじゃないかしら。じゃあ時間の単位のほうを、1 秒でなくて T 秒にしたら？

学生 C: 加速度の単位が 1 秒あたり、「1 秒あたり動く距離」の変化だったのが、T 秒あたり、「 T 秒あたり動く距離」の変化に変わるわけですね。例えば $T = 2$ なら、$1\mathrm{m/s}$ は $2\mathrm{m}/(2\mathrm{s})$ だから、 $1\mathrm{m/s}^2$ は T 秒後に $T\mathrm{m/s}$ で、それは $T^2\mathrm{m}/(\mathrm{Ts})$ で、だから T^2 で大きくなればいいわけですね。

赤木 ：そう。そうすると、 X と T を上手くとれば G が変わらないでしょ？

学生 C： $T^2 = X^3$ ということですね。だから $T = X^{3/2}$ ですね。

赤木 ：そう。これケプラーの第三法則と同じ形でしょ？

学生 C : 第三法則ってなんでしたっけ？

赤木 ： 軌道周期が軌道長半径の 3/2 乗ってやつ。

学生 C : 確かに同じですが、でも？

赤木 ：つまり、天文単位を 1、1 年を 2π にとる代わりに、木星の軌道長半径を 1、木曜の軌道周期を 2π にする単位系でもいいし、他の惑星でもなんでもいいわけ。

星団でも、例えば 1 パーセクの中に合計で太陽質量の 100 倍があると思って、それを $M = G = 1$ で計算するなら、時間の単位が決まるわけ。

あ、質量の単位を M 倍にしたらどうなるかしら？

学生 C : 加速度は M 倍だから、 G も M 倍？

赤木 ：そうね。だから、地球の周りの人工衛星でも月でも、木星の衛星でも、土星のリングでも、$G = M = 1$ として、あと長さの単位を決めると、それから時間の単位が決まるわけ。で、それはもちろん、円運動の周期が 2π になるように、つまり、その長さのところでの円運動の角速度が 1 になるように、ということになるわけ。

これは太陽系でなくてお互いの重力で運動する系でも式変形の原理は同じだから、同じように $G = M = 1$ として、あと長さの単位を決めると、それから時間の単位が決まるわけ。

学生 C : でも星団みたいなのだと星は円運動するわけじゃないし、ケプラー運動でもないですよね？時間の意味ってなんですか？

赤木 ：まず、形式的にはとにかく、実際の何年というのに対応するわけ。あと、星の運動は色々だけど、平均の速度とか、系の特徴的な半径とか、そういうのは定義できるわね。

学生 C : うーん、そういうものですか。

赤木 ：まあ、実際に星動かしてみるともうちょっと気分わかるかも。位置は半径 1 の球の中の乱数でやるとして、速度はなんかパラメータ与えてやっぱりその半径の球の中で一様とかしてみて。熱力学的にはボルツマン分布とかがもっともらしいんだけど、星団でボルツマン分布って重力で束縛されない (無限遠までいっちゃう) 粒子がでてきてちょっと取り扱いが厄介なの。

学生 C： だからといって速度分布や粒子分布を適当に作ることになんか意味あるんですか？

赤木 ：うん、そうね、これ、わりと意味ないわけでもないの。これは要するに、重力多体系というものはどう進化するか、という話ね。まず、理論的には、沢山の粒子でできてるわけだから、統計力学で記述されるはずで、古典統計だからボルツマン分布になるんだけど、さっきの話でボルツマン分布だと粒子が無限遠にいっちゃうからそうならないわけ。でも、1 つ 1 つの星についてみると、ランダムに速度変化して、速度というかエネルギーの空間の中で拡散していくのね。なので、そのうちに、初期条件がなんでも、同じような分布になるというか、同じような進化をするようになるの。

学生 C : はあ。

赤木 : まあ、ちょっと実際にやってみて。

学生 C : 了解です。

赤木 : あ、あとね、ケプラー問題では、軌道が楕円軌道で変化しなかったけど、多体問題ではみんな勝手に動くわけで、2 つの粒子がすごく近くを通ることがあるのね。そうすると、タイムステップいくら短くしておいても破綻するの。

この、近くを通るのを、英語では close encounter っていうの。ちょっと余談だけど、君くらい若いと知らないかもだけど、スピルバーグの映画に「未知との遭遇」というのがあって、英語の原題は Close encounters of the third kind なのね。この第三種というのは誰かが勝手に作った宇宙人だかとの遭遇の分類だけど、 close encounter 自体は普通につかう言葉なのね。

学生 C : はい？えーと、、、

赤木 : なので、今回、昔から使われている方法なんだけど、重力ポテンシャルの式を、 $-1/r$ じゃなくて $-1/\sqrt{r^2+\epsilon^2}$ としてみて。ϵ は定数で、パラメータとして入力できるようにして。

学生 C : そうすると、軌道とか変わるわけで、なんか違うものを計算していることにならないですか？

赤木 : もちろん違うんだけど、統計的には同じ性質になると信じましょうみたいな感じ。まあ、 ϵ いくつか変えてみて、同じような性質がでてきたら大丈夫とかね。ちなみにこの ϵ を重力 softening (ソフトニング) というの。

学生 C : そんなものでしょうか？

赤木 : あ、もちろん、もうちょっと色々もっともらしい議論はできるし、じゃあ結果が ϵ を 0 にもっていった極限で収束するのかとかそういう話はできるし、しないといけないわけよ。まあそういうのはそのうちにね。

学生 C : わかりました。では実際に作ってみます。

(このあと無限に色々あったが全部省略)

10.2 多体問題のプログラム

学生 C : あ、なんか動いてるような気がします。プログラム particle0.cr というのを作りました。入力パラメータはこんな感じです

```
  -n     --numner-of-particles          int        2 Number of particles
  -s                --softening        float      0.0 Size of softening
  -e             --eccentricity        float      0.0 Eccentricity. Used only when n=2
  -d                --step-size        float     0.01 Size of timestep
  -T                 --end-time        float      1.0 Time to stop integration
  -w              --window-size        float      1.5 Window size for plotting
  -e                      --ecc        float      0.0 Initial eccentricity of the orbit
  -v          --velocity-scale        float      0.5 Scaling factor for the initial velocity
```

粒子数、ソフトニングを指定できて、あと n=2 の時には離心率指定して、そうでない時には velocity scale のほうを与えます。後は時間刻みと、シミュレーションやめる時刻と、絵書くのでその座標範囲いれます。この辺はケプラー問題の時のを流用してます。

で、アニメーションがでるんですが、2 体だとそれっぽく回って、3 とかそれ以上だとなんか動いてました。

赤木 ：んーと、計算正しいかどうかってなんかわからない？まあその前にプログラム見せて。
学生 C：はい。

```
 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:  --numner-of-particles
14:  Value type:int
15:  Default value: 2
16:  Variable name: n
17:  Description:Number of particles
18:  Long description:     Number of particles
19:
20:  Short name:           -s
21:  Long name:  --softening
22:  Value type:float
23:  Default value: 0.0
24:  Variable name: eps
25:  Description:Size of softening
26:  Long description:     Size of softening
27:
28:  Short name:           -e
29:  Long name:  --eccentricity
30:  Value type:float
31:  Default value: 0.0
32:  Variable name: ecc
33:  Description:Eccentricity. Used only when n=2
34:  Long description:     Eccentricity. Used only when n=2
35:
36:  Short name:           -d
37:  Long name:  --step-size
38:  Value type:float
39:  Default value: 0.01
40:  Variable name: h
41:  Description:Size of timestep
42:  Long description:     Size of timestep
43:
44:  Short name: -T
45:  Long name:  --end-time
46:  Value type:int
47:  Default value:1.0
48:  Value type:float
```

```
 49:  Variable name:tend
 50:  Description:Time to stop integration
 51:  Long description:     Time to stop integration
 52:
 53:  Short name:-w
 54:  Long name:  --window-size
 55:  Value type:  float
 56:  Variable name:wsize
 57:  Default value:1.5
 58:  Description:Window size for plotting
 59:  Long description:
 60:    Window size for plotting orbit. Window is [-wsize, wsize] for both of
 61:    x and y coordinates
 62:
 63:  Short name:-e
 64:  Long name:--ecc
 65:  Value type:float
 66:  Default value:0.0
 67:  Variable name:ecc
 68:  Description:Initial eccentricity of the orbit
 69:  Long description:     Initial eccentricity of the orbit
 70:
 71:
 72:  Short name:-v
 73:  Long name:--velocity-scale
 74:  Value type:float
 75:  Default value:0.5
 76:  Variable name:vscale
 77:  Description:Scaling factor for the initial velocity
 78:  Long description:
 79:    Scaling factor for the initial velocity.
 80:    positions and velocities are set from random vectors within
 81:    spheres of radius 1 and vscale.
 82:
 83:END
 84:
 85:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
 86:options=CLOP.new(optionstr,ARGV)
 87:
 88:require "./vector3.cr"
 89:class Particle
 90:  property :m, :x, :v, :acc, :phi
 91:  def initialize(m : Float64=0, x : Vector3=Vector3.new(0,0,0),
 92:                 v : Vector3=Vector3.new(0,0,0))
 93:    @m = m; @x = x;  @v = v
 94:    @acc = Vector3.new(0,0,0); @phi=0.0
 95:  end
 96:  def self.random(m, rx, rv)
 97:    Particle.new(m.to_f, randomvector(rx), randomvector(rv))
 98:  end
 99:  def calc_gravity(p,eps2)
100:    dr = @x - p.x
```

```
101:    r2inv = 1.0/(dr*dr+eps2)
102:    rinv = sqrt(r2inv)
103:    r3inv = r2inv*rinv
104:    @phi -= p.m*rinv
105:    p.phi -= @m*rinv
106:    @acc -= p.m*r3inv*dr
107:    p.acc += @m*r3inv*dr
108:  end
109:
110:end
111:
112:class ParticleSystem
113:  property :particles
114:  def initialize()
115:    @particles = Array(Particle).new
116:  end
117:  def +(p : Particle)
118:    @particles.push p
119:    self
120:  end
121:  def self.random(n, rx, rv)
122:    ps=ParticleSystem.new
123:    m = 1.0/n
124:    n.times{ ps += Particle.random(m,rx,rv)}
125:    ps.adjust_cm
126:    ps
127:  end
128:  def self.twobody(x,v)
129:    ps=ParticleSystem.new
130:    n=2
131:    m = 1.0/n
132:    ps += Particle.new(m,x,v)
133:    ps += Particle.new(m,-x,-v)
134:    ps
135:  end
136:  def clear_gravity
137:    @particles.each{|p|p.acc=Vector3.new; p.phi=0.0}
138:  end
139:  def calc_gravity(eps2)
140:    @particles.each_with_index{|p,i|
141:      ((i+1)..(particles.size-1)).each{|j|
142:        p.calc_gravity(@particles[j], eps2)
143:      }
144:    }
145:  end
146:  def leapfrog(h,eps2)
147:    @particles.each{|p|p.x += p.v*h + p.acc*(h*h*0.5)}
148:    @particles.each{|p|p.v += p.acc*(h*0.5)}
149:    clear_gravity
150:    calc_gravity(eps2)
151:    @particles.each{|p|p.v += p.acc*(h*0.5)}
152:    self
```

```
153:  end
154:  def calc_cm
155:    sumx=Vector3.new
156:    sumv=Vector3.new
157:    summ=0.0
158:    @particles.each{|p|
159:      sumx += p.m*p.x
160:      sumv += p.m*p.v
161:      summ +=p.m
162:    }
163:    {sumx*(1/summ),sumv*(1/summ)}
164:  end
165:  def adjust_cm
166:    cmx,cmv = calc_cm
167:    @particles.each{|p|
168:      p.x -= cmx
169:      p.v -= cmv
170:    }
171:  end
172:end
173:  def randomvector(r)
174:    sqsum= r*r*2
175:    v=Vector3.new
176:    while sqsum > r*r
177:      v = Array.new(3){rand(r)*2-r}.to_v
178:      sqsum = v*v
179:    end
180:    v
181:  end
182:
183:
184:ENV["GKS_DOUBLE_BUF"]= "true"
185:
186:#ps = ParticleSystem.random(2,1.0,0.4)
187:if options.n == 2
188:  ps = ParticleSystem.twobody([0.5*(1+options.ecc),0.0,0.0].to_v,
189:                              [0.0,0.5*sqrt((1-options.ecc)/(1+options.ecc)),0.0].to_v)
190:else
191:  ps =ParticleSystem.random(options.n,1.0,options.vscale)
192:end
193:eps2= options.eps*options.eps
194:ps.calc_gravity(eps2)
195:wsize=options.wsize
196:setwindow(-wsize, wsize,-wsize, wsize)
197:
198:time=0
199:while time < options.tend+options.h/2
200:  ps = ps.leapfrog(options.h, eps2)
201:  time += options.h
202:  clearws()
203:  box
204:  setcharheight(0.05)
```

```
205:  mathtex(0.5, 0.06, "x")
206:  mathtex(0.06, 0.5, "y")
207:  text(0.6,0.91,"t="+sprintf("%.3f",time))
208:  setmarkertype(4)
209:  setmarkersize(1)
210:  polymarker(ps.particles.map{|p| p.x[0]}, ps.particles.map{|p| p.x[1]})
211:  updatews()
212:end
213:c=gets
214:
```

です。解説しますか？

赤木 ：お願い。

学生 C：最初の 4 行はグラフィクスとかコマンドラインとか数学ライブラリ使いますですね。6 から 83 行目まではオプション定義です。上の -n から -v まで。88 行は前に作ってもらった 3 次元ベクトルを使います。

89 行から粒子クラスです。こういうの作るのがいいかどうかよくわかってないですが、質量 m、 位置 x、速度 v、加速度 acc、ポテンシャル phi がメンバー変数としてあって、それら全部普通に p.phi みたいなふうにアクセスできるとしてます。

initialize は粒子作る時に Particle.new するわけですが、それから呼ばれる関数でした。なにも引数なければ全部 0 に、ということでデフォルト値と、あとここでは引数に型を与えます。

その次の self.random は、クラスメソッドというやつですね。Particle.random(m, rx, rv) で、質量 m、位置と速度はそれぞれ半径 rx, rv の球の中の一様乱数です。

中では randomvector っていう関数で球の中の乱数生成します。

赤木 ：あら、その randomvector ってどこにあるの？

学生 C：173 行からですね。これは、球に外接する立方体の中の乱数は、3 成分独立に作ればいいのでそうして、半径 1 の外側ならその乱数使わないで、球の中に入るまで繰り返す、という方法です。わりとこれ標準的みたいです。

赤木 ：そうね。 rejection method ね。乱数の半分くらい捨てるわけだから計算量的にはもったいないけど、まあ、ちゃんと書けてる可能性が高いから。そういえば、これ Crystal の面白いとこで、関数定義ってどこにあってもいいのね。

これが普通のコンパイル言語だと、よっぽど古いのでない限り、あるところで使う関数はその前で定義がでてこないといけないのね。で、その定義って、関数自体の型と引数の型なの。古い Fortran (Fortran77 とかその前) だと、使うところで関数自体の型は宣言できるけど、引数は特に宣言する方法とかないから、呼ぶところと定義するところでちゃんとつじつまがあってるかどうかはプログラム書く人任せで、もちろん間違えるわけ。で、 Algol 系の言語のほとんどでは、関数定義と使うところで矛盾がないかどうかチェックするようになってるの。でも、そうすると、コンパイラが、ある関数が呼ばれるところを処理するにはあらかじめその関数がどういうものか知らないといけないわけ。

単純には、これ、関数の定義が関数使うところより前にあればいいわけで、Pascal とかは基本的にそうなってるのね。でも、例えば 2 つの関数がお互いを自分の中で呼びたいとかあるとこれでは不足だから、そういう時のために関数と引数の型だけ宣言できるような文法があるの。で、最近の言語、特に C や C++ では、関数の引数と型の宣言のことをプロトタイプ宣言といって、それが書いてあるファイルを、その関数使うところの前で読み込む、というふうにするわけ。 C だと

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

みたいなのとか、C++ だと

```
#include <iostream>
```

とかね。

学生 C： Crystal でも require でなんかインクルードしてるからおんなじじゃないですか？

赤木 ： あ、でも、それ、必ずしも使うところの前にないといけないわけではないの。普通

```
require "./vector3.cr"
pp! Vector3.new(0)
```

みたいに使う前に require するけど、実は

```
pp! Vector3.new(0)
require "./vector3.cr"
```

でもコンパイル通って動くの。これなんか気持ち悪いし、Ruby だと最初の実行文のところで Vector3 知りません、になんるんだけ、Crystal はあとでもかまわないのね。これは、コンパイラが、プログラム全体読み込んでから、実行箇所の最初から辿り直す、みたいことをするからなのね。呼ばれる関数のほうでは型書いてなくてもいいから、そうしないと逆にコンパイルできないわけ。

C++ なんかだと、そういう、コンパイル時に呼び出し側から型がつたわってくる関数はテンプレート関数というものになって、「これはテンプレート関数です」という形の宣言が必要ですごく複雑になって、それをもうちょっと簡単にするための新しい文法ができたりして一層わけがわからなくなんるんだけど、Crystal だとその辺コンパイラが何とかするように頑張って作ってあって、見かけ上実行時に型がわたってくる Ruby と概ね同じなんだけど、コンパイル言語である、ということからこんなふうに Ruby でもできない気持ちが悪いこともできるのね。

学生 C : (あんまり良くわかってない) はあ、、、

赤木 ：と、ちょっと余談よねこれ。次の関数は？

学生 C : 次は calc_gravity で、2 粒子間の重力を計算します。式の通りであんまりいうことないですが、自分と、もうひとつの同じ型の粒子で相互重力計算して、両方の acc と phi にそれぞれが受ける加速度とポテンシャルを加算していきます。粒子クラスはこれだけです。

赤木 ：その次は？

学生 C： 112 行目からは ParticleSystem です。これ基本的には単に粒子の配列をもっているクラスで、その配列に particles という名前をつけてます。initialize は、なので、particles を長さ 0 の Particle クラスの配列、としてます。

その次の self.random と self.twobody は、それぞれ乱数で n 個の粒子を作るのと、指定した位置、速度 (一方は符号ひっくり返して) で 2 粒子、という関数です。2 粒子のほうは、このプログラムでちゃんとケプラー問題できるかどうかのチェック用です。

で、次の clear_gravity は、関数でなくてもいいんですが、各粒子の acc, phi をゼロにして、calc_gravity は粒子間重力の計算です。

10.3 数値積分法ライブラリと系をあらわすクラス

赤木 ： その次は？

学生 C： はい、これがなんかちょっと嫌なんですが、この ParticleSystem クラス用のリープフロッグ積分関数になってます。

赤木 ：嫌ってのは？

学生 C : 時間積分法としては同じなのに、折角作った integratorlib.cr の中の leapflog とはちがうものをもう一度書いてるわけで、なんか無駄というか格好悪くないですか？

赤木 ：うーん、そうねえ、これ確かにちょっと難しいところね。intergratorlib.cr の leapfrog は

```
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
```

で、ParticleSystem のは

```
def leapfrog(h,eps2)
  @particles.each{|p|p.x += p.v*h + p.acc*(h*h*0.5)}
  @particles.each{|p|p.v += p.acc*(h*0.5)}
  clear_gravity
  calc_gravity(eps2)
  @particles.each{|p|p.v += p.acc*(h*0.5)}
  self
end
```

だから、なんかプログラムの見かけは全然違うわね。でも、やってることはもちろん本当は同じよね。intergratorlib.cr のほうはプログラム適当に書いてあるから、加速度 2 回計算しているけど、加速度の変数別にもってればもちろんそれ省略できるし。

数学的というか、古典力学の定式化のほうからみると、シンプレクティック公式だから、ハミルトニアンに対して定義されるわけで、ハミルトニアンって $H(p,q)$ じゃない？ p が一般化運動量で q が一般化座標で。そう思うと、intergratorlib.cr の leapfrog は割とそれに近い形で、f が正準方程式の $-\partial H/\partial q$ になってるわけね。ParticleSystem のは、粒子、つまり Particle クラスのデータ構造が、ハミルトニアンの型式とはあってないのね。

じゃあ、粒子クラスなんてのは止めてしまって、x とか v を 3N 次元のベクトルにしちゃえばいいか、っていうと、それもなんだかなあというところがあって。だって、「1 つの粒子」ってやっぱり実体としてある気がするわけで、それがデータ構造として存在しないのはなんか気持ちが悪いし、初期条件を作るとか結果を解析するとかの時にはやっぱり粒子を単位に扱いたいじゃない。

学生 C : はい、でも、それと今のシンプレクティック公式のライブラリは上手くあわないですよね？

赤木 ：そうねえ、、、いろいろ考え方があると思うんだけど、一つは、ParticleSystem みたいな物理系をあらわすクラスのほうに、リープフロッグ公式を作るのに必要なだけのメソッドを用意する、ということね。つまり、リープフロッグで必要なのは

- 加速度を計算する
- 加速度と時間刻みを使って速度をアップデートする
- 速度と時間刻みを使って位置座標をアップデートする

だから、まあ、あんまり難しいことを考えないで、クラスが決めうちの名前でそういう関数提供するとすれば、リープフロッグ公式のほうは例えば

```
def leapfrog(s, h)
  s.inc_vel(h*0.5)
  s.inc_pos(h)
  s.calc_accel
  s.inc_vel(h*0.5)
end
```

でいいし、高次のシンプレクティック公式もこれ使って書けるわけね。名前決めうちは美しくないと思うなら、3 つ関数渡すのでもいいけどかえってややこしくなるだけだと思うわ。

学生 C： じゃあやってみます。

(次の日くらい)

一応できて動いてると思いますが、、、

赤木 ：あら、速いわね。プログラムは？

```
  1:require "grlib"
  2:require "clop"
  3:require "./integratorlib.cr"
  4:include Math
  5:include GR
  6:
  7:optionstr= <<-END
  8:  Description: Test integrator  driver for Kepler problem
  9:  Long description:
 10:    Test integrator driver for Kepker problem
 11:    (c) 2020, Jun Makino
 12:
 13:  Short name:           -n
 14:  Long name:  --numner-of-particles
 15:  Value type:int
 16:  Default value: 2
 17:  Variable name: n
 18:  Description:Number of particles
 19:  Long description:     Number of particles
 20:
 21:  Short name:           -s
 22:  Long name:  --softening
 23:  Value type:float
 24:  Default value: 0.0
 25:  Variable name: eps
 26:  Description:Size of softening
 27:  Long description:     Size of softening
 28:
 29:  Short name:           -e
 30:  Long name:  --eccentricity
 31:  Value type:float
 32:  Default value: 0.0
 33:  Variable name: ecc
 34:  Description:Eccentricity. Used only when n=2
 35:  Long description:     Eccentricity. Used only when n=2
 36:
```

```
37:  Short name:           -d
38:  Long name:  --step-size
39:  Value type:float
40:  Default value: 0.01
41:  Variable name: h
42:  Description:Size of timestep
43:  Long description:     Size of timestep
44:
45:  Short name: -T
46:  Long name:  --end-time
47:  Value type:int
48:  Default value:1.0
49:  Value type:float
50:  Variable name:tend
51:  Description:Time to stop integration
52:  Long description:     Time to stop integration
53:
54:  Short name:-w
55:  Long name:  --window-size
56:  Value type:  float
57:  Variable name:wsize
58:  Default value:1.5
59:  Description:Window size for plotting
60:  Long description:
61:    Window size for plotting orbit. Window is [-wsize, wsize] for both of
62:    x and y coordinates
63:
64:  Short name:-e
65:  Long name:--ecc
66:  Value type:float
67:  Default value:0.0
68:  Variable name:ecc
69:  Description:Initial eccentricity of the orbit
70:  Long description:     Initial eccentricity of the orbit
71:
72:
73:  Short name:-v
74:  Long name:--velocity-scale
75:  Value type:float
76:  Default value:0.5
77:  Variable name:vscale
78:  Description:Scaling factor for the initial velocity
79:  Long description:
80:    Scaling factor for the initial velocity.
81:    positions and velocities are set from random vectors within
82:    spheres of radius 1 and vscale.
83:
84:END
85:
86:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
87:options=CLOP.new(optionstr,ARGV)
88:
```

```
 89:require "./vector3.cr"
 90:class Particle
 91:  property :m, :x, :v, :acc, :phi
 92:  def initialize(m : Float64=0, x : Vector3=Vector3.new(0,0,0),
 93:                 v : Vector3=Vector3.new(0,0,0))
 94:    @m = m; @x = x;  @v = v
 95:    @acc = Vector3.new(0,0,0); @phi=0.0
 96:  end
 97:  def self.random(m, rx, rv)
 98:    Particle.new(m.to_f, randomvector(rx), randomvector(rv))
 99:  end
100:  def calc_gravity(p,eps2)
101:    dr = @x - p.x
102:    r2inv = 1.0/(dr*dr+eps2)
103:    rinv = sqrt(r2inv)
104:    r3inv = r2inv*rinv
105:    @phi -= p.m*rinv
106:    p.phi -= @m*rinv
107:    @acc -= p.m*r3inv*dr
108:    p.acc += @m*r3inv*dr
109:  end
110:end
111:
112:class ParticleSystem
113:  property :particles, :eps2
114:  def initialize(eps2 : Float64 = 0.0 )
115:    @particles = Array(Particle).new
116:    @eps2 = eps2
117:  end
118:  def +(p : Particle)
119:    @particles.push p
120:    self
121:  end
122:  def self.random(n, rx, rv)
123:    ps=ParticleSystem.new
124:    m = 1.0/n
125:    n.times{ ps += Particle.random(m,rx,rv)}
126:    ps.adjust_cm
127:    ps
128:  end
129:  def self.twobody(x,v)
130:    ps=ParticleSystem.new
131:    n=2
132:    m = 1.0/n
133:    ps += Particle.new(m,x,v)
134:    ps += Particle.new(m,-x,-v)
135:    ps
136:  end
137:  def calc_accel
138:    @particles.each{|p|p.acc=Vector3.new; p.phi=0.0}
139:    @particles.each_with_index{|p,i|
140:      ((i+1)..(particles.size-1)).each{|j|
```

```
141:        p.calc_gravity(@particles[j], @eps2)
142:      }
143:    }
144:  end
145:  def inc_vel(h)
146:    @particles.each{|p|p.v += p.acc*(h)}
147:  end
148:  def inc_pos(h)
149:    @particles.each{|p|p.x += p.v*h}
150:  end
151:  def calc_cm
152:    sumx=Vector3.new
153:    sumv=Vector3.new
154:    summ=0.0
155:    @particles.each{|p|
156:      sumx += p.m*p.x
157:      sumv += p.m*p.v
158:      summ +=p.m
159:    }
160:    {sumx*(1/summ),sumv*(1/summ)}
161:  end
162:  def adjust_cm
163:    cmx,cmv = calc_cm
164:    @particles.each{|p|
165:      p.x -= cmx
166:      p.v -= cmv
167:    }
168:  end
169:end
170:  def randomvector(r)
171:    sqsum= r*r*2
172:    v=Vector3.new
173:    while sqsum > r*r
174:      v = Array.new(3){rand(r)*2-r}.to_v
175:      sqsum = v*v
176:    end
177:    v
178:  end
179:
180:
181:ENV["GKS_DOUBLE_BUF"]= "true"
182:
183:if options.n == 2
184:  ps = ParticleSystem.twobody([0.5*(1+options.ecc),0.0,0.0].to_v,
185:                              [0.0,0.5*sqrt((1-options.ecc)/(1+options.ecc)),0.0].to_v)
186:else
187:  ps =ParticleSystem.random(options.n,1.0,options.vscale)
188:end
189:ps.eps2= options.eps*options.eps
190:ps.calc_accel
191:wsize=options.wsize
192:setwindow(-wsize, wsize,-wsize, wsize)
```

```
193:
194:time=0
195:while time < options.tend-options.h/2
196:#  SymplecticIntegrators.leapfrog(ps,options.h)
197:  SymplecticIntegrators.yoshida4(ps,options.h)
198:  time += options.h
199:  clearws()
200:  box
201:  setcharheight(0.05)
202:  mathtex(0.5, 0.06, "x")
203:  mathtex(0.06, 0.5, "y")
204:  text(0.6,0.91,"t="+sprintf("%.3f",time))
205:  setmarkertype(4)
206:  setmarkersize(1)
207:  polymarker(ps.particles.map{|p| p.x[0]}, ps.particles.map{|p| p.x[1]})
208:  updatews()
209:end
210:c=gets
211:
```

です。とりあえず前のと違うところを順番に説明します。diff の出力だしますね。

```
gravity> diff particle0.cr particle1.cr
2a3
> require "./integratorlib.cr"
109d109
<
113,114c113,114
<   property :particles
<   def initialize()
---
>   property :particles, :eps2
>   def initialize(eps2 : Float64 = 0.0 )
115a116
>     @eps2 = eps2
136c137
<   def clear_gravity
---
>   def calc_accel
138,139d138
<   end
<   def calc_gravity(eps2)
142c141
<         p.calc_gravity(@particles[j], eps2)
---
>         p.calc_gravity(@particles[j], @eps2)
146,152c145,149
<   def leapfrog(h,eps2)
<     @particles.each{|p|p.x += p.v*h + p.acc*(h*h*0.5)}
<     @particles.each{|p|p.v += p.acc*(h*0.5)}
<     clear_gravity
```

```
<     calc_gravity(eps2)
<     @particles.each{|p|p.v += p.acc*(h*0.5)}
<     self
---
>   def inc_vel(h)
>     @particles.each{|p|p.v += p.acc*(h)}
>   end
>   def inc_pos(h)
>     @particles.each{|p|p.x += p.v*h}
186d182
< #ps = ParticleSystem.random(2,1.0,0.4)
193,194c189,190
< eps2= options.eps*options.eps
< ps.calc_gravity(eps2)
---
> ps.eps2= options.eps*options.eps
> ps.calc_accel
199,200c195,197
< while time < options.tend+options.h/2
<   ps = ps.leapfrog(options.h, eps2)
---
> while time < options.tend-options.h/2
> #  SymplecticIntegrators.leapfrog(ps,options.h)
>   SymplecticIntegrators.yoshida4(ps,options.h)
```

これだと、下にでているほうが新しいバージョンですね。最初の、新しいほう、えーと、全部行数は新しいほうでいきますが、3 行目は、integratorlib.cr にリープフロッグとかいれたので、それを require するようにしてます。 integratorlib.cr に追加したのは、

```
module SymplecticIntegrators
  extend self
  def leapfrog(s, h)
    s.inc_vel(h*0.5)
    s.inc_pos(h)
    s.calc_accel
    s.inc_vel(h*0.5)
  end
  D1 = 1.0 / (2-exp(log(2.0)/3))
  D2 = 1 - 2*D1

 def yoshida4(s,h)
    leapfrog(s,h*D1)
    leapfrog(s,h*D2)
    leapfrog(s,h*D1)
  end

  def  yoshida6(x,v,h,f)
    d = {0.784513610477560, 0.235573213359357,
         -1.17767998417887, 1.31518632068391};
    4.times{|i|leapfrog(s,h*d[i])}
    3.times{|i|leapfrog(s,h*d[2-i])}
```

```
    end
  end
```

です。積分公式は新しい名前のモジュールにして、前に作ったのと同じ名前ですが違う引数で違う動作にしてます。やることはあんまり変わらないです。なので、同じようなことをする関数が 2 種類ある、というのは変わらないです。でもまあ、その気になれば 9 章のプログラムもこっち使うようにはできるので、、、

赤木 ：そうねえ。まあ、前のと同じではできない、というところから始めたから、どうしてもそうなるわね。

学生 C : はい。その次、113-114、116 行目は、 eps2 を ParticleSustem のメンバー変数にしたのでこうなってます。相互作用計算する関数がこのクラスの中の calc_accel として定義されるので。calc_accel に Proc オブジェクトを渡すとかして中身変更できてもいいかもしれないですが、そんな変なのが必要とも思えないのでこうしてます。 137, 138 辺りは、元々clear_gravity と calc_gravity の 2 つあったのを calc_accel 1 つにしたからですね。 141 行目は eps2 がメンバー変数になった関係です。

146-149 行目は、元々 leapfrog があったところに inc_vel、inc_pos をいれてます。そもそもここから後はまだ話してないですね？

赤木 ：あ、そうね。

学生 C : その次の calc_cm と adjust_cm は、乱数で粒子ばらまいただけだと系の重心速度が 0 じゃなののので、時間が立つと系全体がどっかにいっちゃうので、そうならないように系の重心の位置、速度を 0 にしてます。

赤木 ：なるほど。これ、 x, v に同じことするじゃない？その辺もうちょっと工夫できなかった？

学生 C : えー、どうでしょう？例えば

```
summ = @particles.sum{|p| p.m}
sumx = @particles.sum{|p| p.m*p.x}
sumv = @particles.sum{|p| p.m*p.v}
```

とかですか？作者こっちが好きな感じですね。

赤木 ： そうね。もとのよりこっちのほうが、間違えないで書けてる可能性が高いし、いいと思うわ。あと、こっちの形なら、別に sumx とか sumv に一旦代入しなくても

```
invsumm = 1.0/@particles.sum{|p| p.m}
{@particles.sum{|p| p.m*p.x}*invsumm,
 @particles.sum{|p| p.m*p.v}*invsumm}
```

でいいわけ。これでもプログラムの意味はわかりにくくはならないでしょ？元の君のだと 9 行あったのが 3 行だし。

学生 C : そうですね。(なんか悔しい、、、)

赤木 ：あ、えーと、ごめんなさい、これは、格好良く書けるとかいうことじゃなくて、間違える可能性が少ない、というのが大事なの。1 から 10 まで合計します、というプログラムを

```
1 から 10 まで合計します
```

と書くのと

```
最初に sum を 0 にします
次に i を 1 から 1 から 10 まで変えながら
 sum に i を加えます
おしまい
```

みたいなのを書くのは手間も違うしどっかで間違えるかどうかも違うから。まあ、Crystal とかだと「次に i を 1 から 1 から 10 まで変えながら」のところが、

```
@particles.each{|p| ...}
```

みたいなのですむから、最初を忘れるとか最後の次までいっちゃうとかが割合起こりにくいけど、 C とか Fortran だとそういうことにも注意しないといけないから。

学生 C : えーと、次いっていいですか？

赤木 ：いいわよ。

学生 C : 181 行目の GKS なんたらはこれやるそアニメーションがチラツかないで動く、というやつです。GR 結構ドキュメントわかりにくくて、これ見つけるの苦労しました。

赤木 ：まあ大抵なんでもドキュメントってわかりにくいのよね、、、

学生 C : はい。で、183-188 は初期条件で、今まで作った関数を呼ぶだけです。まああと大体そうなので、リープフロッグ、あ、4 次公式使ってみてますね、それ呼んでるのが 197 行目です。あとは図を書くとかなので省略で。

赤木 ：で、これ動かすと何が起こるの？

学生 C :

と、単に

```
particle1
```

で実行すれば、2 粒子が円軌道で動くはずです。

```
particle1 -T 100
```

とかすれば沢山回ります。

```
particle1 -n 100 -s 0.01
```

とかすれば、なんか沢山粒子がでてうにょうにょ動きます。

10.4 シミュレーション結果の「検証」

赤木 ：動くのはいいとしてして、これちゃんと正しく計算できてる？

学生 C : と思いますが、、、

赤木 ：こういうのは思いますでは駄目で、根拠がいるのよ。だって、 シミュレーションしてこう答がでました、っていっても、それだけではそれが本当かどうかわからないでしょ？

学生 C : でも、元々解析解がなくて答がわからないからシミュレーションするんだから、その答が正しいかどうかってわからないんではないですか?

赤木 ：それはそうなんだけど、それでも、全然間違ってればわかるチェック、というのはあるの。例えば、古典力学系だから保存量はあるでしょ？

学生 C : 3 次元なので、エネルギー、全体の運動量、全体の角運動量ですね？

赤木 ：そう。だから、その辺のチェックは最低限して欲しいの。あ、あと、恒星系だと、「重力ポテンシャルエネルギーと運動エネルギーの比」が大事な量なの。これは「ビリアル比」っていって、自己重力系の定常状態だと変化しない、というのが証明できるの。

学生 C : すみません、定常状態ってなんですか？

赤木 ：分布関数が、、、あ、えーと、つまり、統計力学と同じで、星の数が無限に大きな極限を考えると、恒星系って、位置と速度の 6 次元空間の中での密度分布とみなせるでしょ？

学生 C : でしょって言われても、、、統計力学なら原子はみんな同じですが、星はひとつひとつ違うじゃないですか?統計力学って成り立たつんですか?

赤木 ：今は星は重力相互作用する質点だから、違うのは質量だけじゃない？そうすると、星の数が無限に多いなら、分布関数が質量にも依存するとして統計力学の式をたてればいいわけ。ただ、まずここで問題なのは、統計力学的な平衡、熱平衡ね、それじゃなくて、力学平衡というものなの。

学生 C : どう違うんですか?

赤木 ：統計力学的平衡はエントロピー最大なんだけど、力学平衡は単に分布関数が定常というだけね。普通の気体だと、力学平衡は圧力がつりあってる状態、静水圧平衡にあたるものね。静水圧平衡だと温度分布はあってもいいわけ。

学生 C： でも星は動くわけじゃないですか?

赤木 ：そう。だから、割合その辺違うのね。気体分子運動論から流体力学を導く時には、分子同士は頻繁に衝突するから、局所的には統計力学的に平衡だとして、局所的なマクロな物理量を導くんだけど、星同士は滅多にぶつからないから局所的にも大局的にも熱平衡にならないの。

で、今は、熱平衡は問題じゃなくて、その前の静水圧平衡とか力学平衡ね。この辺、ここで説明するとそれだけで何十時間かかかるから、作者の講義ノート[1]みといてね。3 回目くらいまででとりあえずいいから。

学生 C : はあ、、、、

赤木 ：で、以下、一応見たという仮定のもとで話をするけど、ビリアル比は自己重力系の定常状態だと -1/2 になるから、その数字もだしてね、というのが話ね。

学生 C： すみません、理屈わかってないですが計算はしてみます。

プログラムの変更したところをだします。

```
gravity> diff particle1.cr particle1a.cr
3a4
> require "./mathvector.cr"
44a46,62
>   Short name:           -D
>   Long name:  --diagnostics-interval
>   Value type:int
>   Default value: 5
>   Variable name: diaginterval
>   Description:Interval for diagnostics
>   Long description:     Interval for diagnostics
>
>   Short name:           -G
```

[1] http://jun-makino.sakura.ne.jp/kougi/stellar_dynamics/index.html

```
>   Long name:  --graphics-interval
>   Value type:int
>   Default value: 5
>   Variable name: graphicinterval
>   Description:Interval for graphics
>   Long description:     Interval for graphics
>                         No output if 0 or negative
>
76c94
<   Default value:0.5
---
>   Default value:1
150a169
>
152,160c171,173
<     sumx=Vector3.new
<     sumv=Vector3.new
<     summ=0.0
<     @particles.each{|p|
<       sumx += p.m*p.x
<       sumv += p.m*p.v
<       summ +=p.m
<     }
<     {sumx*(1/summ),sumv*(1/summ)}
---
>    invsumm = 1.0/@particles.sum{|p| p.m}
>    {@particles.sum{|p| p.m*p.x}*invsumm,
>     @particles.sum{|p| p.m*p.v}*invsumm}
168a182,187
>
>   def energies
>     ke = @particles.sum{|p| p.v*p.v*p.m}*0.5
>     pe = @particles.sum{|p| p.phi*p.m}*0.5
>     [pe+ke, pe, ke].to_mathv
>   end
190a210
> e0=ps.energies
194a215,216
> istep=0
> pp! options
196,197c218,219
< #  SymplecticIntegrators.leapfrog(ps,options.h)
<   SymplecticIntegrators.yoshida4(ps,options.h)
---
>   SymplecticIntegrators.leapfrog(ps,options.h)
> #  SymplecticIntegrators.yoshida4(ps,options.h)
199,208c221,239
<   clearws()
<   box
<   setcharheight(0.05)
<   mathtex(0.5, 0.06, "x")
<   mathtex(0.06, 0.5, "y")
```

```
<   text(0.6,0.91,"t="+sprintf("%.3f",time))
<   setmarkertype(4)
<   setmarkersize(1)
<   polymarker(ps.particles.map{|p| p.x[0]}, ps.particles.map{|p| p.x[1]})
<   updatews()
---
>   istep+=1
>   if options.graphicinterval > 0 && (istep % options.graphicinterval) == 0
>     clearws()
>     box
>     setcharheight(0.05)
>     mathtex(0.5, 0.06, "x")
>     mathtex(0.06, 0.5, "y")
>     text(0.6,0.91,"t="+sprintf("%.3f",time))
>     setmarkertype(4)
>     setmarkersize(1)
>     polymarker(ps.particles.map{|p| p.x[0]}, ps.particles.map{|p| p.x[1]})
>     updatews()
>   end
>   if (istep % options.diaginterval) == 0
>     e=ps.energies
>     de=e - e0
>     print sprintf("T= %e E, de, Q = %e %e %e\n",time, e[0], de[0], e[2]/e[1])
>   end
>
210c241,242
< c=gets
---
> c=gets if options.graphicinterval > 0
>
```

新しいので 4 行目は、mathvector も使うようにしたのでいれてます。46-61 行は、それぞれ、エネルギーとかの値を何ステップに 1 度書くかと、絵を何ステップに 1 度書くかです。時間ステップ毎に書くと遅いのとやたら沢山出力でるので。

赤木 ：なるほど。

学生 C : 次は、さっきの sum のところですね。まあ、こっちのほうがわかりやすいですね、、、、energies も関数作って、同じような感じで計算します。 pe にも 0.5 かけてるのは、そうしないと 2 重になるからです。

エネルギーは 209 行で最初に値を計算して、230 行あたりで新しい値だして誤差とビリアル比もだしますか。何ステップかに 1 度にするために、 istep という変数でステップ数えておいて、それとオプションで指定した値を比べます。絵書くのも同様ですが、こちらは 0 だったら書かないようにしてみました。

赤木 ：実行するとどんな感じ？

学生 C :

```
gravity> particle1a -n 100 -s 0.01 -T 10 -d 0.004 -D 250 -G 0 -v 1
options # => #<CLOP:0x7fed0d806f60
 @diaginterval=250,
 @ecc=0.0,
```

```
 @eps=0.01,
 @graphicinterval=0,
 @h=0.004,
 @help=false,
 @n=100,
 @tend=10.0,
 @vscale=1.0,
 @wsize=1.5>
T= 1.000000e+00 E, de, Q = -2.888811e-01 2.213044e-04 -5.454667e-01
T= 2.000000e+00 E, de, Q = -2.891792e-01 -7.672365e-05 -4.797439e-01
T= 3.000000e+00 E, de, Q = -2.889417e-01 1.607362e-04 -4.424230e-01
T= 4.000000e+00 E, de, Q = -2.889795e-01 1.229477e-04 -4.456188e-01
T= 5.000000e+00 E, de, Q = -2.889737e-01 1.287330e-04 -4.739901e-01
T= 6.000000e+00 E, de, Q = -2.890414e-01 6.104928e-05 -5.508729e-01
T= 7.000000e+00 E, de, Q = -2.889936e-01 1.088696e-04 -5.389175e-01
T= 8.000000e+00 E, de, Q = -2.889645e-01 1.379321e-04 -5.186624e-01
T= 9.000000e+00 E, de, Q = -2.889936e-01 1.088195e-04 -4.813272e-01
T= 1.000000e+01 E, de, Q = -2.890098e-01 9.258578e-05 -5.015148e-01
```

```
gravity> particle1a -n 100 -s 0.01 -T 10 -d 0.002 -D 500 -G 0 -v 1
options # => #<CLOP:0x7fe01efc5f60
 @diaginterval=500,
 @ecc=0.0,
 @eps=0.01,
 @graphicinterval=0,
 @h=0.002,
 @help=false,
 @n=100,
 @tend=10.0,
 @vscale=1.0,
 @wsize=1.5>
T= 1.000000e+00 E, de, Q = -2.943553e-01 -1.546040e-07 -5.217045e-01
T= 2.000000e+00 E, de, Q = -2.943546e-01 5.082690e-07 -5.017925e-01
T= 3.000000e+00 E, de, Q = -2.943558e-01 -6.141378e-07 -5.053036e-01
T= 4.000000e+00 E, de, Q = -2.943555e-01 -3.326063e-07 -4.985450e-01
T= 5.000000e+00 E, de, Q = -2.943533e-01 1.850267e-06 -4.792927e-01
T= 6.000000e+00 E, de, Q = -2.943537e-01 1.486430e-06 -5.096690e-01
T= 7.000000e+00 E, de, Q = -2.943543e-01 8.110818e-07 -5.008133e-01
T= 8.000000e+00 E, de, Q = -2.943515e-01 3.646515e-06 -5.052206e-01
T= 9.000000e+00 E, de, Q = -2.943517e-01 3.419905e-06 -4.977529e-01
T= 1.000000e+01 E, de, Q = -2.943595e-01 -4.330040e-06 -5.512526e-01
```

```
gravity> particle1a -n 100 -s 0.01 -T 10 -d 0.001 -D 1000 -G 0 -v 1
options # => #<CLOP:0x7f3262562f60
 @diaginterval=1000,
 @ecc=0.0,
 @eps=0.01,
 @graphicinterval=0,
```

```
 @h=0.001,
 @help=false,
 @n=100,
 @tend=10.0,
 @vscale=1.0,
 @wsize=1.5>
T= 1.000000e+00 E, de, Q = -3.271104e-01 -1.472251e-05 -5.417722e-01
T= 2.000000e+00 E, de, Q = -3.270930e-01 2.655497e-06 -5.271642e-01
T= 3.000000e+00 E, de, Q = -3.270955e-01 1.262471e-07 -4.886674e-01
T= 4.000000e+00 E, de, Q = -3.270953e-01 3.309608e-07 -4.421474e-01
T= 5.000000e+00 E, de, Q = -3.270961e-01 -4.265696e-07 -4.931851e-01
T= 6.000000e+00 E, de, Q = -3.270956e-01 8.099669e-09 -5.242083e-01
T= 7.000000e+00 E, de, Q = -3.270957e-01 -5.872798e-08 -4.708640e-01
T= 8.000000e+00 E, de, Q = -3.270952e-01 4.249327e-07 -5.243484e-01
T= 9.000000e+00 E, de, Q = -3.270984e-01 -2.711427e-06 -5.150729e-01
T= 1.000000e+01 E, de, Q = -3.270954e-01 2.347527e-07 -4.830604e-01
```

```
gravity> particle1a -n 100 -s 0.01 -T 10 -d 0.0005 -D 2000 -G 0 -v 1
options # => #<CLOP:0x7fdc01801f60
 @diaginterval=2000,
 @ecc=0.0,
 @eps=0.01,
 @graphicinterval=0,
 @h=0.0005,
 @help=false,
 @n=100,
 @tend=10.0,
 @vscale=1.0,
 @wsize=1.5>
T= 1.000000e+00 E, de, Q = -2.979887e-01 -2.131936e-08 -5.243235e-01
T= 2.000000e+00 E, de, Q = -2.979887e-01 -3.427204e-09 -4.971522e-01
T= 3.000000e+00 E, de, Q = -2.979887e-01 -4.936541e-08 -4.802587e-01
T= 4.000000e+00 E, de, Q = -2.979887e-01 -7.977912e-08 -4.946278e-01
T= 5.000000e+00 E, de, Q = -2.979887e-01 1.180453e-08 -4.853171e-01
T= 6.000000e+00 E, de, Q = -2.979887e-01 -2.729549e-08 -5.223997e-01
T= 7.000000e+00 E, de, Q = -2.979887e-01 -7.352084e-08 -5.152769e-01
T= 8.000000e+00 E, de, Q = -2.979891e-01 -4.610489e-07 -4.924154e-01
T= 9.000000e+00 E, de, Q = -2.979889e-01 -2.270968e-07 -5.000359e-01
T= 1.000000e+01 E, de, Q = -2.979901e-01 -1.418567e-06 -5.185928e-01
```

すみません、エネルギーしかみてないですが、一応タイムステップ小さくするとそれないに小さくなっていて大丈夫ではないかと、、、

赤木 ：その辺は読者の練習問題ね。

10.5 課題

1. 粒子数 100 程度で、時刻 10 程度まで積分した時のエネルギーの誤差の絶対値の最大値を、タイムステップの関数としてプロットしてみよ。ソフトニングは 0.01 で固定でよいが、変化させたらどのように変わるかも余力があれば調べてみよ。

2. 全運動量、全角運動量について、保存精度を調べよ。
3. 4 次シンプレクティック公式も選択できるようにプログラムを拡張し、課題 1, 2 についてどうなるか調べよ。

10.6 まとめ

1. シンプレクティック積分公式を、ハミルトン力学系を表現するクラスに対して一般的に適用できる形のライブラリとして作成した。
2. 重力多体問題を例に、上の公式を適用して系の性質を学んだ。
3. 重力多体問題では、ソフトニングを導入してポテンシャルの発散をふせいだ。

10.7 参考資料

http://jun-makino.sakura.ne.jp/kougi/stellar_dynamics/index.html|

Chapter 11

ハミルトニアン分割

赤木 ： 前回、多体問題をやったわけだけど、重力ソフトニングっていうのをいれたじゃない？

学生 C : はい。そうしないと上手くいかないからとかで。あんまりそれでいいのかどうかよくわかってないですが。

赤木 ： まあその、簡単にいっちゃうと、いい時もあるけど駄目な時もあるのね。例えば、星の数がすごく少ない、3 とか 10 とかだと、なんか全然違いそうでしょ？2 つの星が近づいて、お互いの重力で軌道変えるんだけど、そのたびに曲がりかたが違うからちょっと時間たつと全然変わっちゃうわね。あと、例えば 3 体でやってみるとわかるけど、連星ができることがあるでしょ？3 つがぐるぐる動いているうちに、1 つがはねとばされて、あとの 2 つが結合状態になるのね。これ、最終的にどうなるかはもちろんソフトニングとかいれると全然変わるから。

逆にいうと、星とダークマターだけの楕円銀河とかで、星の数もすごく多い時には、星は銀河全体の重力うけて運動するから、近くの星の影響をそんなにうけなくて、ソフトニングがあっても軌道そんなにすぐにはかわらないわけ。

学生 C : そうすると、銀河とかならこれでいい、ということですか?

赤木 ： 楕円銀河でガスがなければそうなんだけど、我々の銀河系みたいな、ガスがあって円盤がある銀河だとそうでもないのね。というのは、今でも温度が低いガスが重力で集まって、そこで星ができてるわけ。ガスが集まってるのを分子雲といって、星ができているところを星形成領域というんだけど、今の銀河系 d も数百個とかの星ができてつつある星形成領域とかあるの。

そうすると、新しくできた星は、それらが自己重力で集まった星団になってるのね。これは星の数少ないから、銀河全体は沢山星があっても星団で生まれた星がどうなるかを計算しようと思ったらソフトニングとか使えないわけ。

学生 C : じゃあどうするんですか?

赤木 ： というのが今日の話ね。こういう、星が集まった星団とか銀河とかを計算シミュレーションしたい、というのはわりと昔からやられていて、1960 年くらいにもう最初の論文があるのね、で、色々発展があったんだけど、基本的な考え方は 3 つなの。

1. 時間刻みを可変にする
2. 時間刻みを星毎ににバラバラに可変にする
3. 二つの星の間の重力ポテンシャル自体を、複数の成分に分けてそれぞれバラバラに時間積分する。

学生 C : えーと、、、

赤木 ： 今回やってみて欲しいのは最終的には 3 つめなんだけど、基本的な考え方とししては最初の 2 つもわかっておく必要があるから、簡単にに説明するわね。

11.1 従来の考え方

赤木 ： まず、「時間刻みを可変にする」っていうのは、どっかで 2 つの星が近づいて、今のタイムステップで上手く計算できなくなりそうなら、タイムステップ短くすればいい、というものね。常微分方程式の数値解法の研究としては、どういうふうに時間刻みを変えればいいか、というのはすごく大事なことで、そういうのがはいった実用的なアルゴリズムや数値計算パッケージは一杯あるの。

例えば Python の数値計算用のパッケージに scipy ってのがあって、この中に odeint という常微分方程式の初期値問題用のライブラリがあるのね。これは ODEPACK っていう、1980 年代に Fortran77 で書かれたライブラリを中で使ってて、これは普通の問題、 non-stiff っていうんだけど、には時間刻み可変の ABM 公式、 stiff な問題には BDF を使うのね、 ABM 公式のうち A の部分は 19 世紀に、、、

学生 C : あの、すみません、日本語で話を、、、

赤木 ： あー、ごめんなさい。例えば星が N 個ある星団で、みんなが勝手に動いているとして、半径 1、全質量 1、重力定数 1 で、速度も 1 ぐらいだとしましょう。隣の星までの距離はオーダーとしては $N^{-1/3}$ だから、普通の星はそれくらいの時間刻みで積分すればいいわけね。でも、実際やってみたら起こったみたいに、時々2 つの星が重力でどんどん近づいて、また離れていく「近接遭遇」が起こるから、そういうのをちゃんとつかまえるように時間刻み短くすればいいだろうというわけ。

例えばルンゲクッタ公式だと、1 ステップ進めるたびに時間刻みかえても大丈夫だから、そういうことができるように工夫された公式があるの。つまり、積分誤差を推定できるようにして、その推定された誤差がこちらで設定した値を超えないようにするわけ。

一番簡単には、まずはある時間刻みで 1 ステップ積分して、また、元の値から、時間刻み半分にして 2 ステップ積分したら、こっちのほうが正確なはずでしょ？だから、この 2 つ比べると誤差が大体わかるから、その誤差があまり大きくならないように、次のステップを調整するわけ。

学生 C : なるほど。それでよさそうな気がしますが、駄目なんですか?

赤木 ： これは、星団とか銀河だと駄目なの。理由は 2 つあって、一つは、星の数が増えると、平均の距離は $N^{-1/3}$ でも、「一番近いペア同士」の距離はもっと小さいのね。これは確率の問題だけど、$N^{-2/3}$ で小さくなるはず。だから、沢山の星の中でたまたま 2 つが近いだけで、全部の星を無駄に短いステップで積分することになるわけね。

学生 C : でも、数値積分ってそういうものじゃないんですか?

赤木 ： そうじゃない、っていう話をこれからするからもうちょっとまって。もうひとつの問題は、銀河系でも球状星団でもそうなんだけど、密度構造があるのね。つまり、中心のほうが星の数密度が高いの。銀河だと、銀河中心にブラックホールがあるけど、それ別にしても太陽から内側くらいだと大体半径の 2 乗に反比例して星というか質量密度が上がるのね。

学生 C : 2 乗に反比例だと、、、質量自体だと半径に比例ですね？

赤木 ： そう。そうすると、例えば円軌道で回ってる星の速度は半径でどう変わるかしら？

学生 C : えーと、重力は質量に比例で距離の 2 乗に反比例だから、半径に反比例ですね。速度を v, 半径 r として遠心力は v^2/r でこれが $1/r$ に比例と、速度一定ですか?

赤木 ： そう、我々の銀河系は大体そうなってて、半径どこでも回転速度は 240km/s なの。この値自体は銀河の観測が精密になると段々変わるんだけど、一定、というのはあんまり変わらないはず。

学生 C : えーと、で、なんの話でしたっけ？

赤木 ： つまり、速度が同じで星の密度が高いと、タイムステップ短くしないといけないでしょ？な

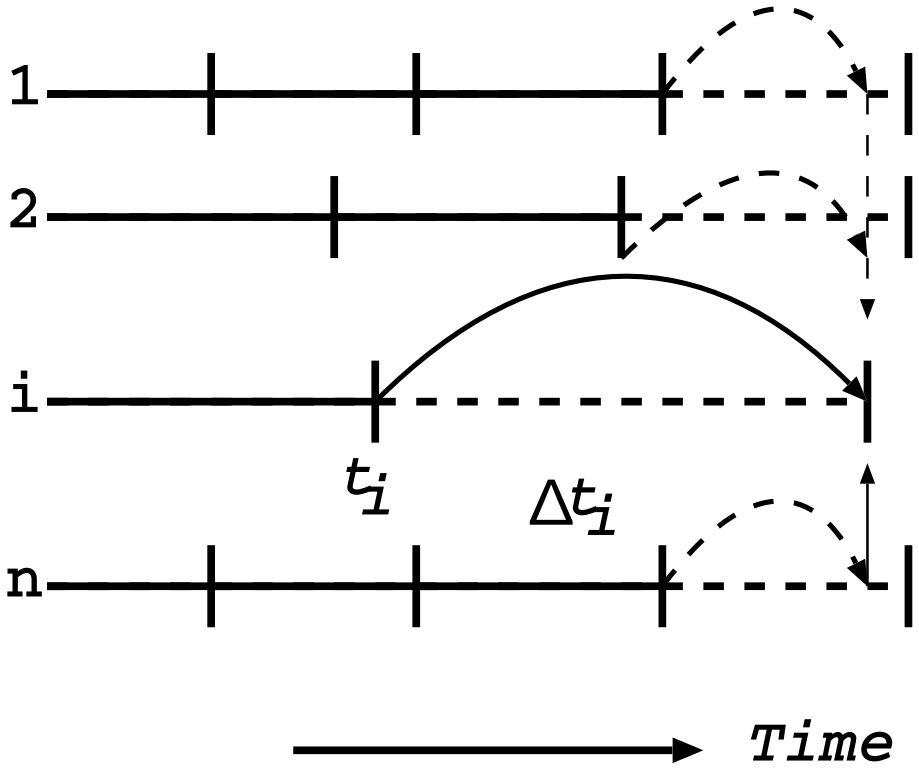

Figure 11.1: 独立時間刻みの考え方

ので、星団でも銀河でも、中心に近い星ほど短い刻みが必要だから、全体を中心で必要な短い時間刻みで積分しないといけないことになるのね。

我々の銀河系だと、中心に太陽質量の 400 万倍の質量の超巨大、天文学者の用語で英語で supermassive っていうんだけど、まあだから超大質量かな、ブラックホールがあって、見える星で、15 年周期でそのブラックホールの周りを回ってるのがあるのね。ちょっと余談だけど 2020 年 4 月 16 日にプレスリリースがあって、その星の近点移動が観測できて、一般相対性理論の予言と 10%くらいの精度であったとのことで、移動自体は 1 周期で 12 分だったかな、それくらいですごく大きいんだけど、それが地上の望遠鏡で観測できたのもすごいわよね。

と、それはともかく、15 年で 1 周期の星を積分する時間刻みで太陽みたいな 2 億年周期で銀河内運動する星の軌道積分をしたら計算おわらないでしょ？

学生 C：と思いますが、じゃあどうするんですか?

赤木 ：1960 年代以降の基本的な考え方は、「独立時間刻み」というものね。要するに、それぞれの星に、「自分の時間」と「自分の時間刻み」をもたせるの。図 11.1 みたいな感じ。

これ、それぞれの星が、今の自分の時刻と次の時刻というのをもってて、次の時刻が一番小さい星を積分するのね、積分するには、少なくとも新しい時刻で他の星からの重力を計算する必要があるわけで、それは実際に他の星をそこまで積分はできないから場所を「予測」するの。で、他の星の予測した位置から自分の予測した位置への重力を計算して、それ使って自分の位置・速度は計算しなおして、で、新しい「次の時刻」を決めるわけ。そうしたら、1 個積分すると次の時刻が一番小さい星が変わるから、それを次々に積分していくのね。

学生 C：えーと、そんなの本当に上手くいくんですか?大体その予測ってどうするんですか?

赤木 ： 1960 年代から 90 年代くらいまで使われてたのは、3 ステップ前くらいまでの加速度をとっ

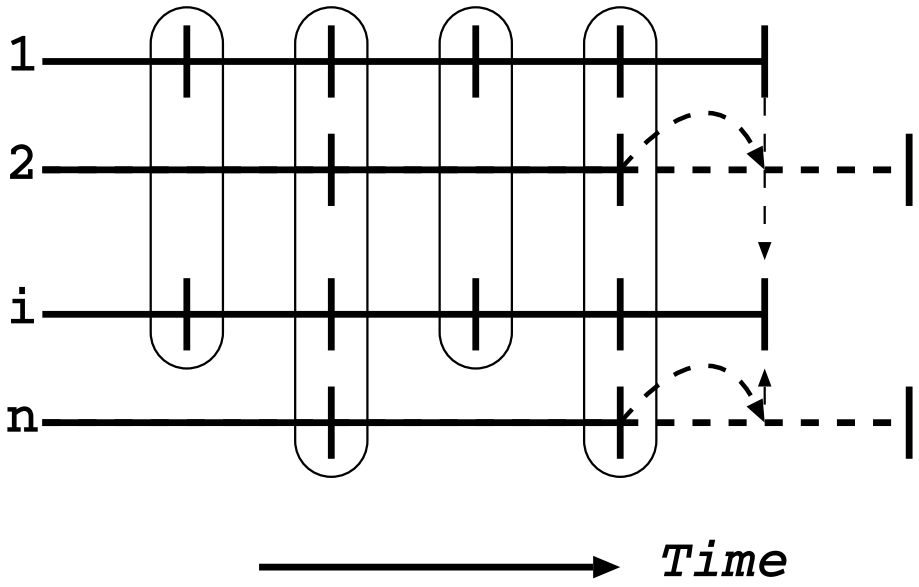

Figure 11.2: ブロックステップの考え方

ておいて、それぞの時刻ではその値になるようなの時間の 3 次多項式を作って、それで予測する、という方法ね。これはさっきの君のいうところの日本語じゃないところの、ABM 公式の A のところにあたる方法なの。

学生 C : 補間多項式っていうやつですね。聞いたことあります。

赤木 : そう。細かいことをいうと、加速度そのものじゃなくて、 divided difference っていう、なんかややこしい量を計算してとってもおくと、補間計算が簡単になるのでそっち使うの。考え方としてはニュートン補間とかニュートン・コーツ補間ね。

学生 C : はあ。

赤木 : で、90 年代くらいまでは、といったのは、その辺からもうちょっと簡単な方法をみんな使うようになったから。これでは、重力加速度自体の時間微分を 2 階微分まで直接計算して、それでテイラー展開を作ってそれをそのまま計算する方法。常微分方程式の数値解法としては、そういう、導関数の微分を使う方法はもちろん昔から知られていたんだけど、重力多体問題に実際に実用的な形で使ったのは作者氏と、元々の独立時間刻みを発明した Aarseth 先生の共著の論文ね。

学生 C : へえ。偉い人なんですね。

赤木 : うーん、どうかしら？あと、2 回導関数までだと 4 次精度なんだけど、6 次と 8 次は似鳥さんっていう人が論文にしてて、あとそれ以上も論文にはしてないけどセミナーで発表した資料はあって、それ引用して論文かいてた人がいたわ。

学生 C : はあ。

赤木 : あともうちょっと独立時間刻みの話ね。最近のスパコンってみんな並列計算機だから、こういう、1 個星選んでそれを積分、では効率でないの。これは 1980 年代くらいから問題になりはじめて、なのでブロックステップというのが発明されたのね。これは 11.2 にある通りで、時間刻みを 2 のべき乗に合わせておくと、時間や時間刻みの値が同じになる星がでてくるから、それらは並列に計算できる、というわけ。

学生 C : そりゃ、元々よりましそうですが、どれくらい上手くいくんですか？

赤木 : まあ理論的には、さっきの話で一番短いステップが $N^{-2/3}$ くらいで他が平均的に $N^{-1/3}$ なら、一番短いステップで進めていった時に $N^{1/3}$ ステップで N 粒子だから、平均的には $N^{2/3}$ 粒

子よね。結構沢山。

でも、中心密度が高いとかになるとあんまり上手くいかないのね。

学生 C : そうですよね。もっとうまい方法はないんですか？

11.2 新しい考え方

赤木 : その辺をこれからね。ただ、今のところ世界的にも、星団でも銀河でも、この独立時間刻みが基本的に使われている方法なの。いわゆる state of the art というのね。だから、これから話をするのは、もちろん論文とかあって上手くいくのもわかってるけど、世界で広く使われている、というわけではまだないのね。

学生 C : それ、大丈夫なんですか？

赤木 : まあ、その辺ちょっと調べるのもやってみようと思ってるわけ。考え方は単純なのよ。といってもいきなり重力多体問題だと説明もちょっとややこしいので、まず 2 体問題を例にして考えることにするわね。

話をリープフロッグ公式に戻すと、

$$\begin{aligned} x_{i+1} &= x_i + hv_i + (h^2/2)f(x_i) \\ v_{i+1} &= v_i + (h/2)[f(x_i) + f(x_{i+1})] \end{aligned} \tag{11.1}$$

という形で書いたけど、もともと 2 つの公式を組合せてた、ということを思い出すと、

$$\begin{aligned} v_{i+1/2} &= v_i + (h/2)f(x_i) \\ x_{i+1} &= x_i + hv_{i+1/2} \\ v_{i+1} &= v_{i+1/2} + (h/2)f(x_{i+1}) \end{aligned} \tag{11.2}$$

とも書けるわけね。これって、

1. 速度を今の位置の加速度で時間刻みの半分だけ進める
2. 位置を上の速度で時間刻み分進める。
3. 速度を今の位置の加速度で時間刻みの半分だけ進める

となるじゃない？

学生 C : そうですが、だからどうと？

赤木 : ここで、微分方程式が、

$$\begin{aligned} \frac{dv}{dt} &= f(x) + g(x) \\ \frac{dx}{dt} &= v \end{aligned} \tag{11.3}$$

と、加速度が 2 つの関数の和だとするわね。で、 f は計算が大変だけどあんまり速くは変わらない、g はそれほど大変じゃないけど f より急に変わる、あるいはずっと大きくて正確に計算する必要がある、という場合を考えると、

学生 C : えーと、でも、2 体問題では重力相互作用って 1 つの関数じゃないですか？この話どう関係するんですか？

赤木 : そうなんだけど、もうちょっと待って。とりあえず、2 つだとすると、こういう手順もありえるわけ。

1. 速度を今の位置の f で時間刻みの半分だけ進める
2. g だけ使って次の時刻まで数値積分する
3. 速度を今の位置の加速度で時間刻みの半分だけ進める

例えば、太陽系の惑星とかだとこれ上手くいくのね。 f を惑星間の重力相互作用、 g を太陽重力とすると、惑星は沢山あるから f の計算は大変で、g は数値計算といっても解析解もあるからそっちつかうこともできるし、高精度の公式で数値積分してもいいわけ。

学生 C : それはなんとなく上手くいきそうですが。惑星間相互作用は太陽からの重力に比べてずっと小さいわけですよね？

赤木 : そう。この方法は MIT の Jack Wisdom と Matthew Holman が 1991 年の Symplectic maps for the N-body problem っていう論文で提案して、すごく広く使われるようになったの。

でね、それとちょっと似たことを、でも太陽重力と惑星間みたいな違うものの和ではなくて、ケプラー問題みたいな、1 つなんだけど、例えば離心率がすごく大きくて、太陽に近い時と遠い時で時間ステップ変えたい、みたいな時に適用したのが Skeel と Biesiadecki の 1994 年の、Symplectic Integration with Variable Stepsize っていう論文なのね。どういう考え方かというと、太陽からの重力を無理矢理 2 つとかそれ以上の成分に分けるのね。

つまり、重力加速度は

$$\frac{d\mathbf{v}}{dt} = \mathbf{a}(\mathbf{x}) = -GM\frac{\mathbf{x}}{|x|^3} \tag{11.4}$$

として、適当な関数 $K(x)$ をもってきて、

$$\begin{aligned} \mathbf{f}(\mathbf{x}) &= K(|x|)\mathbf{a}(\mathbf{x}) \\ \mathbf{g}(\mathbf{x}) &= [1-K(|x|)]\mathbf{a}(\mathbf{x}) \end{aligned} \tag{11.5}$$

とするわけ。で、前に書いた、リープフロッグで f を積分して、但し途中では g を積分するのね。$K(x)$ が、 x が小さい、つまり太陽に近いところでは 0 になるようにすれば、太陽に近いところでは g の、なんか正確な方法での積分、遠いところでは逆に f だけの、普通のリープフロッグでの積分になって、リープフロッグでは太陽に近くなると上手くいかない、という問題が回避されるわけ。ここでは g は 0 だから、計算しなくていいのね。

学生 C : なんか無駄に面倒じゃないですか?独立時間刻みとか普通の可変時間刻みでは駄目なんですか?

赤木 : まあだから、少なくとも重力多体系ではこれすごく得なの。なぜかというと、どこかで 2 つの粒子がすごく近づいいたとして、その 2 つの軌道とその 2 つの間の重力だけ細かく計算されるのね、他の粒子は全く影響うけなくてすむわけ。だから、独立時間刻みだと、動かす粒子が 2 個しかなくても他の全部の粒子からの力計算してたけど、こっちだと 2 個の間だけでいいわけ。

もうちょっというと、あちこちで同時とかちょっと違う時間に近接遭遇がおきても、それぞれちょっとづつ計算時間増えるだけだから、あんまり全体に影響しないのね。

これは、並列計算機使うとか、あと重力の計算法工夫して減らすとかの時にすごくメリット大きいのね。ちょっと減らすほうの話をすると、前に作ってもらった重力多体問題のプログラムだと、運動方程式の通りに 1 つの粒子は他の全粒子からの重力を受けるとしたじゃない？

学生 C : はい。だって、実際にそうですよね？

11.3 遠距離力の高速計算 1 FFT と球面調和関数展開法

赤木 : そうなんだけど、もうちょっと上手く計算できないか、という話ね。伝統的にというか、1970 年代くらいに考えられた方法は 2 つあって、一つは高速フーリエ変換を使う方法ね。

物理数学で偏微分方程式はやった？

学生 C : ええ、熱伝導とか波動方程式とか。

赤木 : ポアソン方程式ってやった？

学生 C : あんまり記憶がないんですが、、、

赤木 : ラプラス方程式は？

学生 C : そっちは記憶があるような気がします。

赤木 : あらら、電磁気でマクスウェルの方程式は？

学生 C : えーと、やったはずですね。

赤木 : それに、電場の方程式ってのがあるでしょ？

$$\nabla \mathbf{D} = \rho \tag{11.6}$$

っていうやつ。これ、静電ポテンシャル ϕ を使うと

$$\begin{aligned} \nabla^2 \phi &= -\rho \\ \mathbf{D} &= -\nabla \phi \end{aligned} \tag{11.7}$$

になる、ってのはやったでしょ？

学生 C : そういわれるとそんな気がしてきました。

赤木 : これで、 ρ は電荷密度で、 ϕ は静電ポテンシャルで、$\mathbf{D}$ は電場で、そこに電荷 q があると $\mathbf{D}q$ の力を受けるわけね、これは、電荷が 2 つあると距離の二乗に反比例する力をお互いに及ぼす、というのと同じことになってるのはわかるわね？

学生 C : あんまりわかってないですが、、、えーと、そうなるんですか?

赤木 : そうなるの。で、重力相互作用と静電相互作用って、質量と電荷とで力の種類は違うけど式の形は同じでしょ？だから重力場と重力ポテンシャルも、静電場の方程式と同じように質量密度に対するポアソン方程式を解けば求められるのね。なので、ポアソン方程式を解くのにフーリエ変換を使う、というのがこの方法の考え方で、具体的には、質量密度を、空間を格子に切って、各点での値での値と思うことにして、それを数値的にフーリエ級数展開というかフーリエ変換して、フーリエ級数のほうで係数掛けるとポアソン方程式解けるでしょ？それで逆フーリエ変換してポテンシャルにするわけ。力も逆フーリエ変換でもいいし、精度ちょっと落ちるけどポテンシャルを数値微分でもまあ求めるわね。

高速フーリエ変換は、格子点の数を N として計算量が $O(N \log N)$ ですむから、まともに粒子間相互作用を計算すると $O(N^2)$ になるのに比べてずっと速く計算できるの。

学生 C : えーと、でも、フーリエ変換だと周期境界になってなんか変じゃないですか?

赤木 : そこはなんとかする方法があるの。私よく知らないけど。

学生 C : そうなんですか。あと、これ、2 つの粒子が近づいたとかあったらどうするんですか?

赤木 : 基本的にはどうにもならなくて、初めてから粒子の質量は空間的に広がっている、と思うのが 1 つの方法で、上にでてきた K みたいなのを使って相互作用 2 つにわけて、近くは直接計算、遠くだけフーリエ変換で、というのがもうひとつの方法ね。これは particle-particle-particle-tree (P^3M) 法っていわれていて 1980 年代に提案されたの。

学生 C : よさげな方法にみえますが、どうなんでしょう？

赤木 : いいんだけど、天文のシミュレーションだと、中心の密度がどんどん上がるとか、銀河だとずーっと外側まで広がってるとかあって、あんまり上手くいかないの。

学生 C : あ、あともうひとつあるっていってませんでしたっけ？

赤木 : あるけど、これも同じような問題があるので簡単にね。物理数学で球面調和関数展開ってやった？量子力学とかでは？

学生 C : やったのかもしれないですがえーと、、、

赤木 : これもやっぱりラプラス方程式とかポアソン方程式と関係するんだけど、ラプラス方程式って境界条件与えると解決まるでしょ？2 次元で、円板領域だと、境界は円で、円周上での周期関数が境界条件で、その固有関数を求めると $\sin(n\theta)r^n$ みたいなのがでてくるでしょ？

学生 C : そうだったような気がします。

赤木 : で、円板の外側領域だと $\sin(n\theta)r^{-n}$ で書けて、これを外部解っていうんだけど、これ、円板の中の任意の粒子分布について、それが作る電場とか重力場の角度方向のフーリエ変換をすれば、無限遠方までの解が求められることになるでしょ？

学生 C : なんか騙されてる気がしますが、そうなんですか?

赤木 : 数学としてはそうなるでしょ？でね、2 次元で円板だとフーリエ変換でいいけど、3 次元だと球面になるじゃない？なので、球面上にフーリエ変換を拡張したものが必要で、それが球面調和関数なのね。具体的な形は、3 次元極座標を (r, θ, ϕ) で書いた時に、θ の三角関数と ϕ のルジャンドル陪関数っていう直交多項式系の積で書けるんだけど、ここはまあ具体的な形はよくて要するにそういう、円周上のフーリエ級数展開を球面上に拡張したものと思って。これを多重極展開っていうの。

そうすると、ある半径での重力ポテンシャルは、その内側の粒子が作る重力場の外部解と、その外側の粒子が作るポテンシャルの内部解の合計になるから、それを順番に、あるいはなんか上手く並列化して計算すればいいわけ。これも、じゃあ展開項数どこまでとるの問題というのがあって、楕円銀河みたいな丸いのならこれでいいけど、、、みたいな話で最近あんまり使われてないかも。

11.4 遠距離力の高速計算 2 ツリー法

学生 C : もうちょっと新しい方法とかはないんですか?

赤木 : もちろんあって、ちょっとその話ね。今割合よく使われる方法の 1 つはツリー法というものね。

3 次元面倒なので 2 次元で説明するわね。図 11.3 みたいに、なにか粒子分布があったとして、それ全体がはいる正方形を、再帰的に 4 個に分けるのを、それぞれ正方形に粒子が 1 または 0 個になるま繰り返してできる木構造を 4 分木、3 次元で立方体だと 8 分木というのね。これ使うと、例えば右下の粒子から、左上の、全体の 1/4 の面積の正方形をみると、割合離れてるから、そうすると多重極展開で重力評価してもあんまり誤差大きくないかもしれないわけ。つまり、この木構造の各節点に対応する粒子分布について全部多重極展開を求めておけば、それらをを上手く使って各粒子への力を高速に求められる、というのがツリー法ね。

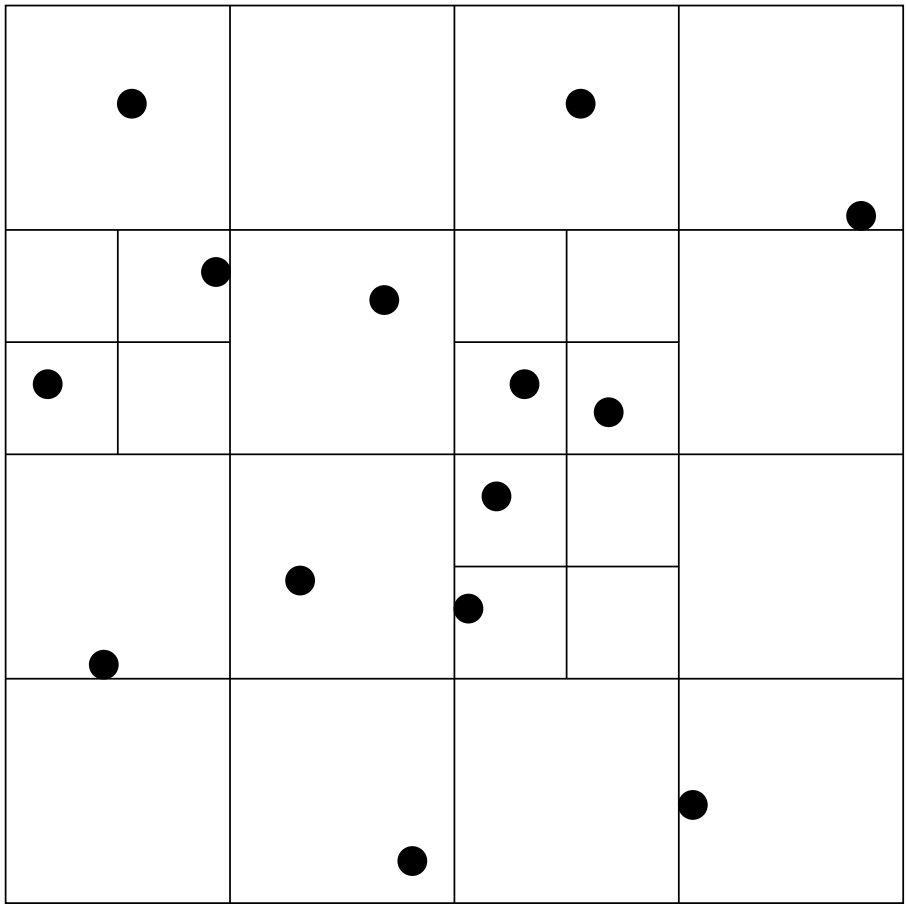

Figure 11.3: ツリー法の考え方。2 次元の場合。

上手く使う、というのが実は簡単で、ツリーを上から辿っていって、精度的に大丈夫ならそこで多重極展開を評価する、というだけね。まあ、並列計算機とか高速化とかいうと無限に色々なことがあるんだけど、それはちょっとまた別の話。

1980 年代に、よく似た考え方で少なくとも 2 つ論文がでてるんだけど、8 分木を使う、というのは Barnes と Hut のアイディアで、こちらがプログラム書くのが割合楽でツリー作るのも高速ということがあって圧倒的に一番使われてるわね。

ツリー法のいいところは、高速フーリエ変換使う方法みたいに格子を切るわけではなくて、粒子分布に合わせて木構造作るので、密度が高いところと低いところがあっても大丈夫で、計算時間もあんまりのびないということね。

というわけで、この、ツリー法と、 $K(x)$ みたいな関数使って重力相互作用を分けて、短い時間刻み必要な粒子というか粒子間相互作用を別途計算する、というのが、まあ一応現状で最先端の計算法ということになるわね。これは作者のところでドクターとった人の博士論文。

学生 C : えー、それ、本当に大丈夫な方法なんですか?

11.5 ハミルトニアン分割の条件

赤木 : まあ、どうなんだろう?ということで、色々研究されてるわけ。天文での応用としては、2 つの粒子が近づくというのはそこそこ稀だから、そこは短い刻みとか高精度の公式とかでちゃんとやって、特に 2 つの粒子がすごく近づくとかあるから可変時間刻みの高精度公式を使いたいのね。一方、それ以外の遠くの粒子からの重力は、一定刻みで、まあ簡単な話としてはリープフロッグですませたいわけ。そうすると、ひとつ問題なのは、近接遭遇に対して使う公式と $K(x)$ の関係なのね。

学生 C : えーと、どういうことでしょうか?

赤木 : 高精度の公式って、軌道や加速度が少なくともその精度の次数くらいの回数微分可能じゃないと上手くいかない「かもしれない」のね。

学生 C : 何回も微分可能ってどういうことでしたっけ?

赤木 : ちょっと確認だけど、微分可能ってどういうこと?

学生 C : 1 変数でいいですよね?各点で微分可能であればいいんじゃないですか?

赤木 : それだと上手くいかない変な関数沢山つくれるから、普通微分可能っていう時には、微分した関数が連続関数になることをさすのね。連続的微分可能とも。

連続関数の定義はいいわよね?

学生 C : こっちは各点でいいですよね?その点での値に、その点でに近傍の値が収束すること、つまり、任意の x に対して、 $h \to 0$ で $f(x+h) \to f(x)$ になることです。

赤木 : はい、よくできました。なので、n 回連続的微分可能っていうのは、1 回から n 回導関数までが全て連続関数ということね。 C^n 級って書くことが数学では多いかも。

なので、数値積分法が n 次精度だったとしても、積分する関数のほうが C^n じゃなくてもっと微分できる回数少ないと上手くいかないかもしれないじゃない?

学生 C : それなら、無限回微分可能ならいいんじゃないですか?なんか C^∞ とかいうのがなかったでしたっけ?

赤木 : あ、それは、「良い質問」というやつね。どういう関数が $K(x)$ として欲しいかというと $r_0, r_1 \quad (0 < r_0 < r_1)$ に対して

- $r < r_0$ で 0
- $r > r_1$ で 1

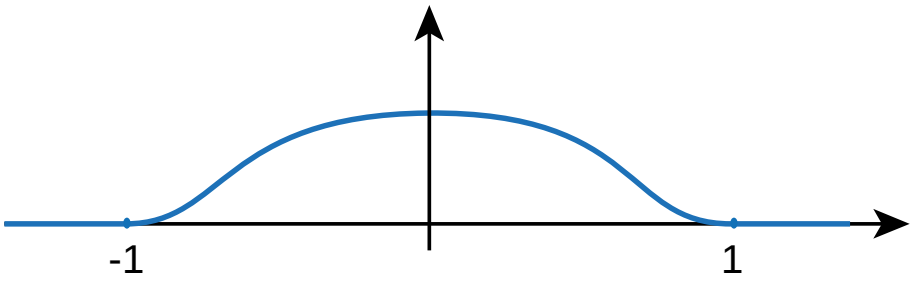

Figure 11.4: 隆起関数の例 (ウィキペディアの隆起関数の項から)

- その間を連続かつ微分可能につなぐ

わけ。で、多項式でなんかできそうだけど、多項式だとその次数まで微分したら必ず境界のところで不連続になるでしょ？例えば、 $x < 0$ で 0、$x \geq 0$ で x^n としたら、 n-1 次導関数までは $x = 0$ で 0 だけど、n 次導関数は 0 じゃないわけ。

逆に、誤差関数、って知ってるわね？正規分布を積分して累積分布関数にしたやつ、これは無限回微分可能で左側で 0 で右側で 1 になるからいい感じなんだけど、数学的に厳密には無限遠まで 0 じゃないから、

学生 C：それ、多項式にするしかないっていってます？

赤木 ：概ねそうなんだけど、一応この辺数学的に厳密には、そうじゃなくって上の性質をもって無限回連続的微分可能な関数って存在するのね。数学の話としては、

- $-1 < x < 1$ で正、その区間での積分が有限値
- その両側で 0
- 実数全体で無限回連続的微分可能

な関数というのがあって、 bump function (隆起関数) っていうの、って私も昨日まで知らなかったんだけど。だから、これの不定積分をつくると、上の欲しい性質満たして無限回微分可能にはなるのね。

学生 C：じゃあそれ使えるんじゃないですか？

赤木 ：と一瞬私も思ったんだけど、よく解説読むと隆起関数は必ず解析的じゃないって書いてあってね、、、

学生 C：解析的ってなんでしたっけ？

赤木 ：テイラー展開が、少なくとも近傍で、収束することが、ある関数がある点で解析的であることの定義。隆起関数は 1 と -1 以外のどこでも解析的なんだけど、この 2 点では解析的じゃないことが数学的に証明できるんだって。

学生 C：はあ。

赤木 ：例えば、

$$\psi(x) = \begin{cases} \exp[-1/(1-x^2)] & \text{for } |x| < 1 \\ 0 & \text{for } |x| \geq 1 \end{cases} \tag{11.8}$$

はそういうものなんだって。図 11.4 みたいな感じ。

学生 C：よくこんな関数考えつくものですね、、、

赤木 ：そうねえ。まあ、世界には色々な人がいるのよ。なんか話がそれたけど、なんだっけ？えーと、だから、無限回は無理として有限回ならどうかということね。隆起関数と同じように、(r_0, r_1) で正で微分可能で両端で n 回導関数まで 0、みたいなのを考えればいいから、多分一番簡単な例は

$$K(r) = \int_{r_0}^{r} (x - r_0)^n (r_1 - x)^n dx \tag{11.9}$$

とかね。これを右端で 1 になるように規格化すればいいわけ。

学生 C : えーと、これでいいってのはわかる気がしますが、他にもっといいのとかあったりしないんですか？

赤木 ：それはまあ、良いとは何かということね。まず、「条件を満たす、最低次の多項式」が、まあ、計算が楽という観点では一番良いから、そういうものを、としてみると、両端で n 回導関数まで 0、関数の値は 0, 1 ということから条件は 2n+2 個あるから、一般には係数が 2n+1 個ある、2n+1 次多項式がいるでしょ？で、上のは 2n 次多項式を 1 回積分して 2n+1 次多項式になってるから、最低次の条件を満たしてて、次数の意味ではよい、ということにはなるわね。

学生 C : 次数がもっと高いののほうがなんかいいことがあるとかそういうのはどうなんでしょう？

赤木 ：その辺あんまりわかってないから、調べてみましょうみたいなね。というわけで、今日の話としては、

- ケプラー問題を、関数 11.9 で分けて、外側は時間刻み一定のリープフロッグ、内側は 4 次ルンゲクッタで積分する
- 離心率は色々かえてみて欲しいけど、とりあえず 0.9, 0.99, 0.999 くらい
- リープフロッグのほうがの時間刻みは適当にに色々
- r_0, r_1 は、軌道長半径は 1 として、仮に $2r_0 = r_1 = 0.5$ とか、¡r_1¿ は色々かえてみて欲しいわ。
- あと、ルンゲクッタの時間刻みは、例えば $|r|/|v|$ になんか係数かけて

で、色々やってみる、ということでどうかしら？

学生 C : はあ、、、なんか $x^2(1-x)^n$ の積分とかでてきてますけどこれどうするんですか？

赤木 ：まあその辺も考えてみて。もちろん、がんばって紙と鉛筆で計算してくれてもいいわよ。

学生 C : えー、ちょっと考えます。

11.6 多項式クラス、多項式の乗算と積分

以下学生 C 独白。

そんなこといっても、今なら Mathematica とか、Web の Wolfram Alpha でちょいちょいと。Alpha にいれてテキストでだすと、

```
\int x^4 (1 - x)^4 dx
```

お、でますね

```
integral x^4 (1 - x)^4 dx = x^9/9 - x^8/2 + (6 x^7)/7 - (2 x^6)/3 + x^5/5 + 定数
```

えーと、でも、これプログラムにするのか。べき乗定義して、あと数値を全部浮動小数点に書き直せば、、、うーん、多項式くらいなんかもうちょっとちょいちょいと計算してくれるライブラリとかないのかな？

(ちょっと捜すが Ruby のは一杯見つかったが Crystal のは発見できなかったっぽく)

多項式って、要するに x のべき乗の係数の配列だから、MathVector みたいに Array から適当につくればいいか、必要なのは、、、掛け算と積分だけかな。一応加減算と微分もいれておくと、、、

```
 1:#
 2:# polynomial handling library
 3:# Copyright 2020- Jun Makino
 4:#
 5:class Polynomial(T) < Array(T)
 6:  def Polynomial.zero
 7:    [T.zero].to_poly
 8:  end
 9:  def extended(i)
10:    (i < self.size) ? self[i] : T.zero
11:  end
12:  def +(a)
13:    newsize = {self.size, a.size}.max
14:    newp=(0..(newsize-1)).map{|k|  self.extended(k)+ a.extended(k)}
15:    while newp[(newp.size) -1] == T.zero
16:      newp.pop
17:    end
18:    newp.to_poly
19:  end
20:  def -() self.map{|x| -x}.to_poly  end
21:  def -(a) self + (-a) end
22:  def +() self end
23:  def *(a)
24:    newp = Array.new(self.size+a.size-1){T.zero}
25:    self.size.times{|i|
26:      a.size.times{|j|
27:        newp[i+j] += self[i]* a[j]
28:      }
29:    }
30:    newp.to_poly
31:  end
32:  def ^(n : Int)
33:    newp=self
34:    (n-1).times{newp *= self}
35:    newp
36:  end
37:  def differentiate
38:    newp= self.map_with_index{|x,k| x*k}
39:    newp.shift
40:    newp.to_poly
41:  end
42:  def integrate
43:    ([T.zero] +  self.map_with_index{|x,k| x/(k+1)}).to_poly
44:  end
```

```
 45:  def evaluate(x)
 46:    self.reverse.reduce{|p,a| p = p*x+a}
 47:  end
 48:end
 49:class Array(T)
 50:  def to_poly
 51:    x=Polynomial(T).new
 52:    self.each{|v| x.push v}
 53:    x
 54:  end
 55:end
```

こんなのかな？加算とか使わないけど。乗算は普通の多項式の筆算と同じで、全部の項同士の積で、同じべきのものを足すから、

```
def *(a)
  newp = Array.new(self.size+a.size-1){T.zero}
  self.size.times{|i|
    a.size.times{|j|
      newp[i+j] += self[i]* a[j]
    }
  }
  newp.to_poly
end
```

と。 T.zero は、この多項式クラスの要素が何か分からないけど、zero をクラスメソッドとしてもってるとしてそれはゼロにあたるものであるという約束で使いましょうということ。整数とか浮動小数点数以外に、有理数でもこれで。あと、理屈としてはこれで多項式の多項式もできるから、多変数にも使えるかな？例えば 2 次と 3 次の多項式の積は 5 次だけど、係数の数としては 3、4 で 6 になるから、まずゼロで埋まったサイズ self.size+a.size-1 の配列を作って、あとはそこに加算していくだけ。to_poly は Array にメソッド追加して多項式に変換するやつ。これは MathVector を真似して作ってみたけどなんか一応これで動くんだけどもうちょっといい方法がありそう。

積分は多項式作るから不定積分で、定数は 0 にするとして、今の多項式の各係数は添字+1 次の項になるからそれで割ると。

```
def integrate
  ([T.zero] +  self.map_with_index{|x,k| x/(k+1)}).to_poly
end
```

なので、これだけ。あとべき乗も作ると、単に n 個の間の掛け算するだけで

```
def ^(n : Int32)
  newp=self
  (n-1).times{newp *= self}
  newp
end
```

かな。これももうちょっと賢く書ける気が。あと、多項式に実際に x の値いれて評価する関数がいるから、これを evaluate で、これホーナーの公式だったかな、 ax^2+bx+c = (ax+b)*x+c みたいな次数が高いほうから順番にするのを使うと、、、配列の順番ひっくり返す必要があるのか。 reverse って、、、あ、あるから、これと reduce で

```
  def evaluate(x)
    self.reverse.reduce{|p,a| p = p*x+a}
  end
```

あれ、これだけか。ちょっと色々テストを、、、

```
  1:#
  2:# test_polynomial.cr
  3:#
  4:require "./polynomial.cr"
  5:require "grlib"
  6:include GR
  7:
  8:poly=[1,1].to_poly
  9:pp! poly
 10:pp! poly+poly
 11:pp! poly - [1,1,1].to_poly
 12:pp! poly*poly*poly
 13:pp! ([1,1].to_poly)^10
 14:pp! ([0.1,0.1].to_poly)^10
 15:pp! ([1,1].to_poly)^5
 16:pp! (([1,1].to_poly)^5).differentiate
 17:pp! (([1.0,1.0].to_poly)^5).integrate
 18:base= [1.0,-1.0].to_poly*[0.0,1.0].to_poly
 19:pp! base
 20:pp! (base^3).integrate
 21:pp! [1.0,2.0,1.0].to_poly.evaluate(1.0)
 22:pp! [1.0,2.0,1.0].to_poly.evaluate(2.0)
 23:pp! [1.0,2.0,1.0].to_poly.evaluate(0.5)
 24:a = [1.0,2.0,1.0].to_poly
 25:b = [a,a].to_poly
 26:pp! a
 27:pp! b
 28:
 29:def create_switch_function(order, rin : Float64, rout : Float64)
 30:  poly= (([1.0,-1.0].to_poly*[0.0,1.0].to_poly)^order).integrate
 31:  normalization = 1.0/poly.evaluate(1.0)
 32:  scale = 1.0/(rout-rin)
 33:  -> (x : Float64){
 34:    if x > rout
 35:      1.0
 36:    elsif x < rin
 37:      0.0
 38:    else
 39:      poly.evaluate((x-rin)*scale)*normalization
 40:    end
 41:  }
 42:end
 43:
 44:f5 = create_switch_function(5,0.5,1)
 45:11.times{|i| pp! f5.call(i*0.1)}
 46:
```

```
  47:setwindow(-0.2,1.2,-0.2,1.2)
  48:box
  49:setcharheight(0.05)
  50:mathtex(0.5, 0.06, "x")
  51:mathtex(0.03, 0.5, "K(x)")
  52:x = Array.new(100){|i| 1.4*i/100 -0.2}
  53:polyline(x, x.map{|v| f5.call(v)})
```

必要なのは、$[x(1-x)]^n$ の積分で、あとこれを $(r_{\rm in}, r_{\rm out})$ で 0 から 1 に変わるように入力の値を細工してから計算するのを前にやった $->$ を使って作るのかな。

```
def create_switch_function(order, rin : Float64, rout : Float64)
  poly= (([1.0,-1.0].to_poly*[0.0,1.0].to_poly)^order).integrate
  normalization = 1.0/poly.evaluate(1.0)
  scale = 1.0/(rout-rin)
  -> (x : Float64){
    if x > rout
      1.0
    elsif x < rin
      0.0
    else
      poly.evaluate((x-rin)*scale)*normalization
    end
  }
end
```

で、最初の行で多項式作ってそれ積分して、次ではそれの x=1 での値をだしてあとでスケールするのにつかう、3 行目は x のほうのスケール用の係数を計算しておく。で、あとは定義通りに範囲内なら poly.evaluate で、左側は 0、右側は 1 と。

実行してみると、

```
gravity> crystal test_polynomial.cr
poly # => [1, 1]
poly + poly # => [2, 2]
poly - [1, 1, 1].to_poly # => [0, 0, -1]
(poly * poly) * poly # => [1, 3, 3, 1]
([1, 1].to_poly) ^ 10 # => [1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]
([0.1, 0.1].to_poly) ^ 10 # => [1.0000000000000006e-10,
 1.000000000000009e-9,
 4.500000000000003e-9,
 1.200000000000001e-8,
 2.100000000000016e-8,
 2.520000000000002e-8,
 2.100000000000016e-8,
 1.200000000000001e-8,
 4.500000000000003e-9,
 1.000000000000009e-9,
 1.0000000000000006e-10]
([1, 1].to_poly) ^ 5 # => [1, 5, 10, 10, 5, 1]
(([1, 1].to_poly) ^ 5).differentiate # => [5, 20, 30, 20, 5]
```

```
(([1.0, 1.0].to_poly) ^ 5).integrate # => [0.0, 1.0, 2.5, 3.3333333333333335, 2.5, 1.0, 0.16666666666660
base # => [0.0, 1.0, -1.0]
(base ^ 3).integrate # => [0.0, 0.0, 0.0, 0.0, 0.25, -0.6, 0.5, -0.14285714285714285]
[1.0, 2.0, 1.0].to_poly.evaluate(1.0) # => 4.0
[1.0, 2.0, 1.0].to_poly.evaluate(2.0) # => 9.0
[1.0, 2.0, 1.0].to_poly.evaluate(0.5) # => 2.25
a # => [1.0, 2.0, 1.0]
b # => [[1.0, 2.0, 1.0], [1.0, 2.0, 1.0]]
f5.call(i * 0.1) # => 0.0
f5.call(i * 0.1) # => 0.0
f5.call(i * 0.1) # => 0.0
f5.call(i * 0.1) # => 0.0
f5.call(i * 0.1) # => 0.0
f5.call(i * 0.1) # => 0.0
f5.call(i * 0.1) # => 0.011654205440003432
f5.call(i * 0.1) # => 0.24650186752007183
f5.call(i * 0.1) # => 0.7534981324802157
f5.call(i * 0.1) # => 0.9883457945601857
f5.call(i * 0.1) # => 1.0
```

グラフは 11.5 と。これで大丈夫かな？

11.7 ハミルトニアン分割によるケプラー問題数値解

じゃあこれいれてケプラー問題か。どうするんだこれ？単純に考えると、リープフロッグは

1. 加速度計算する
2. 速度を半分更新する
3. 位置を更新する
4. 加速度計算する
5. 速度を半分更新する

でよかったはずで、さっきの話だと「位置を更新する」のところがルンゲクッタで軌道積分するになればいいのか。だから、

1. 加速度計算する
2. 速度を半分更新する
3. ルンゲクッタで軌道積分
4. 加速度計算する
5. 速度を半分更新する

ライブラリで作ったリープフロッグが 2 つあるけど、よくわかんないからまず x, v があるほうでやるか。元々が

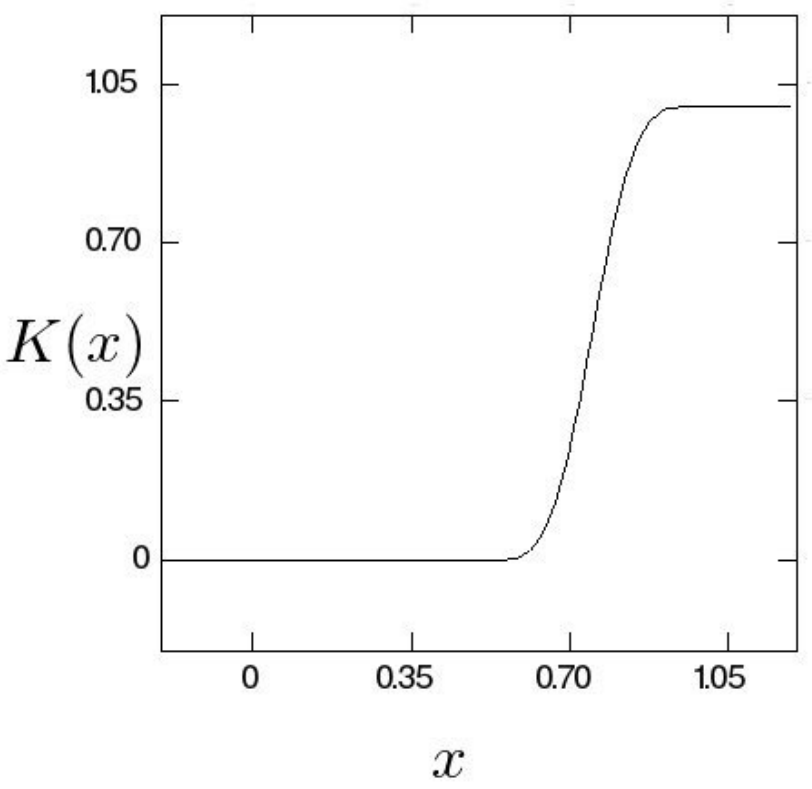

Figure 11.5: 5 次で作ってみたスイッチ関数のグラフ

```
  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
```

だから、これを

```
  def hybrid(x,v,h,f,g,q)
    f0 = f.call(x)
    variable_step_rk4(x,v,h,g,q)
    f1 = f.call(x)
    v+= (f0+f1)*(h/2)
    {x,v}
  end
```

みたいにすればいいのかな。 q はルンゲクッタのほうの精度パラメータとして。f, g がそれぞれリープフロッグ部分とルンゲクッタ部分の加速度。これあと、g が 0 の時に無駄な計算しないで、というのをいれないともったいないけど、それは後回しにすることで。あれ、これだとルンゲクッタに入る前の速度の半分更新がないな。えーと。

```
  def hybrid(x,v,h,f,g,q)
    f0 = f.call(x)
    v+= f0*(h/2)
    variable_step_rk4(x,v,h,g,q)
    f1 = f.call(x)
    v+= f1*(h/2)
    {x,v}
  end
```

ならいいのかな。じゃあ、あとは、、、

(数日後)

学生 C : なんかできたみたいです。

赤木 : え、本当？

学生 C : 本当って、、とりあえずエネルギーのグラフだけ。

```
  splitintegrator -g -w 0.0001 -e 0.99999 -q 0.005 -s 0.01 -E -o 10 -p 4
```

で、エネルギー誤差が図 11.6 です。

赤木 : 離心率は、、、 0.99999 ってこと？それでちゃんと積分できたなら割とあってるかもね。プログラムはどんなの？

学生 C :

```
    1:#
    2:# splitintegrator.cr
    3:# Copyright 2020 J. Makino
```

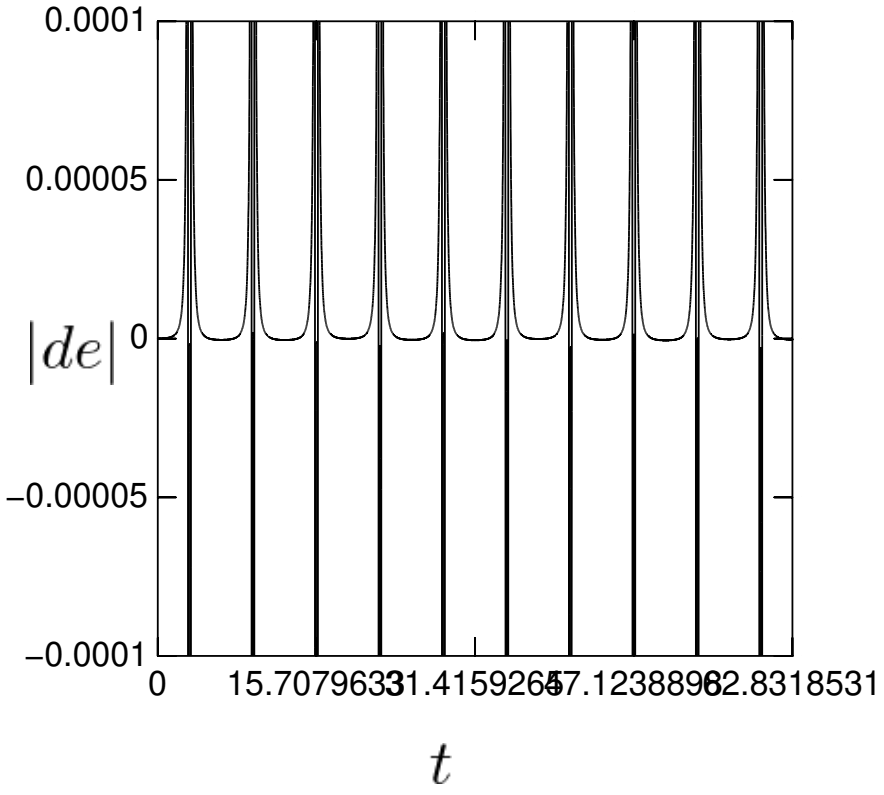

Figure 11.6: ハミルトニアン分割での積分結果

```
 4:#
 5:
 6:require "grlib"
 7:require "./integratorlib.cr"
 8:require "./vector3.cr"
 9:require "./mathvector.cr"
10:require "./polynomial.cr"
11:require "clop"
12:include Math
13:include GR
14:
15:optionstr= <<-END
16:  Description: Test integrator for Kepker problem with split integrator
17:  Long description:
18:    Test integrator for Kepker problem with split integrator
19:    (c) 2020, Jun Makino
20:
21:  Short name:           -s
22:  Long name:  --soft-step
23:  Value type:float
24:  Default value: 0.1
25:  Variable name: h
26:  Description:timestep for leapfrog part
27:  Long description:     timestep for leapfrog part
28:
29:  Short name: -o
30:  Long name:  --norbits
31:  Value type:int
32:  Default value:1
33:  Variable name:norb
34:  Description:Number of orbits
35:  Long description:     Number of orbits
36:
37:  Short name:-w
38:  Long name:  --window-size
39:  Value type:  float
40:  Variable name:wsize
41:  Default value:1
42:  Description:Window size for plotting
43:  Long description:
44:    Window size for plotting orbit. Window is [-wsize, wsize] for both of
45:    x and y coordinates
46:
47:  Short name:-e
48:  Long name:--ecc
49:  Value type:float
50:  Default value:0.0
51:  Variable name:ecc
52:  Description:Initial eccentricity of the orbit
53:  Long description:     Initial eccentricity of the orbit
54:
55:  Short name:-g
```

```
 56:  Long name:--graphic-output
 57:  Value type:        bool
 58:  Variable name:gout
 59:  Description:
 60:    whether or not create graphic output (default:no)
 61:  Long description:
 62:    whether or not create graphic output (default:no)
 63:
 64:
 65:  Short name:-E
 66:  Long name:--plot-energy-error
 67:  Value type:        bool
 68:  Variable name:eplot
 69:  Description:
 70:    plot de instead of orbit
 71:  Long description:
 72:    plot de instead of orbit
 73:
 74:  Short name:-r
 75:  Long name:--outer-cutoff-radius
 76:  Value type:        float
 77:  Variable name:rout
 78:  Default value:0.5
 79:  Description:          outer cutoff radius for splitting
 80:  Long description:     outer cutoff radius for splitting
 81:
 82:
 83:  Short name:-p
 84:  Long name:--cutoff-order
 85:  Value type:        int
 86:  Variable name:cutoff_order
 87:  Default value:4
 88:  Description:          Smoothness order for the cutoff function
 89:  Long description:     Smoothness order for the cutoff function
 90:
 91:  Short name:-q
 92:  Long name:--accuracy-parameter
 93:  Value type:        float
 94:  Variable name:eta
 95:  Default value:0.1
 96:  Description:          Parameter for timestep criterion for hard part
 97:  Long description:     Parameter for timestep criterion for hard part
 98:END
 99:
100:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
101:options=CLOP.new(optionstr,ARGV)
102:
103:def variable_step_rk4(x,v,h,g,q)
104:  t=0.0
105:  ff =  ->(x : MathVector(Vector3), t : Float64){ [x[1], g.call(x[0])].to_mathv}
106:  while t < h
107:    dt = sqrt((x*x)/(v*v))*q
```

```
108:    dt = h-t if t+dt > h
109:    xv = [x,v].to_mathv
110:    xv,t = Integrators.rk4(xv,t,dt,ff)
111:    x,v= xv
112:  end
113:  {x,v}
114:end
115:
116:def hybrid(x,v,h,f,g,q)
117:  f0 = f.call(x)
118:  v+= f0*(h/2)
119:  x,v = variable_step_rk4(x,v,h,g,q)
120:  f1 = f.call(x)
121:  v+= f1*(h/2)
122:  {x,v}
123:end
124:
125:def kepler_acceleration(x,m)
126:  r2 = x*x
127:  r=sqrt(r2)
128:  mr3inv = m/(r*r2)
129:  -x*mr3inv
130:end
131:def energy(x,v,m)
132:  m*(-1.0/sqrt(x*x)+v*v/2)
133:end
134:
135:def create_switch_function(order, rin : Float64, rout : Float64)
136:  poly= (([1.0,-1.0].to_poly*[0.0,1.0].to_poly)^order).integrate
137:  normalization = 1.0/poly.evaluate(1.0)
138:  scale = 1.0/(rout-rin)
139:  -> (x : Float64){
140:    if x > rout
141:      1.0
142:    elsif x < rin
143:      0.0
144:    else
145:      poly.evaluate((x-rin)*scale)*normalization
146:    end
147:  }
148:end
149:
150:m=1.0
151:switch= create_switch_function(options.cutoff_order,
152:                                options.rout*0.5,
153:                                options.rout)
154:fout =  -> (xx : Vector3){kepler_acceleration(xx,m)*switch.call(sqrt(xx*xx))}
155:fin=  -> (xx : Vector3){kepler_acceleration(xx,m)*(1-switch.call(sqrt(xx*xx)))}
156:t=0.0
157:x= Vector3.new(1.0+options.ecc,0.0,0.0)
158:v= Vector3.new(0.0,sqrt((1-options.ecc)/(1+options.ecc)),0.0)
159:if options.gout
```

```
160:  wsize=options.wsize
161:  if options.eplot
162:    setwindow(0.0, options.norb*PI*2,-wsize, wsize)
163:  else
164:    setwindow(-wsize, wsize,-wsize, wsize)
165:  end
166:  box
167:  setcharheight(0.05)
168:  if options.eplot
169:    mathtex(0.5, 0.06, "t")
170:  mathtex(0.06, 0.5, "|de|")
171:  else
172:    mathtex(0.5, 0.06, "x")
173:    mathtex(0.06, 0.5, "y")
174:  end
175:end
176:e0 = energy(x,v,m)
177:emax = 0.0
178:de=0.0
179:while t < options.norb*PI*2 - options.h
180:  dep=de
181:  xp=x
182:  tp=t
183:  x, v = hybrid(x, v, options.h, fout, fin, options.eta)
184:  t+=options.h
185:  de =energy(x,v,m)-e0
186:  emax = {de.abs, emax}.max
187:  if options.gout
188:    if options.eplot
189:      polyline([tp, t], [dep, de])
190:    else
191:      polyline([xp[0], x[0]], [xp[1], x[1]])
192:    end
193:  end
194:  pp! x, v, energy(x,v,m)
195:end
196:p! -emax/e0
197:c=gets if options.gout
198:
```

です。元々のケプラー軌道の積分のプログラムから結構あんまり変わってないので、簡単に説明します。プログラム本体は 154 行目からで、 158 行目からの

```
switch= create_switch_function(options.cutoff_order,
                               options.rout*0.5,
                               options.rout)
fout =  -> (xx : Vector3){kepler_acceleration(xx,m)*switch.call(sqrt(xx*xx))}
fin=  -> (xx : Vector3){kepler_acceleration(xx,m)*(1-switch.call(sqrt(xx*xx)))}
```

は、スイッチ関数と、それ使った、ハミルトニアン分割の f と g を定義してます。create_switch_function は、多項式扱うクラス自分で作って、関数定義しました。次数と、内側、外側の境界を与えると、それぞれで 0, 1 に、与えられた次数まで微分可能につなげる関数を生成します。

fout, fin は外側、内側の関数で、もともとの相互作用がケプラーなので距離の 2 乗に反比例の力、これは前に使ったのと同じですね。それにスイッチ関数と、1 からスイッチ関数をひいたものをそれぞれ掛けてます。

そのあとは、エネルギー誤差を書けるようにしたのの他は前のケプラー問題の数値積分とほとんど同じなのですが、数値積分するとこの本体が、前のは

```
x, v = integrator.call(x,v,h)
```

だったのが

```
x, v = hybrid(x, v, options.h, fout, fin, options.eta)
```

になってます。今のところ、積分公式はリープフロッグ+4 時間ルンゲクッタだけなので、そのへんはオプションないですが、リープフロッグの時間刻み、外側の関数、内側の関数、内側の積分の時間刻みパラメータの順番です。この hybrid の本体は

```
def hybrid(x,v,h,f,g,q)
  f0 = f.call(x)
  v+= f0*(h/2)
  x,v = variable_step_rk4(x,v,h,g,q)
  f1 = f.call(x)
  v+= f1*(h/2)
  {x,v}
end
```

です。システムじゃなくて x, v, f をもらうほうのリープフロッグが

```
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
```

だったんですが、これをちょっと書換えて

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

として、

```
  x+= v*h
```

のところを

```
x,v = variable_step_rk4(x,v,h,g,q)
```

に置き換えてます。これは、そうして、といわれた通りです。

赤木 ：なるほど。もっともらしいわね。

学生 C：variable_step_rk4 は g のほうを時間刻み可変のルンゲクッタで時間 h まで積分するものです。これはこんな感じです。

```
def variable_step_rk4(x,v,h,g,q)
  t=0.0
  ff =  ->(x : MathVector(Vector3), t : Float64){ [x[1], g.call(x[0])].to_mathv}
  while t < h
    dt = sqrt((x*x)/(v*v))*q
    dt = h-t if t+dt > h
    xv = [x,v].to_mathv
    xv,t = Integrators.rk4(xv,t,dt,ff)
    x,v= xv
  end
  {x,v}
end
```

これあんまり美しくないんですが、ライブラリで定義したルンゲクッタをそのまま使うためとりあえずこうしてます。ライブラリで定義したルンゲクッタはケプラー問題なら x と v を一緒に 1 つのベクトルと思う関数で、これには MathVector クラスを渡す必要がありますが、ここまではリープフロッグで x, v を別の Vector3 クラスでもってます。なので、加速度を計算する関数とかもその辺変える必要がありました。それが

```
ff =  ->(x : MathVector(Vector3), t : Float64){ [x[1], g.call(x[0])].to_mathv}
```

で、ここでは、 MathVector として、 Vector3 を 2 つ要素に持つようにしてます。最初は 6 要素のベクトルを作って書いてたんですが、こっちでいけるかなと思ってやったらコンパイルできたので。多分大丈夫かなあと。

赤木 ：まあいいんじゃないかしら？速度気にするとまた違う書き方がいるかもだけど。

学生 C： はい。で、 while の中で h だけ積分します。時間刻みを

```
dt = sqrt((x*x)/(v*v))*q
```

でだして、 t+dt が h を超えないように

```
dt = h-t if t+dt > h
```

で小さくします。

赤木 ：これ、このあとで t+dt が永久に h にならないとかは起こらない？

学生 C： 引き算で、 h のほうが大きいと大丈夫じゃないですか？

赤木 ：そうね、IEEE-754 の規定からは大丈夫なはずなんだけど、何故そういえるかは調べといてね。

学生 C : 読者がですね？で、あとは、 x,v を MathVector にして rk4 に渡してまた Vector3 2 つに戻すだけです。

11.8 課題

1. 関数 11.9 が両端で n 回導関数まで 0 であることを証明せよ。
2. 4 次ルンゲクッタの場合に、スイッチ関数の次数を変えるとエネルギー誤差の振舞いがどう変わるか調べよ
3. 8 次程度のルンゲクッタを実装して、4 次の場合と同様にエネルギー誤差とスイッチ関数の次数の関係を調べよ。

11.9 まとめ

1. ハミルトニアン分割によって、重力多体問題を高速かつ高精度に計算する方法を、まずケプラー問題で振舞いを調べた。
2. この目的のため、多項式の乗算・不定積分を行なうライブラリを作成した。

11.10 参考資料

Skeel, R. D. and Biesiadecki, J. J., "Symplectic Integration with Variable Stepsize", Annals of Numer. Math. 1994, 1, 191-198. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.9532&rep=rep1&type=pdf

Hernandez, David M. Improving the accuracy of simulated chaotic N-body orbits using smoothness, Monthly Notices of the Royal Astronomical Society, Volume 490, Issue 3, p.4175-4182 https://academic.oup.com/mnras/abstract/490/3/4175/5575201?redirectedFrom=fulltext

Chapter 12

粒子データ入出力

赤木 ：ここからは、割合本格的に多体計算しようみたいな話にはいろうかと思って。

学生 C : 今までとどう違うんですか？

赤木 ：今までのプログラムは、N 体とかでもあんまり粒子数大きいのじゃなくて、論文というか研究に使える感じじゃないんだけど、それを、まあ、研究レベルにね。

学生 C : なんか大変そうですが、、、

赤木 ：まあ、その辺を、 Crystal で書くことでどれくらい楽にできるかをみていこうみたいなね。

学生 C : はあ。

赤木 ：まず、ある程度の粒子数を扱う、とすると、結果をファイルに書いたり、入力をファイルから読んだり、としたくなるのね。今回その話。

学生 C : ずーっと前の章の練習問題になんかなかったでしたっけ？FDPS のサンプルプログラムのフォーマットとか。

赤木 ：粒子数とか時刻を最初に書いて、あと 1 行 1 粒子みたいなのね？

学生 C : それです。

赤木 ：あれはあれでいけないわけじゃないんだけど、このデータは何か、どうやって作ったものか、というのが記録されてないじゃない？例えば、どういうコマンドで作ったか、とか、そもそもこの数字は何か、とか。

学生 C : えー、データファイルってそういうものじゃなんですか？数字が並ぶわけですよね？

赤木 ：それだとなんだかわからなくなるから、わかるようにしておこうよ、という話が色々なあるわけ。IT 業界だと XML とか JSON とか YAML とか。天文学だと FITS というのが観測データではよく使われているし、大規模数値計算では HDF かしらね。

学生 C : じゃあそのどれか使えばいいんでは?

赤木 ： それでもいいんだけど、そういうの使うのなんというか結構面倒でプログラムも大変だから、なるべく簡単なのですませたいのね。まあ、他の人が書いたプログラムに、だと変換プログラム書かないといけないけど、それはまあしょうがないとして。

学生 C : そうすると、どうするんですか？

赤木 ：最近だと楽に使えて、あとできたファイルも扱いやすいのは YAML かな。作者氏とあと色々な人で昔論文書いてて、標準形式にしようと提案はしてるの。

学生 C : でもそんなには使われてないと、、、、

赤木 ：まあそうなんだけど、この際だから。少なくとも論文があるから、これに従ってやってね、といえるのはメリットね。YAML っぽいものはすで使ってて、コマンドラインオプションを書く形式がそれなんだけど、あそこでは割合簡単なのしか使ってなかったらから、論文に従って粒子データの例をまず出すわね。例えば。番号 (名前)、位置、速度、質量をもつ 1 つの粒子を以下のように表しましょう、というのが主張で、これに PSDF (Particle Stream Data Format) っていう名前つけてるの。

```
--- !Particle
id: 0
r:
- 0.1
- 0.2
- 0.3
v:
- -1
- -2
- -3
m: 1.0
```

基本的に

```
key: value
```

みたいな、名前+コロン、スペース、値 である名前の変数なり物理量の値を指定、位置とか速度みたいに複数の値の時には、上の例みたいに次の行を ”-” で始めて、それで始まる行が続く限り、配列に値がはいるみたいな。最初の「—] は、新しく何かの始まりで、そのあとの 「!Particle」は「タグ」っていう仕掛けになるのね。この辺、YAML のちゃんとした話は少し複雑なんだけど、とりあえず 1 つの粒子のデータが

```
--- !Particle
```

で始まる、なので、次に 「—」 がでてくるかファイルの終わりがくるかしたらその粒子は終わりで別にの粒子なりなんか違うものになるわけ。

PSDF の論文では、 粒子の名前、物理量として以下を使おう、としてるわね。

```
id     名前 (整数でもテキストでもよい)
m      質量
t      粒子の時刻
t_max  この粒子が有効な最大時刻
r      位置 3 次元ベクトル
v      速度 3 次元ベクトル
pot    重力ポテンシャル
acc    加速度 3 次元ベクトル
jerk   加速度の一階時間導関数 3 次元ベクトル
snap     2 階 3 次元ベクトル
crackle 3 階 3 次元ベクトル
pop      4 階 3 次元ベクトル
```

これに従っておけば、とりあえずファイルの意味は明らか、ということね。本当は、!Particle のところで、このデータ構造が定義されている URI、ウェブ上の場所ね、を書くとかあるんだけど、論文ではそこまでやってないからとりあえずこれで。

学生 C : なんか一杯ありますね。全部ないといけないんですか？

赤木 : 良い質問ね。もちろん、使うものだけあればいい、というのと、プログラムの作り方として、例えばこの粒子データファイル、みんなスナップショットというからここでもそう呼ぶけど、スナップショットをなんか加工するプログラムがあったとして、それは、自分が扱うデータのことはしか知らないとして、他のデータは読んだものをそのまま書き出して欲しいわけ。で、この時に、何なのか知らないデータがあっても、それを消したり変なものにしないような仕掛けがほしいのね。

あと、データファイルに、こういう粒子のデータだけでなくて実行したコマンドとかオプションとかの記録があったら付け加える、というのも。

学生 C : 注文が多い気がしますが、、、なんか複雑なので、少し整理させて下さい。まず、ここで作るのは、あくまでも多体シミュレーションのスナップショットファイルである、ということでいいですね？

赤木 : いいわよ。でも、それに、粒子データじゃない、実行したコマンドの名前とかオプションとか、あとそれ以外の色々なプログラムのログ出力とかも書けるようにしたいわけ。

学生 C : それはだから、 Particle じゃない何かが必要だってことですね。そうしたら、なんかこんなふうなファイルですかね？

```
---- !CommandLog
command:  mkplummer -n 10 ...
log:
  (if any)
---- !Particle
id: 0
....

---- !Particle
id: 1
....

---- !Particle
id: 9
....
```

赤木 : そうね。

学生 C : なんか、クラスから YAML に変換は色々書くとやってくれるみたいなので、とりあえず上の CommandLog と Particle に対応するクラス定義と、それでなんか書くプログラム作ってみました。こっちがクラス定義

```
 1:#
 2:# nacsio0.cr
 3:#
 4:# Basic YAML-based IO library for
 5:# nacs (new art of computational science)
 6:# Copyright 2020- Jun Makino
 7:
 8:require "yaml"
 9:require "./vector3.cr"
10:
11:struct Vector3
```

```
 12:  def initialize(ctx : YAML::ParseContext, node : YAML::Nodes::Node)
 13:    a=Array(Float64).new(ctx, node)
 14:    @x=a[0]; @y=a[1]; @z=a[2]
 15:  end
 16:end
 17:
 18:module Nacsio
 19:  class CommandLog
 20:    YAML.mapping(
 21:      command: {type: String, default: "",},
 22:      log: {type: String, default: "",},
 23:    )
 24:    def self.new
 25:      CommandLog.from_yaml("")
 26:    end
 27:    def self.new(logstring : String)
 28:      c=CommandLog.new
 29:      c.command = ([PROGRAM_NAME]+ ARGV).join(" ")
 30:      c.log = logstring
 31:      c
 32:    end
 33:    def to_nacs
 34:      self.to_yaml.gsub(/---/, "--- !CommandLog")
 35:    end
 36:    def add_command
 37:      @command += "\n"+([PROGRAM_NAME]+ ARGV).join(" ")
 38:      self
 39:    end
 40:
 41:  end
 42:
 43:  class Particle
 44:    def to_nacs
 45:      self.to_yaml.gsub(/---/, "--- !Particle")
 46:    end
 47:    def self.new
 48:      Particle.from_yaml("")
 49:    end
 50:    YAML.mapping(
 51:      id: {type: Int64, default: 0i64,},
 52:      time: {type: Float64, key: "t", default: 0.0,},
 53:      mass: {type: Float64, key: "m",default: 0.0,},
 54:      pos: {type: Vector3, key: "r",default: [0.0,0.0,0.0].to_v,},
 55:      vel: {type: Vector3, key: "v",default: [0.0,0.0,0.0].to_v,},
 56:    )
 57:  end
 58:end
```

こっちがテストプログラム

```
  1:# nacsiotest0.cr
```

```
 2:#
 3:# Test code for Basic YAML-based IO library for
 4:# nacs (new art of computational science)
 5:
 6:require "./nacsio0.cr"
 7:include Nacsio
 8:
 9:print CommandLog.new("Sample log message").to_nacs
10:(0..2).each{|id|
11:  obj=Particle.new
12:  obj.id =id.to_i64
13:  obj.pos[0]=id*0.1
14:  print obj.to_nacs
15:}
```

で、実行結果が

```
gravity> crystal nacsiotest0.cr
[2mShowing last frame. Use --error-trace for full trace.[0m

In [4mnacsio0.cr:20:10[0m

[2m 20 | [0m[1mYAML.mapping([0m
           [32;1m^------[0m
[33;1mError: undefined method 'mapping' for YAML:Module[0m
```

です。 nacs という名前は、作者が元々使っていた acs に new の意味で n つけました。

赤木 ：なるほど。コードの中身説明してみて。

学生 C：まず、CommandLog クラスですね。その前に Vector3 をなんかしてますがこの話はあとで。

```
YAML.mapping(
  command: {type: String, default: "",},
  log: {type: String, default: "",},
)
```

の、YAML.mapping がクラスと YAML 書式の関係つける仕掛けみたいです。最初の command とか log がメンバー変数の名前、{} の中は type が型を、default が (あれば) 指定がなかった時のデフォルト値、 key が (あれば) YAML の中でのキー (なければメンバー変数の名前)、あと nilable (値を nil にできるかどうか) と、もっと色々なことの指定ができるみたいですがまあ普通これくらいで。

ちょっと面白かったのは、これで型とデフォルトの値を書いておくと、initialize を別に定義しなくても from_yaml というクラス関数が new の代わりになることで。でも new はないので、

```
def self.new
  CommandLog.from_yaml("")
end
```

でデータがない文字列を解釈して、というのでデフォルトの値がはいったインスタンスをつくれます。もっとちゃんとしたやり方もありそうですが、動いたからこれでいいかなと。 new とか書換えられるんですね。

あと property も勝手に入るみたいで、

```
property :log
```

とか書かなくてもクラスの外から読み書きできました。あと、 new で文字列渡すと、 command のほうに実行したコマンド、 log のほうにその文字列を入れるようにしました。

赤木 ：実行結果の command のところが謎文字列だけど？

学生 C : これ多分、 crystal foo.cr とすると実際にはコンパイルされるわけで、コンパイルされたものがそれだということなんではないかと、、、

赤木 ：そうね。多分。

学生 C : あと、 to_yaml で YAML の文字列にしてくれるんですが、!CommandLog のとこがつかないので gsub でつける関数が to_nacs です。この辺なんかもっと YAML 的に正しい作法とかあるんじゃないかと思いますが、、、、

赤木 ：まあとりあえずこれでいいわよ。Particle は？

学生 C : YAML.mapping の中身と、あと、つけるタグが CommandLog じゃなくて Particle だというだけですね。あ、ややこしいのが、 pos, vel で、これ Vector3 にしてるじゃないですか？最初は Array(Float64) で書いてたんですが、それだとあとで不便な気がしたのでこっちにしました。で、コンパイルしたら、なんか

```
> 197 |
> 198 |
> 199 |                   Vector3.new(ctx, value_node)
                                ^--
Error: no overload matches 'Vector3.new' with types YAML::ParseContext, YAML::Nodes::Node+

Overloads are:
 - Vector3.new(x : Float64 = 0, y : Float64 = 0, z : Float64 = 0)
```

みたいな、謎なエラーなんですが、要するに Vector3.new(ctx : YAML::ParseContext, node : YAML::Nodes::Node) という関数がないといってるっぽくて、これが Array では通ってたのは Array にはそういうものがあるのかな？とレファレンスみたらあったので、それを Array で実行した結果を Vector3 にする initialize 関数を Vector3 に追加したら上手くいきました。

```
struct Vector3
  def initialize(ctx : YAML::ParseContext, node : YAML::Nodes::Node)
    a=Array(Float64).new(ctx, node)
    @x=a[0]; @y=a[1]; @z=a[2]
  end
end
```

のとこです。

赤木 ： ctx, node って何？

学生 C : そこ調べてないです。動くからいいんじゃないかと。

赤木 ： そう、、、ねえ、、、 *Crystal* 公式の *YAML* ドキュメント[1] も正直なところよくわからないしね、、、

学生 C : ちょっとこの辺まだドキュメント書けてない感じしますね。

[1] https://crystal-lang.org/api/latest/YAML.html

赤木 ：そうね。これ、新しく作って書くのはこうでいいとして、読んでなんかして書く、というのをやってみてくれない？つまり、例えば、このファイル読んで、 id を 1 増やす、みたいな、粒子に変更加えるもの。

この時に、自分が知らない粒子クラスの YAML データでも、 id のデータがあればそれをいじる、それ以外はさわらない、として欲しいの。

学生 C： それは、この YAML 用の Particle クラスは使わない、ということになります？

赤木 ：そうね。まずはそれで。

学生 C : CommandLog はどうしましょうか？

赤木 ：これは、使って、あったコマンドログの最後に自分を付け加えるでどう？

学生 C : 考えてみます。

(数日後)

一応できたことにしたいですが、、、、

赤木 ：あら、早いわね。動かすとどんな感じ？

学生 C :

```
gravity> crystal nacsiotest0.cr > test.yml
[2mShowing last frame. Use --error-trace for full trace.[0m

In [4mnacsio0.cr:20:10[0m

[2m 20 | [0m[1mYAML.mapping([0m
           [32;1m^------[0m
[33;1mError: undefined method 'mapping' for YAML:Module[0m
```

```
gravity> crystal nacsreadwrite3.cr < test.yml
[2mShowing last frame. Use --error-trace for full trace.[0m

In [4mnacsreadwrite3.cr:10:8[0m

[2m 10 | [0m[1mYAML.mapping([0m
           [32;1m^------[0m
[33;1mError: undefined method 'mapping' for YAML:Module[0m
```

と、こんな感じで、 id が 0-2 だったのを 1 増やして 1-3 になってます。

赤木 ：それっぽいわね。プログラム本体は？

学生 C： とりあえずこれです

```
   1:# nacsreadwrite3.cr
   2:#
   3:# Test code for Basic YAML-based read/write for
   4:# nacs (new art of computational science)
   5:
   6:require "./nacsio.cr"
```

```
 7:include Nacsio
 8:
 9:class IDParticle
10:  YAML.mapping(
11:    id: {type: Int64, default: 0i64,},
12:  )
13:end
14:
15:update_commandlog
16:
17:while (sp= CP(IDParticle).read_particle).y != nil
18:  p = sp.p
19:  p.id += 1
20:  sp.p = p
21:  sp.print_particle
22:end
```

ここでは、6 行目で nacsio.cr を読み込んで、9-13 行で IDParticle という id だけもったクラスを定義します。クラスは YAML.mapping だけです。

で、 nacsio.cr のほうでは

```
update_commandlog
```

でログ部分を読んで書く、

```
CP(T).read_particle
```

でクラス T の粒子を読み込む、 print_particle で CP(T) 型を YAML で書く、となるわけです。

赤木 ：CP(T) 型って何？

学生 C： これは、T のとこがクラスで、そのクラスの変数をメンバーとしてもつクラスを作ります。ライブラリ本体は

```
 1:#
 2:# nacsio.cr
 3:#
 4:# Basic YAML-based IO library for
 5:# nacs (new art of computational science)
 6:# Copyright 2020- Jun Makino
 7:
 8:require "yaml"
 9:require "./vector3"
10:
11:# struct Vector3
12:#   def initialize(ctx : YAML::ParseContext, node : YAML::Nodes::Node)
13:#     a=Array(Float64).new(ctx, node)
14:#     @x=a[0]; @y=a[1]; @z=a[2]
15:#   end
16:# end
17:
```

```
18:module Nacsio
19:  extend self
20:  class CommandLog
21:    include YAML::Serializable
22:    @[YAML::Field(key: "command")]
23:    property command : String
24:    @[YAML::Field(key: "log")]
25:    property log : String
26:
27:    # YAML.mapping(
28:    #   command: {type: String, default: "",},
29:    #   log: {type: String, default: "",},
30:    # )
31:    def self.new
32:      CommandLog.from_yaml(%(
33:  command:
34:  log:
35:))
36:    end
37:    def self.new(logstring : String, progname = PROGRAM_NAME, options = ARGV)
38:      c=CommandLog.new
39:      c.command = ([progname]+ options).join(" ")
40:      c.log = logstring
41:      c
42:    end
43:    def to_nacs
44:      self.to_yaml.gsub(/---/, "--- !CommandLog")
45:    end
46:    def add_command(progname = PROGRAM_NAME, options = ARGV)
47:      @command += "\n"+([progname]+ options).join(" ")
48:      self
49:    end
50:
51:  end
52:
53:  class Particle
54:    def to_nacs
55:      self.to_yaml.gsub(/---/, "--- !Particle")
56:    end
57:    def self.new
58:        Particle.from_yaml(%(
59:  id: 0
60:  t:  0.0
61:  m:  0.0
62:  r:
63:    x: 0.0
64:    y: 0.0
65:    z: 0.0
66:  v:
67:    x: 0.0
68:    y: 0.0
69:    z: 0.0
```

```
 70:))
 71:    end
 72:    include YAML::Serializable
 73:    @[YAML::Field(key: "id")]
 74:    property id : Int64
 75:    @[YAML::Field(key: "t")]
 76:    property time : Float64
 77:    @[YAML::Field(key: "m")]
 78:    property mass : Float64
 79:    @[YAML::Field(key: "r")]
 80:    property pos : Vector3
 81:    @[YAML::Field(key: "v")]
 82:    property vel : Vector3
 83:    # YAML.mapping(
 84:    #   id: {type: Int64, default: 0i64,},
 85:    #   time: {type: Float64, key: "t", default: 0.0,},
 86:    #   mass: {type: Float64, key: "m",default: 0.0,},
 87:    #   pos: {type: Vector3, key: "r",default: [0.0,0.0,0.0].to_v,},
 88:    #   vel: {type: Vector3, key: "v",default: [0.0,0.0,0.0].to_v,},
 89:    # )
 90:  end
 91:
 92:  def update_commandlog(progname = PROGRAM_NAME, options = ARGV,
 93:                        of = STDOUT)
 94:    s=gets("---")
 95:    s=gets("---") if s == "---"
 96:    a=s.to_s.split("\n")
 97:    a.pop if a[a.size-1]=="---"
 98:    if a[0] != " !CommandLog"
 99:      raise("Input line #{a[0]} not the start of Commandlog")
100:    else
101:      a.shift
102:      ss = (["---\n"] + a).join("\n")
103:      of.print CommandLog.from_yaml(ss).add_command(progname,options).to_nacs
104:    end
105:  end
106:  def read_commandlog(ifile= STDIN)
107:    c=CommandLog.new
108:    s=ifile.gets("---")
109:    s=ifile.gets("---") if s == "---"
110:    a=s.to_s.split("\n")
111:    a.pop if a[a.size-1]=="---"
112:    if a[0] != " !CommandLog"
113:      raise("Input line #{a[0]} not the start of Commandlog")
114:    else
115:      a.shift
116:      ss = (["---\n"] + a).join("\n")
117:      c= CommandLog.from_yaml(ss)
118:    end
119:    c
120:  end
121:
```

```
122:  class CP(T)
123:    property :p, y
124:    def initialize(p : T, y : YAML::Any)
125:      @p=p
126:      @y=y
127:    end
128:    def self.read_particle(ifile = STDIN)
129:      s=ifile.gets("---")
130:      retval=CP(T).new(T.new,YAML::Any.new(nil))
131:      if  s != nil
132:        a=s.to_s.split("\n")
133:        a.pop if a[a.size-1]=="---"
134:        if a.size > 0
135:          if a[0] != " !Particle"
136:            raise("Input line #{a[0]} not the start of Particle")
137:          else
138:            a.shift
139:            ystr = (["--- \n"] + a).join("\n")
140:            retval=CP(T).new(T.from_yaml(ystr), YAML.parse(ystr))
141:          end
142:        end
143:
144:      end
145:      retval
146:    end
147:
148:    def print_yamlpart(of = STDOUT)
149:      of.print @y.to_yaml.gsub(/---/, "--- !Particle")
150:    end
151:
152:    def print_particle(of = STDOUT)
153:      yy=@y.as_h.to_a
154:      ycore = YAML.parse(@p.to_yaml).as_h.to_a
155:      ycore.each{|core|
156:        yy.reject!{|x| x[0]== core[0]}
157:      }
158:      yy = ycore + yy
159:      newstring = YAML.build{|yaml|
160:        yaml.mapping{
161:          yy.each{|x|
162:            yaml.scalar x[0].as_s
163:            if x[1].raw.class == Int64
164:              yaml.scalar x[1].as_i64
165:            elsif x[1].raw.class == Float64
166:              yaml.scalar x[1].as_f
167:            elsif x[1].raw.class == String
168:              yaml.scalar x[1].as_s
169:            elsif x[1].raw.class == Array(YAML::Any)
170:              xx = x[1].as_a
171:              yaml.sequence { xx.each{|v| yaml.scalar v.as_f}}
172:            end
173:          }
```

```
174:        }
175:      }
176:      of.print newstring.gsub(/---/, "--- !Particle")
177:    end
178:  end
179:
180:  def repeat_on_snapshots(p = Particle.new, read_all = false)
181:    sp= CP(typeof(p)).read_particle
182:    no_time = (sp.y["t"] == nil )
183:    while sp.y != nil
184:      pp=[sp]
185:      time = sp.y["t"].as_f  unless read_all
186:      while (sp= CP(typeof(p)).read_particle).y != nil &&
187:            (read_all || no_time || sp.y["t"].as_f == time)
188:        pp.push sp
189:      end
190:      yield pp
191:    end
192:  end
193:end
194:
```

で、CP(T) は

```
def initialize(p : T, y : YAML::Any)
  @p=p
  @y=y
end
```

が new で呼ぶ関数なので、 T 型の p と、YAML::Any という謎型の y をメンバ変数に持つわけです。で、 read_particle はこの型のクラスメソッドで

```
sp= CP(IDParticle).read_particle
```

で標準入力、これ変更できたほうがいい気もしますが、から 1 粒子読んで、引数で与えた粒子クラスで解釈できるものを sp.p に、それを含めた全部を Crystal の YAML ライブラリ のデフォルトの型である YAML::Any にいれる、ということをします。あと、ファイルが終わってるとかの時には sp.y のほうを nil にします。こっちは Any なので nil にしてもコンパイラが文句をいかなかったので。

プログラム本体のほうでは、 sp.p が IDParticle 型になってるので、 id を変更したりできるわけです。色々してから、 print_particle をします。これは、色々ややこしいことしてますが、要するに IDParticle 型の p にあるデータはそっちで、ないものは y のほうにあるのをそのまま書く、というものです。ただし、 YAML のデータ構造を触るのがあんまり上手くできてなくて、一旦配列にして、また戻すとかしているので、ちょっと汎用性がなくて、今のところ

- 整数
- 実数
- 文字列
- 実数の配列

しか扱えないです。

赤木 ：まあ、そうわかってればいいかしらね。

学生 C：赤木 ：学生 C：赤木 ：学生 C：赤木 ：学生 C：赤木 ：学生 C：

12.1 課題

1. 上の nacsreadwrite3.cr を参考に、位置座標を 2 倍にして出力するプログラムを作って動作を確認せよ。
2. さらに、コマンドラインオプションで指定した値で 位置、速度、質量をスケールするプログラムに拡張してみよ。

12.2 まとめ

1. YAML 型式で粒子データを読み書きするライブラリを作成した。
2. このライブラリは、プログラムの粒子データクラスにはいってないものはそのまま残して、おいて、粒子データクラスと合わせてあとで出力することもできる

12.3 参考資料

module YAML[2]

PSDF 論文[3]

[2] https://crystal-lang.org/api/latest/YAML.html
[3] https://ui.adsabs.harvard.edu/link_gateway/2012NewA...17..520F/EPRINT_PDF

Chapter 13

プラマーモデル

赤木 ：前回はとりあえず粒子データ読み書きだけだったから、今日はなんかそれらしい初期モデルを作りましょう。

学生 C：作りましょうっていわれても、、、

赤木 ：分布関数とか簡単に書けるのはプラマーモデル Plummer model というやつね。これは、球対称で、星の密度部分が、半径とか密度の値を適当にスケールすると

$$\rho = \frac{1}{(r^2+1)^{5/2}} \tag{13.1}$$

で与えられて、重力ポテンシャルが

$$\phi = -\frac{1}{\sqrt{r^2+1}} \tag{13.2}$$

になるのね。で、初期条件の作り方がこの論文[1] の Appendix に詳しく書いてあるから、みんなまずはこれ使うの。

あとはもちろん、 Hernquist モデルとかその一般化とか NFW プロファイルとかその一般化とかあるんだけど、まあとりあえずは。

学生 C：えっと、その論文読んで作るんですか？

赤木 ：あ、それでもいいけど、 Ruby のがあるから、それを Crystal で動くように直すのでもいいわ。 Ruby のはこれ。

```
 1:#!/usr/local/bin/ruby -w
 2:
 3:require "acs.rb"
 4:require "acs"
 5:options_text= <<-END
 6:
 7:  Description: Plummer's Model Builder
 8:  Long description:
 9:    This program creates an N-body realization of Plummer's Model.
10:    (c) 2004, Piet Hut and Jun Makino; see ACS at www.artcompsi.org
```

[1] https://ui.adsabs.harvard.edu/link_gateway/1974A%26A....37..183A/ADS_PDF

```
11:
12:    The algorithm used is described in Aarseth, S., Henon, M., & Wielen, R.,
13:    Astron. Astroph. 37, 183 (1974).
14:
15:
16:  Short name:-n
17:  Long name:            --n_particles
18:  Value type:           int
19:  Default value:        1
20:  Variable name:        n
21:  Print name:           N
22:  Description:          Number of particles
23:  Long description:
24:    Number of particles in a realization of Plummer's Model.
25:
26:    Each particles is drawn at random from the Plummer distribution,
27:    and therefore there are no correlations between the particles.
28:
29:    Standard Units are used in which G = M = 1 and E = -1/4, where
30:      G is the gravitational constant
31:      M is the total mass of the N-body system
32:      E is the total energy of the N-body system
33:
34:
35:  Short name:           -s
36:  Long name:            --seed
37:  Value type:           int
38:  Default value:        0
39:  Description:          pseudorandom number seed given
40:  Print name:
41:  Variable name:      seed
42:  Long description:
43:    Seed for the pseudorandom number generator.  If a seed is given with
44:    value zero, a preudorandom number is chosen as the value of the seed.
45:    The seed value used is echoed separately from the seed value given,
46:    to allow the possibility to repeat the creation of an N-body realization.
47:
48:      Example:
49:
50:        |gravity> kali mkplummer1.rb -n 42 -s 0
51:        . . .
52:        pseudorandom number seed given: 0
53:                    actual seed used: 1087616341
54:        . . .
55:        |gravity> kali mkplummer1.rb -n 42 -s 1087616341
56:        . . .
57:        pseudorandom number seed given: 1087616341
58:                    actual seed used: 1087616341
59:        . . .
60:
61:
62:  END
```

```
 63:
 64:class Body
 65:
 66:  attr_accessor :body_id, :mass, :pos, :vel
 67:
 68:  def ekin                          # kinetic energy
 69:    0.5*@mass*(@vel*@vel)
 70:  end
 71:
 72:  def epot(body_array)              # potential energy
 73:    p = 0
 74:    body_array.each do |b|
 75:      unless b == self
 76:        r = b.pos - @pos
 77:        p += -@mass*b.mass/sqrt(r*r)
 78:      end
 79:    end
 80:    p
 81:  end
 82:
 83:end
 84:
 85:class NBody
 86:
 87:  attr_accessor :time, :body
 88:
 89:  def initialize
 90:    @body = []
 91:  end
 92:
 93:  def ekin                         # kinetic energy
 94:    e = 0
 95:    @body.each{|b| e += b.ekin}
 96:    e
 97:  end
 98:
 99:  def epot                         # potential energy
100:    e = 0
101:    @body.each{|b| e += b.epot(@body)}
102:    e/2                            # pairwise potentials were counted twice
103:  end
104:
105:  def adjust_center_of_mass
106:    vel_com = pos_com = @body[0].pos*0     # null vectors of the correct length
107:    @body.each do |b|
108:      pos_com += b.pos*b.mass
109:      vel_com += b.vel*b.mass
110:    end
111:    @body.each do |b|
112:      b.pos -= pos_com
113:      b.vel -= vel_com
114:    end
```

```
115:  end
116:
117:  def adjust_units
118:    alpha = -epot / 0.5
119:    beta = ekin / 0.25
120:    @body.each do |b|
121:      b.pos *= alpha
122:      b.vel /= sqrt(beta)
123:    end
124:  end
125:
126:end
127:
128:def frand(low, high)
129:  low + rand * (high - low)
130:end
131:
132:def spherical(r)
133:  vector = Vector.new
134:  theta = acos(frand(-1, 1))
135:  phi = frand(0, 2*PI)
136:  vector[0] = r * sin( theta ) * cos( phi )
137:  vector[1] = r * sin( theta ) * sin( phi )
138:  vector[2] = r * cos( theta )
139:  vector
140:end
141:
142:def mkplummer(c)
143:  if c.seed == 0
144:    srand
145:  else
146:    srand c.seed
147:  end
148:  nb = NBody.new
149:  c.n.times do |i|
150:    b = plummer_sample
151:    b.mass = 1.0/c.n
152:    b.body_id = i
153:    nb.body.push(b)
154:  end
155:  nb.adjust_center_of_mass if c.n > 0
156:  nb.adjust_units if c.n > 1
157:  nb.acs_log(1, "             actual seed used\t: #{srand}\n")
158:  nb.acs_write($stdout, false, c.precision, c.add_indent)
159:end
160:
161:def plummer_sample
162:  b = Body.new
163:  scalefactor = 16.0 / (3.0 * PI)
164:  radius = 1.0 / sqrt( rand ** (-2.0/3.0) - 1.0)
165:  b.pos = spherical(radius) / scalefactor
166:  x = 0.0
```

```
 167:  y = 0.1
 168:  while y > x*x*(1.0-x*x)**3.5
 169:    x = frand(0,1)
 170:    y = frand(0,0.1)
 171:  end
 172:  velocity = x * sqrt(2.0) * ( 1.0 + radius*radius)**(-0.25)
 173:  b.vel = spherical(velocity) * sqrt(scalefactor)
 174:  b
 175:end
 176:
 177:
 178:mkplummer(parse_command_line(options_text))
```

学生 C : あー、コマンドラインオプションとかベクトル型とか、同じような感じですね。

赤木 : もちろん、 Ruby のほうが元だから。

学生 C : これならなんとかなるかな。やってみます。

赤木 : お願いね。

学生 C : 一応できて動く気が、、、動かすと

```
gravity> crystal mkplummer.cr -- -n 5 -Y
--- !CommandLog
command: /home/makino/.cache/crystal/crystal-run-mkplummer.tmp -- -n 5 -Y
log: Plummer model created
--- !Particle
id: 0
t: 0.0
m: 0.2
r:
  x: 0.01855609417799849
  y: -0.012221457558339763
  z: 0.22717498173982836
v:
  x: -0.005609548857719633
  y: -0.4152427800709664
  z: 0.34410219575721196
--- !Particle
id: 1
t: 0.0
m: 0.2
r:
  x: 0.24419912181441167
  y: 0.6314777984903683
  z: -0.050871427435152335
v:
  x: -0.14641464545573118
  y: 0.038413721894081126
  z: -0.033473469357231556
--- !Particle
id: 2
```

```
t: 0.0
m: 0.2
r:
  x: -0.12219402354465239
  y: -0.9151740065762606
  z: -0.17931285450581197
v:
  x: -0.397846890032983
  y: 0.030292147641782168
  z: 0.3193987668373569
--- !Particle
id: 3
t: 0.0
m: 0.2
r:
  x: 0.3418377441661433
  y: -0.10345514548999976
  z: 0.05855509875621619
v:
  x: 0.6932651207968673
  y: 0.319933802034601
  z: 0.4337104729340043
--- !Particle
id: 4
t: 0.0
m: 0.2
r:
  x: -0.48239893661390093
  y: 0.3993728111342316
  z: -0.05554579855508031
v:
  x: -0.14339403645043353
  y: 0.0266031085005022
  z: -1.0637379661713415
```

こんな感じです。オプション 2 つあって、 -n で粒子数、 -Y で YAML でだす、という感じです。プログラムは

```
   1:require "clop"
   2:require "./vector3.cr"
   3:require "./nacsio.cr"
   4:include Math
   5:
   6:optionstr = <<-END
   7:
   8:  Description: Plummer's Model Builder
   9:  Long description:
  10:    This program creates an N-body realization of Plummer's Model.
  11:
  12:    Original Ruby code:
  13:    (c) 2004, Piet Hut and Jun Makino; see ACS at www.artcompsi.org
```

```
14:    The algorithm used is described in Aarseth, S., Henon, M., & Wielen, R.,
15:    Astron. Astroph. 37, 183 (1974).
16:
17:    Crystal Version (c) 2020- Jun Makino
18:
19:  Short name:-n
20:  Long name:            --n_particles
21:  Value type:           int
22:  Default value:        10
23:  Variable name:        n
24:  Description:          Number of particles
25:  Long description:
26:    Number of particles in a realization of Plummer's Model.
27:    Each particles is drawn at random from the Plummer distribution,
28:    and therefore there are no correlations between the particles.
29:    Standard Units are used in which G = M = 1 and E = -1/4, where
30:
31:      G is the gravitational constant
32:      M is the total mass of the N-body system
33:      E is the total energy of the N-body system
34:
35:
36:  Short name:           -s
37:  Long name:            --seed
38:  Value type:           int
39:  Default value:        0
40:  Variable name:      seed
41:  Description:  pseudorandom number seed given
42:  Long description:
43:    Seed for the pseudorandom number generator.  If a seed is given with
44:    value zero, a preudorandom number is chosen as the value of the seed.
45:    The seed value used is echoed separately from the seed value given,
46:    to allow the possibility to repeat the creation of an N-body realization.
47:
48:  Short name:           -Y
49:  Long name:            --yaml_io
50:  Value type:           bool
51:  Variable name:      use_nacsio
52:  Description:  use nacs io format (yaml based)
53:  Long description: use nacs io format (yaml based)
54:END
55:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
56:options=CLOP.new(optionstr,ARGV)
57:
58:module Nacsio
59:class Particle
60:  def ekin                        # kinetic energy
61:    0.5*@mass*(@vel*@vel)
62:  end
63:  def epot(body_array)            # potential energy
64:    p = 0
65:    body_array.each{ |b|
```

```
 66:      unless b == self
 67:        r = b.pos - @pos
 68:        p += -@mass*b.mass/sqrt(r*r)
 69:      end
 70:    }
 71:    p
 72:  end
 73:  def write
 74:    printf("%22.15e", @mass)
 75:    @pos.to_a.each { |x| printf("%23.15e", x) }
 76:    @vel.to_a.each { |x| printf("%23.15e", x) }
 77:    print "\n"
 78:  end
 79:end
 80:end
 81:
 82:include Nacsio
 83:
 84:class NBody
 85:
 86:  property :time, :body
 87:  def initialize
 88:    @time = 0.0
 89:    @body = Array(Particle).new
 90:  end
 91:  def ekin                        # kinetic energy
 92:    e = 0
 93:    @body.each{|b| e += b.ekin}
 94:    e
 95:  end
 96:  def epot                        # potential energy
 97:    e = 0
 98:    @body.each{|b| e += b.epot(@body)}
 99:    e/2                           # pairwise potentials were counted twice
100:  end
101:
102:  def adjust_center_of_mass
103:    m_com = @body.reduce(0.0){|s, b| s + b.mass}
104:    pos_com = @body.reduce(Vector3.new){|vec, b| vec + b.pos*b.mass}
105:    vel_com = @body.reduce(Vector3.new){|vec, b| vec + b.vel*b.mass}
106:    pos_com /= m_com
107:    vel_com /= m_com
108:    @body.each do |b|
109:      b.pos -= pos_com
110:      b.vel -= vel_com
111:    end
112:  end
113:
114:  def adjust_units
115:    alpha = -epot / 0.5
116:    beta = ekin / 0.25
117:    @body.each do |b|
```

```
118:      b.pos *= alpha
119:      b.vel /= sqrt(beta)
120:    end
121:  end
122:  def write
123:    print @body.size, "\n"
124:    printf("%22.15e\n", @time)
125:    @body.each do |b| b.write end
126:  end
127:  def nacswrite
128:    @body.each{|b| print b.to_nacs}
129:  end
130:
131:end
132:
133:def spherical(r, ran)
134:  theta = acos(ran.rand(2.0)-1.0)
135:  phi = ran.rand(2*PI)
136:  Vector3.new( r * sin( theta ) * cos( phi ),
137:                     r * sin( theta ) * sin( phi ),
138:                     r * cos( theta ))
139:end
140:
141:def plummer_sample(r)
142:  b = Particle.new
143:  scalefactor = 16.0 / (3.0 * PI)
144:  radius = 1.0 / sqrt( r.rand ** (-2.0/3.0) - 1.0)
145:  b.pos = spherical(radius,r) / scalefactor
146:  x = 0.0
147:  y = 0.1
148:  while y > x*x*(1.0-x*x)**3.5
149:    x = r.rand
150:    y = r.rand(0.1)
151:  end
152:  velocity = x * sqrt(2.0) * ( 1.0 + radius*radius)**(-0.25)
153:  b.vel = spherical(velocity,r) * sqrt(scalefactor)
154:  b
155:end
156:
157:if options.seed == 0
158:  r= Random.new
159:else
160:  r= Random.new(options.seed)
161:end
162:nb = NBody.new
163:options.n.times do |i|
164:  b = plummer_sample(r)
165:  b.mass = 1.0/options.n
166:  b.id = i
167:  nb.body.push(b)
168:end
169:nb.adjust_center_of_mass if options.n > 0
```

```
170:nb.adjust_units if options.n > 1 && options.n < 100000
171:if options.use_nacsio
172:  print CommandLog.new("Plummer model created").to_nacs
173:  nb.nacswrite
174:else
175:  nb.write
176:end
177:
```

です。

赤木 ：解説一応お願いしていい？

学生 C : もとの Ruby のを動くようにしただけなんであんまりわかってないですが、一応やります。

56 行目までは clop とか nacsio.cr とか読み込んで、あとコマンドラインオプションの定義ですね。あ、 -s で乱数のシードを与える、というのもありました。58 行目から 80 行目までは Particle クラスを拡張して、運動エネルギーとポテンシャルエネルギーを計算させてます。ポテンシャルエネルギーは粒子の配列とで全部計算ですね。この辺は、粒子の位置・速度のスケーリングに使ってます。

で、粒子系のクラスというのを作ってあって、それが 84 行目からですね。これは全体の運動エネルギー、ポテンシャルエネルギーを計算する関数 ekin, epot とか、重心速度、重心位置を原点にもってくる adjust_pos、ポテンシャルエネルギーと運動エネルギーをそれぞれ -0.5, 0.25 になるようにスケールする ajust_units と、1 行 1 粒子とか YAML とかで書く write, nacswrite ですう。中身はまあみての通りで。

spherical は球面上の一様乱数ですかね。ここで、 rand は Crystal の乱数で、引数に実数を与えると 0 からそこまでの乱数になります。

plummer_sample が実際にプラマーモデルの分配に従って粒子 1 つを生成する関数です。あんまりわかってないですが論文通りなんじゃないかと、、、元の Ruby のプログラムからあんまりここいじってなくて、乱数の関数の書き方かえたくらいなので。

あとは、157 行からの if で乱数の初期化をして、163 行からで n 個粒子作って、あと重心を 0 にとかエネルギーのスケーリングとかして出力、という感じですね。

赤木 ：これ、粒子数増やしてプロットできる？

学生 C : オプションでですか？

赤木 ：あ、じゃなくて、折角だからこの出力ファイルを読んでプロットするのにして。プロットするの自体は前にやってるわよね」？

学生 C : ですね。読むのもさっきやったから、、、こんなのですね。

```
crystal mkplummer.cr -- -n 1000 -Y | nacsplot
```

で作ってて、nacsplot は

```
  1:require "grlib"
  2:require "clop"
  3:require "./nacsio.cr"
  4:include Math
  5:include GR
  6:include Nacsio
  7:
  8:optionstr= <<-END
```

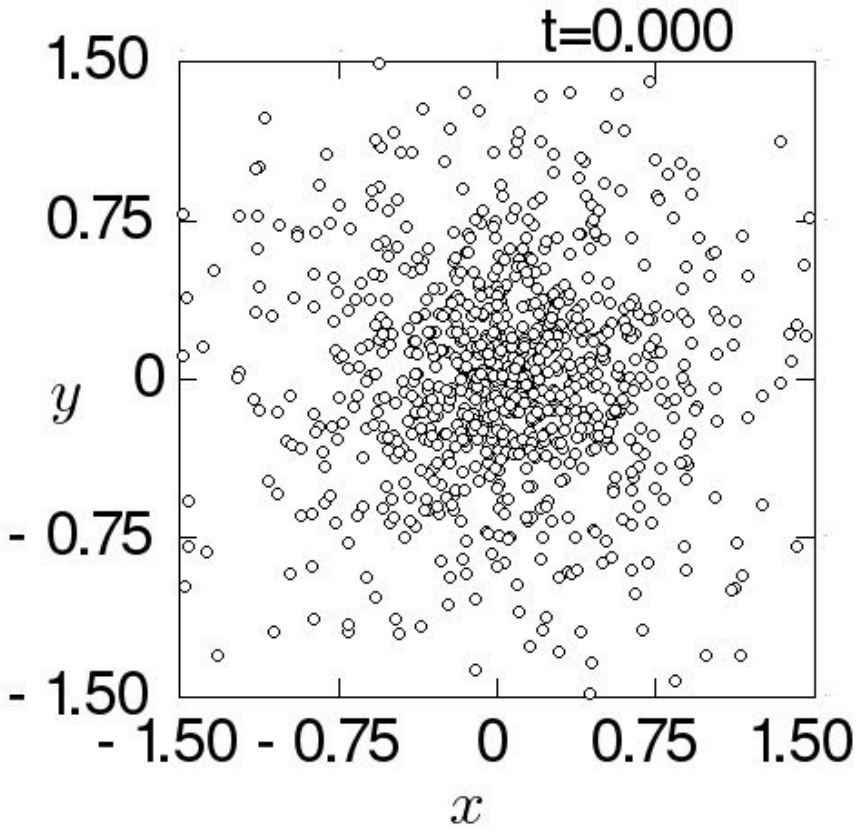

Figure 13.1: 1000 粒子プラマーモデルの出力

```
 9:  Description: Plot program for nacs snapshot
10:  Long description: Plot program for nacs snapshot
11:
12:  Short name:-w
13:  Long name:  --window-size
14:  Value type:  float
15:  Variable name:wsize
16:  Default value:1.5
17:  Description:Window size for plotting
18:  Long description:
19:    Window size for plotting orbit. Window is [-wsize, wsize] for both of
20:    x and y coordinates
21:END
22:
23:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
24:options=CLOP.new(optionstr,ARGV)
25:update_commandlog
26:pp=Array(Particle).new
27:while (sp= CP(Particle).read_particle).y != nil
28:  pp.push sp.p
29:end
30:wsize=options.wsize
31:setwindow(-wsize, wsize,-wsize, wsize)
32:setcharheight(0.05)
33:box
34:mathtex(0.5, 0.06, "x")
35:mathtex(0.06, 0.5, "y")
36:text(0.6,0.91,"t="+sprintf("%.3f",pp[0].time))
37:setmarkertype(4)
38:setmarkersize(1)
39:polymarker(pp.map{|p| p.pos[0]}, pp.map{|p| p.pos[1]})
40:c=STDERR.gets
```

です。説明いります？だいたいみての通りですが、、、

赤木 ：お願いね。

学生 C : 24 行目までは色々なライブラリとかと、あとコマンドラインオプションです。ここでは -w でプロットする座標範囲、というのだけ残してます。

で、CommandLog 読まないといけないので update_commandlog を読んで、それから 26 行目から 29 行目で粒子の配列を作ります。このプログラムは粒子データ出力しないので、Particle クラスのほうだけとって配列にいれます。

30 行目からは前の particle1.cr かなんかと同じです。あ、最後の、入力待つところ、標準入力は粒子データのリダイレクトにしているので、それでもキーボードの入力待ちになるように STDERR にしてます。

赤木 ：こういうのがあると、 x, y 座標に色々指定できるようにしたくなるわね。

学生 C : コマンドラインで式を与えるとかですか？そういうのは Ruby とか Python でしたほうがいいものではないですか？

赤木 ：でもまあ 100 倍の速度の違いは問題で、そうするとやっぱり大きなデータ使うと遅いとかになっちゃうから、、、

学生 C：ちょっと話が長くなりそうなので次回以降にしませんか？

赤木 ：そうね。

13.1 課題

1. mkplummer.cr で生成した粒子分布の半径方向の密度分布が理論分布と一致していることを確認せよ。このために、粒子分布から半径方向の累積密度分布を作成し、理論分布との最大誤差を計算するプログラムを作成せよ。
2. さらに、コルモゴロフ・スミルノフ検定を行え。
3. 粒子分布から慣性モーメントテンソルを計算し、その結果から球対称といえるかどうかを検討せよ。 どのような統計的検定が可能か考えてみよ。
4. 速度分布について同様な検討を行え。特に、半径方向と角度方向の異方性がないかどうかを調べよ。

累積密度分布は、半径方向に粒子をソートして、各粒子の位置までで内側にある質量を求めればよい。ソートには Array#sort! を使う。 sort を使うには、演算子 ¡=¿ を定義するか、あるいはそれに相当するブロックを渡す。以下はブロックを渡す例である。

```
data= [9,6,7,5,4,8,2,3,0,1]
data.sort!{|x,y|
  if x>y
    1
  elsif x<y
    -1
  else
    0
  end }
p! data
```

等しいなら 0、大小に応じて 負または正の値を返すようにする。

13.2 まとめ

1. Plummer model の分布関数に合わせて粒子を生成するプログラムを作成した。
2. PSDF 型式のファイルを読んで、粒子分布をプロットするプログラムを作った。

13.3 参考資料

The Art of Computational Science[2]

Aarseth et al 1974[3]

[2] http://www.artcompsci.org

[3] https://ui.adsabs.harvard.edu/link_gateway/1974A%26A....37..183A/ADS_PDF

Chapter 14

Barnes-Hut treecode

赤木 ：前回、もうちょっと色々プロット、みたいな話をしたけど、その前に時間積分できないと面白くないので、時間積分のプログラムつくりましょう。

学生 C : 前に作った particle1.cr でよくないですか？

赤木 ：あれはもちろんあれで動くんだけど、粒子数が多いと粒子数の自乗に比例して時間かかるからあんまり大きな粒子数扱えないじゃない？なので、今回は 11.4 で話がでてきた Barnes-Hut ツリー法を書いてね、と。

学生 C : 書いてね、いわれても、、、

赤木 ：まあこれも、作者が昔 Ruby で書いたのがあって、それは Josh Barnes が C で書いたのが元になってるの。なので、それを Crystal にしてみて。で、ファイル入出力は PSDF で。

学生 C : はあ、、、

赤木 ： Ruby のはこれ:

```
 1:require "../command_line/clop.rb"
 2:
 3:def get_self_other_acc(myself, other, eps)
 4:  return @pos*0 if myself == other
 5:  rji = myself.pos - other.pos
 6:  r2 = eps * eps + rji * rji
 7:  myself.mass * rji / (r2 * sqrt(r2))
 8:end
 9:
10:class Body
11:
12:  attr_accessor :mass, :pos, :vel, :acc
13:
14:  def pairwise_acc(other, eps)
15:    rji = other.pos - @pos
16:    r2 = eps * eps + rji * rji
17:    r = sqrt(r2)
18:    r3 = r * r2
19:    da = rji / r3
20:    self.acc += other.mass * da
21:    other.acc -= self.mass * da
```

```
  end

  def ekin                          # kinetic energy
    0.5*@mass*(@vel*@vel)
  end

  def epot(body_array, eps)         # potential energy
    p = 0
    body_array.each do |b|
      unless b == self
        r = b.pos - @pos
        p += -@mass*b.mass/sqrt(r*r + eps*eps)
      end
    end
    p
  end

  def get_node_acc(other, tol, eps)
    get_self_other_acc(self, other, eps)
  end

  def to_s
    "mass = " + @mass.to_s +
    "   pos = " + @pos.join(", ") +
    "   vel = " + @vel.join(", ")
  end

  def pp(indent = 0)                # pretty print
    print " "*indent + to_s + "\n"
  end

  def ppx(body_array, eps)          # pretty print, with extra information (acc)
    STDERR.print to_s + "   acc = " + @acc.join(", ") + "\n"
  end

  def write
    printf("%22.15e", @mass)
    @pos.each do |x| printf("%23.15e", x) end
    @vel.each do |x| printf("%23.15e", x) end
    print "\n"
  end

  def read
    a = gets.split.collect{|x| x.to_f}
    @mass = a[0]
    @pos = a[1..3].to_v
    @vel = a[4..6].to_v
  end

end

class NBody
```

```
 74:
 75:  attr_accessor :time, :body, :rootnode
 76:
 77:  def initialize(n=0, time = 0.0)
 78:    @body = [Body.new]
 79:    for i in 0...n
 80:      @body[i] = Body.new
 81:    end
 82:    @time = time
 83:  end
 84:
 85:  def evolve(tol, eps, dt, dt_dia, dt_out, dt_end, init_out, x_flag)
 86:    @dt = dt
 87:    @tol = tol
 88:    @eps = eps
 89:    @nsteps = 0
 90:    get_acc
 91:    e_init
 92:    write_diagnostics(x_flag)
 93:    t_dia = dt_dia - 0.5*dt
 94:    t_out = dt_out - 0.5*dt
 95:    t_end = dt_end - 0.5*dt
 96:
 97:    write if init_out
 98:
 99:    while @time < t_end
100:      leapfrog
101:      @time += @dt
102:      @nsteps += 1
103:      if @time >= t_dia
104:        write_diagnostics(x_flag)
105:        t_dia += dt_dia
106:      end
107:      if @time >= t_out
108:        write
109:        t_out += dt_out
110:      end
111:    end
112:  end
113:
114:  def leapfrog
115:    @body.each do |b|
116:      b.vel += b.acc*0.5*@dt
117:      b.pos += b.vel*@dt
118:    end
119:#    get_acc
120:    get_tree_acc
121:    @body.each do |b|
122:      b.vel += b.acc*0.5*@dt
123:    end
124:  end
125:
```

```
126:  def get_acc
127:    @body.each{|b|b.acc = b.pos*0}
128:    i = 0
129:    while (i < @body.size)
130:      j = i+1
131:      while (j < @body.size)
132:@body[i].pairwise_acc(@body[j], @eps)
133:j += 1
134:      end
135:      i += 1
136:    end
137:  end
138:
139:  def get_tree_acc
140:    maketree
141:    @rootnode.center_of_mass
142:#    @rootnode.pp(0)
143:    @body.each{|b| b.acc = @rootnode.get_node_acc(b, @tol, @eps)}
144:  end
145:
146:  def ekin                          # kinetic energy
147:    e = 0
148:    @body.each{|b| e += b.ekin}
149:    e
150:  end
151:
152:  def epot                          # potential energy
153:    e = 0
154:    @body.each{|b| e += b.epot(@body, @eps)}
155:    e/2                             # pairwise potentials were counted twice
156:  end
157:
158:  def e_init                        # initial total energy
159:    @e0 = ekin + epot
160:  end
161:
162:  def write_diagnostics(x_flag)
163:    etot = ekin + epot
164:    STDERR.print <<END
165:at time t = #{sprintf("%g", time)}, after #{@nsteps} steps :
166:  E_kin = #{sprintf("%.3g", ekin)} ,\
167: E_pot =  #{sprintf("%.3g", epot)} ,\
168: E_tot = #{sprintf("%.3g", etot)}
169:             E_tot - E_init = #{sprintf("%.3g", etot - @e0)}
170:  (E_tot - E_init) / E_init = #{sprintf("%.3g", (etot - @e0)/@e0 )}
171:END
172:    if x_flag
173:      STDERR.print "  for debugging purposes, here is the internal data ",
174:                   "representation:\n"
175:      ppx
176:    end
177:  end
```

```
178:
179:  def pp                                  # pretty print
180:    print "     N = ", @body.size, "\n"
181:    print "  time = ", @time, "\n"
182:    @body.each do |b| b.pp end
183:  end
184:
185:  def ppx                             # pretty print, with extra information (acc)
186:    print "     N = ", @body.size, "\n"
187:    print "  time = ", @time, "\n"
188:    @body.each{|b| b.ppx(@body, @eps)}
189:  end
190:
191:  def write
192:    print @body.size, "\n"
193:    printf("%22.15e\n", @time)
194:    @body.each do |b| b.write end
195:  end
196:
197:  def read
198:    n = gets.to_i
199:    @time = gets.to_f
200:    for i in 0...n
201:      @body[i] = Body.new
202:      @body[i].read
203:    end
204:  end
205:
206:  def maketree
207:    @rootnode = makerootnode
208:    @body.each do |b|
209:      @rootnode.loadtree(b)
210:    end
211:  end
212:
213:  def makerootnode
214:    r = @body.inject(0){|oldmax, b| [oldmax, b.pos.map{|x| x.abs}.max].max}
215:    s = 1
216:    s *= 2 while r > s
217:    Node.new([0.0, 0.0, 0.0], s)
218:  end
219:
220:end
221:
222:class Node
223:
224:  attr_accessor :mass, :pos
225:
226:  def initialize(center, size)
227:    @center, @size = center.to_v, size
228:    @child = Array.new(8)
229:  end
```

```
230:
231:  def octant(pos)
232:    result = 0
233:    pos.each_index do |i|
234:      result *= 2
235:      result += 1 if pos[i] > @center[i]
236:    end
237:    result
238:  end
239:  def loadtree(b : Body)
240:#    print "loadtree for center,size= #{@center}, #{@size}\n"
241:    corner = octant(b.pos)
242:#    print "new octant=#{corner}\n"
243:    if @child[corner] == nil
244:      @child[corner] = b
245:#      p "inserted\n"
246:      return
247:    end
248:    if @child[corner].class == Body
249:      tmp_b = @child[corner]
250:      child_size = @size / 2.0
251:      @child[corner] = Node.new(@center + child_size*offset(corner),child_size)
252:      @child[corner].loadtree(tmp_b)
253:#      p "new cell made\n"
254:    end
255:#    p "recursive call"
256:    @child[corner].loadtree(b)
257:  end
258:  def offset(corner)
259:    r=[]
260:    3.times{ r.unshift( (corner & 1)*2 - 1 ) ; corner>>=1 }
261:    r.to_v
262:  end
263:
264:  def pp(indent = 0)
265:    print " "*indent+"node: center = #{@center.join(" ")} ; size = #{@size}\n"
266:    if @mass
267:      print " "*indent+"      mass = #{@mass}   pos = #{@pos.join(", ")}\n"
268:    end
269:    @child.each{|c| c.pp(indent + 2) if c}
270:  end
271:
272:  def check_body_in_cell
273:    @child.each do |c|
274:      if c.class == Body
275:        (c.pos - @center).each do |x|
276:          raise("\nbody out of cell:\n#{c.to_s}\n") if x.abs > @size
277:        end
278:      elsif c.class == Node
279:        c.check_body_in_cell
280:      end
281:    end
```

```
282:  end
283:
284:  def center_of_mass
285:    @mass = 0
286:    @pos = [0, 0, 0].to_v
287:    @child.each do |c|
288:      c.center_of_mass if c.class == Node
289:      if c
290:        @mass += c.mass
291:        @pos += c.mass * c.pos
292:      end
293:    end
294:    @pos /= @mass
295:  end
296:
297:  def get_node_acc(b, tol, eps)
298:    distance = b.pos - @pos
299:    if 2 * @size > tol * sqrt(distance*distance)
300:      acc = @pos*0
301:      @child.each{|c| acc += c.get_node_acc(b, tol, eps) if c}
302:      acc
303:    else
304:      get_self_other_acc(self, b, eps)
305:    end
306:  end
307:
308:end
309:
310:options_text = <<-END
311:
312:  Description: First very simple version of Barnes-Hut tree code
313:  Long description:
314:    First very simple version of Barnes-Hut tree code
315:
316:    (c) 2005, Piet Hut and Jun Makino; see ACS at www.artcompsi.org
317:
318:    example:
319:    ruby #{$0} -t 1 < cube1.in
320:
321:
322:  Short name: -T
323:  Long name:--opening_tolerance
324:  Value type:float
325:  Default value: 0.5
326:  Global variable: tol
327:  Description:Opening tolerance
328:  Long description:
329:    This option sets the tolerance value that governs the maximum size
330:    of a tree cell that can remain closed; cells (nodes) with a size
331:    large than the product of tolerance and distance to that cell will
332:    be opened, and acceleration to its children will be computed.
333:
```

```
334:
335:  Short name: -s
336:  Long name:--softening_length
337:  Value type:float
338:  Default value: 0.0
339:  Global variable: eps
340:  Description:Softening length
341:  Long description:
342:    This option sets the softening length used to calculate the force
343:    between two particles.  The calculation scheme comforms to standard
344:    Plummer softening, where rs2=r**2+eps**2 is used in place of r**2.
345:
346:
347:  Short name: -c
348:  Long name:--step_size
349:  Value type:float
350:  Default value:0.001
351:  Global variable:dt
352:  Description:Time step size
353:  Long description:
354:    This option sets the size of the time step, which is constant and
355:    shared by all particles.  It is wise to use option -s to specify a
356:    softening length that is significantly larger than the time step size.
357:
358:
359:  Short name: -d
360:  Long name:--diagnostics_interval
361:  Value type:float
362:  Default value:1
363:  Global variable:dt_dia
364:  Description:Interval between diagnostics output
365:  Long description:
366:    The time interval between successive diagnostics output.
367:    The diagnostics include the kinetic and potential energy,
368:    and the absolute and relative drift of total energy, since
369:    the beginning of the integration.
370:        These diagnostics appear on the standard error stream.
371:    For more diagnostics, try option "-x" or "--extra_diagnostics".
372:
373:
374:  Short name: -o
375:  Long name:--output_interval
376:  Value type:float
377:  Default value:1
378:  Global variable:dt_out
379:  Description:Time interval between snapshot output
380:  Long description:
381:    The time interval between output of a complete snapshot
382:    A snapshot of an N-body system contains the values of the
383:    mass, position, and velocity for each of the N particles.
384:
385:        This information appears on the standard output stream,
```

```
386:    currently in the following simple format (only numbers):
387:
388:      N:            number of particles
389:      time:         time
390:      mass:         mass of particle #1
391:      position:     x y z : vector components of position of particle #1
392:      velocity:     vx vy vz : vector components of velocity of particle #1
393:      mass:         mass of particle #2
394:      ...:          ...
395:
396:    Example:
397:
398:       2
399:       0
400:       0.5
401:      7.3406783488452532e-02  2.1167291484119417e+00 -1.4097856092768946e+00
402:      3.1815484836541341e-02  2.7360312082526089e-01  2.4960049959942499e-02
403:       0.5
404:     -7.3406783488452421e-02 -2.1167291484119413e+00  1.4097856092768946e+00
405:     -3.1815484836541369e-02 -2.7360312082526095e-01 -2.4960049959942499e-02
406:
407:
408:  Short name: -t
409:  Long name:--duration
410:  Value type:float
411:  Default value:10
412:  Global variable:dt_end
413:  Description:Duration of the integration
414:  Long description:
415:    This option sets the duration t of the integration, the time period
416:    after which the integration will halt.  If the initial snapshot is
417:    marked to be at time t_init, the integration will halt at time
418:    t_final = t_init + t.
419:
420:
421:  Short name:-i
422:  Long name:  --init_out
423:  Value type:  bool
424:  Global variable: init_out
425:  Description:Output the initial snapshot
426:  Long description:
427:    If this flag is set to true, the initial snapshot will be output
428:    on the standard output channel, before integration is started.
429:
430:
431:  Short name:-x
432:  Long name:  --extra_diagnostics
433:  Value type:  bool
434:  Global variable:x_flag
435:  Description:Extra diagnostics
436:  Long description:
437:    If this flag is set to true, the following extra diagnostics
```

```
438:    will be printed:
439:
440:      acceleration (for all integrators)
441:
442:
443:  END
444:
445:parse_command_line(options_text)
446:
447:include Math
448:
449:nb = NBody.new
450:nb.read
451:nb.evolve($tol, $eps, $dt, $dt_dia, $dt_out, $dt_end, $init_out, $x_flag)
```

学生 C：まあ、、、やってみます。

とりあえずみてみるか。class Body は、、、

```
attr_accessor :mass, :pos, :vel, :acc
```

はまた

```
YAML.mapping(
  id: {type: Int64, default: 0i64,},
  time: {type: Float64, key: "t", default: 0.0,},
  mass: {type: Float64, key: "m",default: 0.0,},
  pos: {type: Vector3, key: "r",default: [0.0,0.0,0.0].to_v,},
  vel: {type: Vector3, key: "v",default: [0.0,0.0,0.0].to_v,},
  acc: {type: Vector3, key: "a",default: [0.0,0.0,0.0].to_v,},
  pot: {type: Float64, default: 0.0,},
)
```

に置き換えて、入出力もそっちになるとして、28 行目の epot、これ tree 使いようになってなくないか？ツリー使うって加速度計算するとこでポテンシャルも計算しないと駄目だこれ。

49 行目の pp とかその次の ppx とかは Crystal なら pp! ですむからいらないか。 57 行目からの read/write も、nacsio の関数使うから不要と。

class NBody は、、、 initialize はいるのか？粒子配列は型だけ与えて長さ 0 でいいんじゃないかなあ？どうせファイルから読むし。そのあとの evolve, leapfrog はコンパイル通れば大丈夫そうかな。get_acc これは使って、、、あ、ちゃんと get_tree_acc でポテンシャル計算してこっち使わないと駄目か。なのでこれはいらないと。

get_tree_acc は、get_tree_acc_and_pot かなんかにして加速度とポテンシャルと両方計算するようにすればいいかな。

あと、179 行目の pp, ppx はやっぱり pp! でいいと。191 行目からの write/read は nacsio 使うからだいぶ変更必要だな。あとはまあコンパイラが文句いわなければ大丈夫かなあ、、、

ではまあやってみますか。

(数日後)

学生 C：なんかできたみたいですが、、、実行すると

```
gravity> crystal mkplummer.cr -- -n 100 -s 1 -Y | crystal hackcode1.cr
[2mShowing last frame. Use --error-trace for full trace.[0m

In [4mlib/nacsio/src/nacsio.cr:21:10[0m

[2m 21 | [0m[1mYAML.mapping([0m
           [32;1m^------[0m
[33;1mError: undefined method 'mapping' for YAML:Module[0m
[2mShowing last frame. Use --error-trace for full trace.[0m

In [4mlib/clop/src/clop.cr:25:8[0m

[2m 25 | [0m[1m{{system("sh ./lib/clop/src/clop_process.sh #{l} #{f} #{d} #{strname}")}}[0m
        [32;1m^-----[0m
[33;1mError: error executing command: sh ./lib/clop/src/clop_process.sh 109 "/home/makino/papers/intro
```

で、エネルギーはまあまあ保存してます。

赤木 ：プログラムはどんな感じ？

学生 C：以下ですが、

```
    1:require "clop"
    2:include Math
    3:require "nacsio"
    4:include Nacsio
    5:
    6:optionstr = <<-END
    7:  Description: First very simple version of Barnes-Hut tree code
    8:  Long description:
    9:    First very simple version of Barnes-Hut tree code
   10:    Crystal version - (c) 2020- Jun Makino
   11:    Original Ruby version -
   12:    (c) 2005, Piet Hut and Jun Makino. see ACS at www.artcompsi.org
   13:    example
   14:    hackcode1 < cube1.in
   15:
   16:  Short name: -T
   17:  Long name:--opening_tolerance
   18:  Value type:float
   19:  Default value: 0.5
   20:  Variable name: tol
   21:  Description:Opening tolerance
   22:  Long description:
   23:    This option sets the tolerance value that governs the maximum size
   24:    of a tree cell that can remain closed; cells (nodes) with a size
   25:    large than the product of tolerance and distance to that cell will
   26:    be opened, and acceleration to its children will be computed.
   27:
   28:  Short name: -s
   29:  Long name:--softening_length
   30:  Value type:float
```

```
31:  Default value: 0.05
32:  Variable name: eps
33:  Description:Softening length
34:  Long description:
35:    This option sets the softening length used to calculate the force
36:    between two particles.  The calculation scheme comforms to standard
37:    Plummer softening, where rs2=r**2+eps**2 is used in place of r**2.
38:
39:  Short name: -c
40:  Long name:--step_size
41:  Value type:float
42:  Default value:0.0078125
43:  Variable name:dt
44:  Description:Time step size
45:  Long description:
46:    This option sets the size of the time step, which is constant and
47:    shared by all particles.  It is wise to use option -s to specify a
48:    softening length that is significantly larger than the time step size.
49:
50:
51:  Short name: -d
52:  Long name:--diagnostics_interval
53:  Value type:float
54:  Default value:0.25
55:  Variable name:dt_dia
56:  Description:Interval between diagnostics output
57:  Long description:
58:    The time interval between successive diagnostics output.
59:    The diagnostics include the kinetic and potential energy,
60:    and the absolute and relative drift of total energy, since
61:    the beginning of the integration.
62:        These diagnostics appear on the standard error stream.
63:    For more diagnostics, try option "-x" or "--extra_diagnostics".
64:
65:  Short name: -o
66:  Long name:--output_interval
67:  Value type:float
68:  Default value:2
69:  Variable name:dt_out
70:  Description:Time interval between snapshot output
71:  Long description:
72:    The time interval between output of a complete snapshot
73:    A snapshot of an N-body system contains the values of the
74:    mass, position, and velocity for each of the N particles.
75:
76:  Short name: -t
77:  Long name:--duration
78:  Value type:float
79:  Default value:1
80:  Variable name:dt_end
81:  Description:Duration of the integration
82:  Long description:
```

```
 83:    This option sets the duration t of the integration, the time period
 84:    after which the integration will halt.  If the initial snapshot is
 85:    marked to be at time t_init, the integration will halt at time
 86:    t_final = t_init + t.
 87:
 88:  Short name:-i
 89:  Long name:  --init_out
 90:  Value type:  bool
 91:  Variable name: init_out
 92:  Description:Output the initial snapshot
 93:  Long description:
 94:    If this flag is set to true, the initial snapshot will be output
 95:    on the standard output channel, before integration is started.
 96:
 97:  Short name:-x
 98:  Long name:  --extra_diagnostics
 99:  Value type:  bool
100:  Variable name:x_flag
101:
102:  Description:Extra diagnostics
103:  Long description:
104:    If this flag is set to true, the following extra diagnostics
105:    will be printed;
106:      acceleration (for all integrators)
107:END
108:
109:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
110:options=CLOP.new(optionstr,ARGV)
111:
112:struct AccPot
113:  property :a, :p
114:  def initialize(a : Vector3 = Vector3.new, p : Float64 = 0.0)
115:    @a=a; @p=p
116:  end
117:  def +(other : AccPot)
118:    AccPot.new(@a+other.a, @p+other.p)
119:  end
120:end
121:
122:class Body
123:
124:  YAML.mapping(
125:    id: {type: Int64, default: 0i64,},
126:    time: {type: Float64, key: "t", default: 0.0,},
127:    mass: {type: Float64, key: "m",default: 0.0,},
128:    pos: {type: Vector3, key: "r",default: [0.0,0.0,0.0].to_v,},
129:    vel: {type: Vector3, key: "v",default: [0.0,0.0,0.0].to_v,},
130:    acc: {type: Vector3, key: "a",default: [0.0,0.0,0.0].to_v,},
131:    pot: {type: Float64, default: 0.0,},
132:  )
133:
134:  def get_other_acc_and_pot( other, eps)
```

```
135:    return AccPot.new if self == other
136:    rji = @pos  - other.pos
137:    r2 = eps * eps + rji * rji
138:    rinv = 1.0/sqrt(r2)
139:    AccPot.new(@mass * rji*rinv*rinv*rinv, -@mass *rinv)
140:  end
141:
142:  def ekin                          # kinetic energy
143:    0.5*@mass*(@vel*@vel)
144:  end
145:
146:  def get_node_acc_and_pot(other, tol, eps)
147:    get_other_acc_and_pot(other, eps)
148:  end
149:
150:  def loadtree(b)  exit  end
151:
152:  def center_of_mass
153:    @pos
154:  end
155:
156:end
157:
158:class NBody
159:  property :time, :body, :rootnode, :ybody
160:
161:  def initialize(time = 0.0)
162:    @body = Array(Body).new
163:    @ybody= Array(YAML::Any).new
164:    @time = time
165:    @eps = 0.0
166:    @e0=0.0
167:    @dt=0.015625
168:    @rootnode = Node.new([0.0,0.0,0.0].to_v, 1.0)
169:    @tol = 0.0
170:    @nsteps = 0
171:  end
172:
173:  def evolve(tol : Float64,
174:             eps : Float64,
175:             dt : Float64, dt_dia : Float64,
176:     dt_out : Float64, dt_end  : Float64,
177:     init_out, x_flag)
178:    @dt = dt
179:    @tol = tol
180:    @eps = eps
181:    @nsteps = 0
182:    get_tree_acc
183:    e_init
184:    write_diagnostics(x_flag)
185:    t_dia = dt_dia - 0.5*dt
186:    t_out = dt_out - 0.5*dt
```

```
187:    t_end = dt_end - 0.5*dt
188:    write if init_out
189:    while @time < t_end
190:      leapfrog
191:      @time += @dt
192:      @nsteps += 1
193:      if @time >= t_dia
194:        write_diagnostics(x_flag)
195:        t_dia += dt_dia
196:      end
197:      if @time >= t_out
198:        write
199:        t_out += dt_out
200:      end
201:    end
202:  end
203:
204:  def leapfrog
205:    @body.each do |b|
206:      b.vel += b.acc*0.5*@dt
207:      b.pos += b.vel*@dt
208:    end
209:    get_tree_acc
210:    @body.each do |b|
211:      b.vel += b.acc*0.5*@dt
212:    end
213:  end
214:
215:  def get_tree_acc
216:    maketree
217:    @rootnode.center_of_mass
218:    @body.each{|b|
219:      #      b.acc = @rootnode.get_node_acc(b, @tol, @eps)
220:      ap= @rootnode.get_node_acc_and_pot(b, @tol, @eps)
221:      b.acc=ap.a
222:      b.pot = ap.p
223:    }
224:  end
225:
226:  def ekin                          # kinetic energy
227:    e = 0.0
228:    @body.each{|b| e += b.ekin}
229:    e
230:  end
231:
232:  def epot                          # potential energy
233:    e = 0.0
234:    @body.each{|b| e += b.pot*b.mass}
235:    e/2                             # pairwise potentials were counted twice
236:  end
237:
238:  def e_init                        # initial total energy
```

```
239:    @e0 = ekin + epot
240:  end
241:
242:  def write_diagnostics(x_flag)
243:    etot = ekin + epot
244:    STDERR.print <<-EOF
245:at time t = #{sprintf("%g", time)}, after #{@nsteps} steps :
246:  e_kin = #{sprintf("%.3g", ekin)}, \
247: e_pot =  #{sprintf("%.3g", epot)}, \
248: e_tot = #{sprintf("%.3g", etot)}
249:             e_tot - e_init = #{sprintf("%.3g", etot - @e0)}
250:  (e_tot - e_init) / e_init = #{sprintf("%.3g", (etot - @e0)/@e0 )}\n
251:EOF
252:
253:    if x_flag
254:      STDERR.print "  for debugging purposes, here is the internal data ",
255:                   "representation:\n"
256:      pp! self
257:    end
258:  end
259:
260:  def write
261:    @body.size.times{|i|
262:      @body[i].time = @time
263:      CP.new(@body[i], @ybody[i]).print_particle
264:    }
265:  end
266:  def read
267:    update_commandlog
268:    while (sp= CP(Body).read_particle).y != nil
269:      @body.push sp.p
270:      @ybody.push sp.y
271:    end
272:    @time = @body[0].time
273:  end
274:
275:  def makerootnode : Node
276:    r = @body.reduce(0){|oldmax, b| [oldmax, b.pos.to_a.map{|x| x.abs}.max].max}
277:    s = 1.0
278:    while r > s
279:        s *= 2
280:    end
281:    Node.new([0.0, 0.0, 0.0].to_v, s)
282:  end
283:  def maketree
284:    @rootnode = self.makerootnode
285:    i=0
286:    @body.each do |b|
287:#      print "loading body #{i} #{b.pos}\n"
288:      @rootnode.loadtree(b)
289:      i+=1
290:    end
```

```
291:  end
292:
293:end
294:
295:class Node
296:  property :mass, :pos
297:
298:  def initialize(center : Vector3, size : Float64)
299:    @child = Array(Node|Body|Nil).new(8,nil)
300:    @pos = Vector3.new
301:    @mass=0.0
302:    @center, @size = center, size
303:  end
304:
305:  def get_other_acc_and_pot( other, eps)
306:    return AccPot.new if self == other
307:    rji = @pos  - other.pos
308:    r2 = eps * eps + rji * rji
309:    rinv = 1.0/sqrt(r2)
310:    AccPot.new(@mass * rji*rinv*rinv*rinv, -@mass *rinv)
311:  end
312:
313:  def octant(pos)
314:    result = 0
315:    p=pos.to_a
316:    c=@center.to_a
317:    p.each_index do |i|
318:      result *= 2
319:      result += 1 if p[i] > c[i]
320:    end
321:    result
322:  end
323:
324:  def loadtree(b : Node)
325:  end
326:  def loadtree(b : Nil)
327:  end
328:  def loadtree(b : Body)
329:    corner = octant(b.pos)
330:    c=@child[corner]
331:    unless c
332:      @child[corner] = b
333:    else
334:      if @child[corner].is_a?(Body)
335:        tmp_b = @child[corner]
336:        child_size = @size / 2.0
337:        c = Node.new(@center + child_size*offset(corner),child_size)
338:        c.loadtree(tmp_b)
339:        @child[corner]=c
340:      end
341:      c.loadtree(b)
342:    end
```

```
343:  end
344:
345:  def offset(corner)
346:    r=[] of Float64
347:    3.times{ r.unshift( ((corner & 1)*2 - 1 )+0.0) ; corner>>=1 }
348:    r.to_v
349:  end
350:
351:  def check_body_in_cell
352:    @child.each do |c|
353:      if c.is_a?(Body)
354:        (c.pos - @center).each do |x|
355:          raise("\nbody out of cell:\n#{c.to_s}\n") if x.abs > @size
356:        end
357:      elsif c.is_a?(Node)
358:        c.check_body_in_cell
359:      end
360:    end
361:  end
362:
363:  def center_of_mass
364:    @mass = 0.0
365:    @pos = [0.0, 0.0, 0.0].to_v
366:    @child.each do |c|
367:      if c
368:        c.center_of_mass if c.is_a?(Node)
369:        @mass += c.mass
370:        @pos += c.mass * c.pos
371:      end
372:    end
373:    @pos /= @mass
374:  end
375:
376:  def get_node_acc_and_pot(b, tol, eps)
377:    distance = b.pos - @pos
378:    if 2 * @size > tol * sqrt(distance*distance)
379:      ap = AccPot.new
380:      @child.each{|c| ap += c.get_node_acc_and_pot(b, tol, eps)  if c }
381:      ap
382:    else
383:      self.get_other_acc_and_pot(b, eps)
384:    end
385:  end
386:
387:end
388:
389:nb = NBody.new
390:nb.read
391:STDERR.print "after nb.read\n"
392:nb.evolve(options.tol, options.eps, options.dt,options.dt_dia,
393:          options.dt_out, options.dt_end, options.init_out, options.x_flag)
```

基本的に Ruby のをコンパイルできるようにちょっと変えただけなので、あんまり独自にどうというのはありません。

大きく違うのは、まず、 112 行目あたりで、 AccPot という struct を作って、加算も定義して、加速度とポテンシャルを 1 つの関数で返すことができてそれを加算もできるようにしました。この型があるとだとポテンシャル返すものでも、ツリーアルゴリズムの基本である、あるノードからの相互作用は子ノードからの相互作用の合計、というのがそのまま書けるので。

あと、Nacsio を使うので、 read が

```
def read
  update_commandlog
  while (sp= CP(Body).read_particle).y != nil
    @body.push sp.p
    @ybody.push sp.y
  end
  @time = @body[0].time
end
```

で、最初にコマンドログを読み書きしてから粒子を読んで、で、このコードの中で使う粒子を @body に、 YAML のデータを @ybody にいれます。出力は

```
def write
  @body.size.times{|i|
    @body[i].time = @time
    CP.new(@body[i], @ybody[i]).print_particle
  }
end
```

で、更新した @body と元の @ybody をあわせて出力です。まあ、mkplummer.cr の出力だと余計なものはないので、これしてもしなくても同じですが、、、

赤木 ：あと、これ、粒子動くところみたいわね。

学生 C：あ、、、これ、今、出力は、最初に CommandLog がでますが、あとはずーっと粒子がでるだけで、どこで次の時刻になるかとかわからないんですが、、、

赤木 ：あ、それは、PSDF の考え方としてはそれでいいの。粒子が ID と時刻もってるでしょ？だから、少なくとも概念としては、この出力は、各粒子について、4 次元時空の中での軌跡を与えてるから。

学生 C：うーん、概念としてなんか格好いいのはそうかもしれないですが、実際に絵にするのはどうすればいいんですか？

赤木 ：そうね、、、メモリが一杯ある計算機なら、まず時間方向全部読み込んで、それから分割、でもいいんだけど、粒子数ちょっと増えると破綻するわね。出力が時間方向に 1000 枚あるとメモリが 1000 倍いるとかになるから。

だから、例えば、今読んでるのと違う (先の) 時刻になったら中断、とかでいいんじゃない？

学生 C：中断、ってどうすればいいんですか？

赤木 ：まあ、単にプログラムの制御構造の中で処理なら、例えば今の nacsplot.cr では

```
pp=Array(Particle).new
while (sp= CP(Particle).read_particle).y != nil
  pp.push sp.p
end
```

としているけど、気分として

```
粒子読む
while まだ粒子がある (ここでは読まない)
  粒子配列を粒子 1 つで新しく作る
  while まだ粒子がある && 読んだ粒子の時刻が他と同じ
     粒子配列に追加
  end
  絵を書く
  粒子配列を捨てる
end
```

みたいな感じかな。関数とかクロージャとか使って格好良く書いてもいいんだけど、まあここでは特にそんなことするほどでもないから。

学生 C：あ、これならなんとか。

```
  1:require "grlib"
  2:require "clop"
  3:require "./nacsio.cr"
  4:include Math
  5:include GR
  6:include Nacsio
  7:
  8:optionstr= <<-END
  9:  Description: Plot program for multiple nacs snapshot
 10:  Long description: Plot program for multiple nacs snapshot
 11:
 12:  Short name:-w
 13:  Long name:  --window-size
 14:  Value type:  float
 15:  Variable name:wsize
 16:  Default value:1.5
 17:  Description:Window size for plotting
 18:  Long description:
 19:    Window size for plotting orbit. Window is [-wsize, wsize] for both of
 20:    x and y coordinates
 21:END
 22:
 23:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
 24:options=CLOP.new(optionstr,ARGV)
 25:update_commandlog
 26:ENV["GKS_DOUBLE_BUF"]= "true"
 27:
 28:wsize=options.wsize
 29:setwindow(-wsize, wsize,-wsize, wsize)
 30:setcharheight(0.05)
 31:setmarkertype(4)
 32:setmarkersize(1)
 33:sp= CP(Particle).read_particle
 34:while sp.y != nil
 35:  pp=[sp.p]
```

```
 36:  time = pp[0].time
 37:  pp! time
 38:  while (sp= CP(Particle).read_particle).y != nil && sp.p.time == time
 39:    pp.push sp.p
 40:  end
 41:  clearws()
 42:  box
 43:  mathtex(0.5, 0.06, "x")
 44:  mathtex(0.06, 0.5, "y")
 45:  text(0.6,0.91,"t="+sprintf("%.3f",pp[0].time))
 46:  polymarker(pp.map{|p| p.pos[0]}, pp.map{|p| p.pos[1]})
 47:  updatews()
 48:end
 49:c=STDERR.gets
```

こんなのですかね？

赤木 ：まあ動いたらこれでいいんじゃない？

学生 C : 動くのは動きました。ただ、1 万粒子くらいで結構遅いです。YAML って格好いいですが、やっぱり遅くないですか？

赤木 ：そこはやっぱり問題なのよね、、、浮動小数点形式をバイナリフォーマットじゃなくて人間が読める形で、というだけで結構遅いから。

まあ、実は、大きな計算の時には、どうせそんなにディスク容量がないから逆に問題なかったりするんだけど。読み書きのとこを並列化すれば結構速くなるし。

学生 C : そんなものですか？

赤木 ：わりと。

14.1 ツリーコード解説

赤木 ：一応ここでツリーコードってどういうものかをコードみながら解説お願い。

学生 C : え、私よくわかってないですが、、、

赤木 ：まあ、わかってない人が読んでくほうが読者のためになるかもしれないし。

学生 C : えー、そうですかね？まあやりますが。

上のプログラムで、4 行目までは色々 require とか include、110 行目までははコマンドラインオプション関係ですね。そこまで飛ばします。

112 行目からの struct AccPot は、加速度とポテンシャルをまとめて、加算を定義したクラスです。これなしで、加速度やポテンシャルをタプルでまとめて返り値にしてもいいのですが、加算があるほうがみやすいコードになるのでこれ定義してます。使ってるのは 380 行目です。

122 行目からは Body クラスですね。これ Particle でないのは元々の Joshua Barnes のプログラムから名前がきてるからですね。124 行目からは I/O 用の YAML.mapping でいつもの通りです。

134 行目 からの get_other_acc_and_pot は、自分から相手への重力とそのポテンシャルです。実際の計算に使うものですね。

146 行目の get_node_acc_and_pot は get_other_acc_and_pot を呼ぶだけですが、引数が違います。これは、ツリーからの重力を計算する関数と同じ名前にしているものです。その下にある loadtree は、実際には使わないはずですが形式的にコンパイラが文句をいわないように作ってあるものです。

center_of_mass は粒子の重心なので位置そのものとなります。

赤木 ：まだこの辺ツリーアルゴリズム本体じゃない感じ？

学生 C : そうですね。 get_node_acc_and_pot は最終的には使われますが。158 行目からの NBody にはいります。こっちもツリーアルゴリズムというよりは、N 体時間積分のプログラム全体、というところです。

161 行目からは initialize で、色々な内部変数に値をいれます。

173 行目からの evolve が時間積分プログラム本体です。184 行目の write_diagnostics はエネルギー保存とか書く関数です。189 行目からが時間積分ループ本体で、基本的には leapfrog 関数を呼ぶだけで、あとは write_diagnostics とか 粒子を書く write を指定された時間毎に呼んでます。

203 行目からが leapfrog ですね。これは Ruby バージョンからそのままなので、今まで作った関数渡すとかのではなくて素朴に書いてあります。get_tree_acc がツリー法で相互作用計算するアルゴリズムの実体です。

215 行目からが get_tree_acc で、これは maketree でツリー構造を作り、rootnode.center_of_mass でツリーの各ノードの重心を再帰的に計算します。それから、218 行目からの @body.each で全粒子へのツリーからの重力を計算します。

あと細かい関数色々ありますが、ここで見ておく必要があるのは 283 行目からの maketree です。

まず、284 行目の makerootnode ですが、これはもどって 275 行目からです。全粒子の座標絶対値の最大値をとって、それがはいるとようツリーのトップレベルの箱の大きさを決めてます。

で、そのあとは 286 行目からの @body.each で、ツリーに粒子を 1 個づつ挿入するアルゴリズムを使ってます。

あとは、なので、その下の Node クラスの loadtree、center_of_mass、get_node_acc_and_pot ですね。

loadtree は再帰的です。これは、8 個の子セルのどこにいくべきかを決めて、そこでが空なら単に子セルの代わりに粒子にするして、それでおしまいです。

で、既に粒子だったら、新しいセルを作って、今までそこにあった粒子を loadtree でいれてそこがセルなら既に少なくとも 1 つ粒子が入ってるいるセルなので、loadtree 自身を呼び出します。

ここまでくると、必ず子セルが Node になっているので、子セルに対して loadtree を呼びます。

赤木 ：これ短くて格好いいけどトリッキーよね、、、

学生 C : まあこれ、元々の Joshua Barnes のプログラムは C で、ポインタが粒子さしたりノードさしたりするんですよね、、、Crystal だとその辺自分が何かというのは is_a? とかでわかるので、わりとまだ、、、

赤木 ：そうねえ、、、

学生 C : あともうちょっとなので、 center_of_mass は再帰的に重心を計算するだけです。 sum とか reduce でもうちょっと綺麗に書ける気がしますが、、、

赤木 ：まあ動いてればね。

学生 C : 376 行目からが get_node_acc_and_pot です。これは、まず、距離と自分のサイズを比べて、サイズが大きいなら子セルからの力の合計、そうでなければ自分の重心からの力、となります。後は 389 行目以降のメイン部分で、粒子を nb.read de「「読んで、evolve を呼んで終わりです。

赤木 ：なんかかけあしだけど、ないよりはいいかしらね。

14.2 課題

1. エネルギーの他、重心の位置、速度、系の全角運動量について、どの程度保存しているかを確認し、それが適切な程度かどうかを検討せよ。
2. 粒子数、時間刻み、ソフトニング、見込み角を系統的に変化させて、エネルギー保存がどうなるかを検討せよ、エネルギーの指標には、t=0 から 1 まで積分して、その間の最大値を用いよ。

14.3 まとめ

1. Barnes-Hut ツリーアルゴリズムの単純な実装を作った。

14.4 参考資料

A hierarchical O(N log N) force-calculation algorithm[1]

[1] `https://www.nature.com/articles/324446a0`

Chapter 15

柔軟な表示プログラム

赤木 ：というわけでさくさく時間積分できるようになったから、色々なものをプロットできるプログラムとか作れないか、という話ね。

学生 C : x, y の代わりに x, z とか、速度空間とか、そういう話ですね？例えば -x pos[0] -y pos[1] みたいなこととかですよね？

赤木 ：そうなんだけど、もうちょっと色々、例えば-x 'pos[0]*vel[1]-pos[1]*vel[0]' で角運動量とか、そういうのができると便利じゃない？

学生 C : 便利なのはわかりますが、どうやるんですか？ Ruby とか Python ならもちろん、インタプリタだからこの文字列を実行時に評価、でいいわけですが、Crystal だとそうはいかないですよね？

赤木 ：そうね。なので、考え方が多分 3 通りくらいあるの。

1. 自前で簡単なインタプリタをもつ
2. C の関数を生成してリンクする
3. Crystal の関数を生成して全体をコンパイルしなおして実行する

学生 C : どれがいいんでしょうか？

赤木 ：ちょっと迷ってるから色々書いてみたんだけど、、、1 はもちろん安全で、あんまり難しくないんだけど、遅いのね。2 は速度はいいんだけど、どれくらい色々なことができるかはちょっとやってみないと。3 は、毎回 Crystal のプログラム全体をコンパイルしなおしになるからさすがちょっと、、、

学生 C : そういわれると 2 しかなさそうですが、、、

赤木 ：あと、これ、プロットするプログラムが知らないデータ、つまり、pos とか vel ではなくて、突然 密度とか温度とかストレスとか指定してもプロットできて欲しいじゃない？

学生 C : でも、そうすると、型は与えるわけですね？

赤木 ：そうね、必要ね。まあでも、浮動小数点数だけでもいいかも。

学生 C : そうすると、C でコードだしても、Crystal の側では結局 YAML の、まだ構造体になってないデータから値を取り出すか、あるいはコマンドラインオプションで与えた型情報から粒子クラスを作ってコンパイルするかですよね、、、なんかインタプリタのほうがまだ簡単？

赤木 ：まあ勉強も兼ねていわゆる LL(1) パーザ自分で書くのはどうかしら？そんなに難しいことないはず。

学生 C : なんかこう車輪の再発明感がありますが、、、、

赤木 ：まあでも一度やってみないと、文法とか構文解析とか意味がわからないから。

どういうことをしたいかというと、

```
r[0]*v[1]-r[1]*vl[0]
```

とか

```
sqrt(pos*pos)
```

とかいった表現から、Particle を読んでできてくるハッシュ構造からこれを計算するためのデータ構造にしたいわけね。BNFっぽい文法で適当に書くと例えばこんな感じ。

```
expression -> [sign] term { add_op term}
sign -> +|-
add_op -> +|-
term -> factor {mul_op factor}
mul_op -> *|/
factor -> variable|element|constant|function| (expression)
function -> function_name(parameter {,parameter})
parameter -> expression
```

これ、 a+b+c みたいなのが expression で、

```
expression -> [sign] term { add_op term}
```

は、最初の sign はあってもなくてもよくて、そのあとに term がきて、そのあとに add_op term が 0 回以上繰り返す、という感じね。で、 sign も add_op も + か - であると。term も同じようにするけど今度は * か / で、factor というものになると。で、factor は、ここでは variable, element, constant, function と、あと expression に括弧つけたもの、variable, element, constant はここで定義してないけど、ここでは 1.5*m*(r[0]*v[1]-r[1]*v[0]) みたいなのがあるとすると、m とか r[0] とか 1.5 とかを正規表現のパターンマッチで読むことにしましょう。

学生 C：すみません、全然わかりますません。

赤木 ：まあもうちょっとまって。これで、何がほしいかというと、1.5*m*(r[0]*v[1]-r[1]*v[0]) が

```
*--1.5
|--*--m
   |--"-"--*--r[0]
        |   |--v[1]
        |-*--r[1]
          |--v[0]
```

みたいなツリー構造になって欲しいわけね。このツリー構造だと、r[0] とかが実数値になれば、あとは演算子をみて計算していけるでしょ？

学生 C：これになればいい、というのはわかりますが、どうすればこれになるのかは、、、

赤木 ：まあその辺がコンパイラの理論のわりと本質だしね。1.5*m*(r[0]*v[1]-r[1]*v[0]) が 、[”1.5”, ”*”, ”m”, ”*”, ”(”, ”r[0]”, ”*”, ”v[1]”, ”-”, ”r[1]”, ”*”, ”v[0]”, ”)”] と配列になってたとしましょう。これは正規表現のパターンマッチだけでできるから。そうすると、これを左から順番に読むのが LL(1) 構文解析なのね。最初は 1.5 ね。

あ、正規表現って知っている?

学生 C : なんか // の中に謎な文字列を書いてなんかするものというくらいですが、、、

赤木 :そう。=∼っていう比較演算子が文字列と正規表現の間にあって、例えば /aaa/ =∼ ”aaabbbccc” だと、単純に aaa が aaabbbccc の中にあれば場所を返すし、/[a-z]/ =∼ ”a+b” だと a-z の間の文字が 1 つ最初にでてくる、この場合には a なので場所 0、/[a-z]+/ だと a-z の間の文字が 1 つ以上の繰り返し、みたいなのね。あと/([a-z]+)/ に括弧でくくると、そのパターンにあたる文字列が$1 みたいな特別な変数に入るの。これは Perl 由来かしらね。あと、この例でわかるように「[」とか「+」は正規表現の中で解釈されるから、文字としてその辺使うには「\[」みたいにするの。あとは [a-z0-9] みたいにしてアルファベット小文字と数字とかにできる、+ の代わりに * だと「0 回」も対応、?だと 0 か 1 回、とかくらいかな？あ、あと、「^」が文字列の先頭、これ今回使うから、と「$」が文字列の最後。まだ無限に沢山文法や機能があるけどこれくらいで今回使うものには十分かな。

で、最初は一番上の expression かな？というとこから解析を始めるわけ。なので、 expression という関数があって、これが、次の term を呼ぶ、だから

```
def expression(token)
  if token[0].type == sign
    n = Node.new(operator= token[0], lhs=nil)
    token.shift
    n.rhs = expression(token)
  else
    n = term(token)
    while token[0].type == sign
       token.shift
       n=Node.new(token[0], n, term(token))
    end
  end
  n
end
```

でいいのかしら？なんかこんな感じで動かないかなと。

学生 C : あんまりわかってないですが、ちょっと書いてみます、、、

(数日後)

なんかできたかな、、、

```
tokens= scan_expression_string( "1.5*m*(r[0]*v[1]-r[1]*v[0])")
x= expression(tokens)
x.ppp
```

と書くと

```
gravity> crystal parser0.cr
        1.5(Constant)
    *
        m(Variable)
*
            r[0](Element)
        *
            v[1](Element)
```

```
      -
              r[1](Element)
          *
              v[0](Element)
      a(Variable)
  Apply
              b(Variable)
          ,
              c(Variable)
      ,
          d(Variable)
  x # => #<NacsParser::Node:0x7fecd89f1c60
   @lhs=#<NacsParser::Token:0x7fecd89eac80 @s="a", @t=Variable>,
   @operator=#<NacsParser::Token:0x7fecd89eab60 @s="Apply", @t=Fapply>,
   @rhs=
    #<NacsParser::Node:0x7fecd89f1c90
     @lhs=
      #<NacsParser::Node:0x7fecd89f1cc0
       @lhs=#<NacsParser::Token:0x7fecd89eac20 @s="b", @t=Variable>,
       @operator=#<NacsParser::Token:0x7fecd89eac00 @s=",", @t=Comma>,
       @rhs=#<NacsParser::Token:0x7fecd89eac60 @s="c", @t=Variable>>,
     @operator=#<NacsParser::Token:0x7fecd89eabe0 @s=",", @t=Comma>,
     @rhs=#<NacsParser::Token:0x7fecd89eabc0 @s="d", @t=Variable>>>
  :481:7 in 'write_string'
    from /usr/share/crystal/src/string.cr:5022:5 in 'to_s'
    from /usr/share/crystal/src/io.cr:174:5 in '<<'
    from /usr/share/crystal/src/io.cr:188:5 in 'print'
    from /usr/share/crystal/src/kernel.cr:125:3 in 'print'
    from mkplummer.cr:172:3 in '__crystal_main'
    from /usr/share/crystal/src/crystal/main.cr:115:5 in 'main_user_code'
    from /usr/share/crystal/src/crystal/main.cr:101:7 in 'main'
    from /usr/share/crystal/src/crystal/main.cr:127:3 in 'main'
    from /lib/x86_64-linux-gnu/libc.so.6 in '__libc_start_main'
    from /home/makino/.cache/crystal/crystal-run-mkplummer.tmp in '_start'
    from ???
```

がでます。これはツリー構造を親ノードを左、子供を右の上下に書くコードで出力してます。

パーサーが、YACC の入力で与えられてるわけじゃなくてプログラムなので、本当にこれ？みたいなところがありますが、括弧ありで式に直すと

```
(1.5*m)*((r[0]*v[1])-(r[1]*v[0]))
```

で、大丈夫そうな気がします。

赤木 ：これ関数とかもあるの？もうちょっと色々テストケースとかも必要ね。まあでもとりあえず話を進めましょう。プログラムどんなの？

学生 C：現状ではテストというか簡単なドライバもつけて

```
1:module NacsParser
2:  extend self
```

```
 3:include Math
 4:  # Grammar
 5:  # expression -> [sign] term { add_op term}
 6:  # sign -> +|-
 7:  # add_op -> +|-
 8:  # term -> factor {mul_op factor}
 9:  # mul_op -> *|/
10:  # factor -> variable|element|constant|function| (expression)
11:  # function -> function_name(parameters)
12:  # parameters -> expression {,expression}
13:
14:  extend self
15:  enum Ttype
16:    Add
17:    Mul
18:    Variable
19:    Element
20:    Constant
21:    Open_Paren
22:    Close_Paren
23:    Comma
24:    Fapply
25:  end
26:
27:  class Token
28:    property s, t
29:    def initialize(s : String,t : Ttype)
30:      @s=s
31:      @t=t
32:    end
33:    def ppp(currentindent : Int64 = 0,  indent : Int64 = 4)
34:      print " "*currentindent, @s,"(#{@t.to_s})\n"
35:    end
36:  end
37:
38:  def scan_expression_string(s)
39:    original_s=s
40:    s= s.delete(" ")
41:    tokens=Array(Token).new
42:    while s.size>0
43:      len = 1
44:      if s[0] == '('
45:        tokens.push(Token.new("(", Ttype::Open_Paren))
46:      elsif s[0] == ')'
47:        tokens.push(Token.new(")", Ttype::Close_Paren))
48:      elsif  /^([1-9][0-9]*\.[0-9]*e[+-]?[0-9]+)/ =~ s
49:        tokens.push(Token.new($1, Ttype::Constant))
50:        len = $1.size
51:      elsif  /^([1-9][0-9]*\.[0-9]*)/  =~ s
52:        tokens.push(Token.new($1, Ttype::Constant))
53:        len = $1.size
54:      elsif  /^([1-9][0-9]*)/  =~ s
```

```
 55:       tokens.push(Token.new($1, Ttype::Constant))
 56:       len = $1.size
 57:     elsif /^([a-z_][a-z_0-9]*\[[0-9]+\])/ =~ s
 58:       tokens.push(Token.new($1, Ttype::Element))
 59:       len = $1.size
 60:     elsif /^([a-z_][a-z_0-9]*)/ =~ s
 61:       tokens.push(Token.new($1, Ttype::Variable))
 62:       len = $1.size
 63:     elsif /^([\+-])/ =~ s
 64:       tokens.push(Token.new($1, Ttype::Add))
 65:     elsif /^([\*\/])/ =~ s
 66:       tokens.push(Token.new($1, Ttype::Mul))
 67:     elsif s[0] == ','
 68:       tokens.push(Token.new(",", Ttype::Comma))
 69:     else
 70:       STDERR.print "Error unrecognizable expression\n"
 71:       pp! original_s
 72:       STDERR.print "Error at", s, "\n"
 73:       exit
 74:     end
 75:     s = s[len..(s.size-1)]
 76:   end
 77:   tokens
 78: end
 79:
 80: class Node
 81:   property :operator, :lhs, :rhs
 82:   def initialize(operator : Token, lhs : (Node|Token|Nil) ,
 83:                  rhs : (Node|Token|Nil) )
 84:     @operator=operator
 85:     @lhs = lhs
 86:     @rhs = rhs
 87:   end
 88:   def ppp(currentindent : Int64 = 0,  indent : Int64 = 4)
 89:     @lhs.ppp(currentindent+indent, indent) if @lhs
 90:     print " "*currentindent, @operator.s,"\n"
 91:     @rhs.ppp(currentindent+indent, indent) if @rhs
 92:   end
 93: end
 94:
 95:
 96: def expression(token)
 97:   signs = Array(Token).new
 98:   while token[0].t== Ttype::Add
 99:     signs.push  token[0]
100:     token.shift
101:   end
102:   n =  term(token)
103:   while signs.size >0
104:     op = signs.pop
105:     n = Node.new(op, nil, n)
106:   end
```

```
107:    while token.size >0 && token[0].t== Ttype::Add
108:      op= token.shift
109:      n=Node.new(op, n, term(token))
110:    end
111:    n
112:  end
113:
114:  def term(token)
115:    n = factor(token)
116:    while  token.size > 0 &&  token[0].t == Ttype::Mul
117:      op= token.shift
118:      n=Node.new(op, n, factor(token))
119:    end
120:    n
121:  end
122:
123:  def factor(token)
124:    if token[0].t== Ttype::Variable||token[0].t== Ttype::Element||
125:       token[0].t== Ttype::Constant
126:      n=token.shift
127:      if token.size >1 &&token[0].t == Ttype::Open_Paren
128:        token.shift
129:        n=Node.new(Token.new("Apply", Ttype::Fapply), n, parameters(token))
130:        if token[0].t == Ttype::Close_Paren
131:          token.shift
132:        else
133:          print "error"
134:          pp! n
135:          pp! token
136:        end
137:      end
138:    elsif token[0].t== Ttype::Open_Paren
139:      token.shift
140:      n=expression(token)
141:      if token[0].t == Ttype::Close_Paren
142:        token.shift
143:      else
144:        print "error"
145:        pp! n
146:        pp! token
147:      end
148:    end
149:    n
150:  end
151:
152:  def parameters(token)
153:    n = expression(token)
154:    while  token.size > 0 &&  token[0].t == Ttype::Comma
155:      op= token.shift
156:      n=Node.new(op, n, expression(token))
157:    end
158:    n
```

```
159:  end
160:
161:  def eval_expression(e : (Node|Token|Nil), t : YAML::Any)
162:    val=0.0
163:    if e.is_a?(Token)
164:      if e.t == Ttype::Constant
165:        val= e.s.to_f
166:      elsif e.t == Ttype::Variable
167:        val = t[e.s].as_f
168:      elsif  e.t == Ttype::Element
169:        /^([a-z_][a-z_0-9]*)\[([0-9]+)\]/ =~ e.s
170:        vname=$1
171:        index = ($2).to_i
172:        val = t[vname][index].as_f
173:      end
174:    elsif e.is_a?(Node)
175:      lhval=0.0
176:      rhval=0.0
177:      if e.operator.s== "Apply"
178:        val = eval_function(e.lhs, e.rhs, t)
179:      else
180:        lhval = eval_expression(e.lhs, t) unless e.lhs.is_a?(Nil)
181:        rhval = eval_expression(e.rhs, t) unless e.rhs.is_a?(Nil)
182:        if e.operator.s== "+"
183:          val = lhval+rhval
184:        elsif e.operator.s== "-"
185:          val = lhval-rhval
186:        elsif e.operator.s== "*"
187:          val = lhval*rhval
188:        elsif e.operator.s== "/"
189:          val = lhval/rhval
190:        end
191:      end
192:    end
193:    val
194:  end
195:
196:  def eval_function(f : (Node|Token|Nil),
197:                args : (Node|Token|Nil),
198:                t : YAML::Any)
199:    val=0.0
200:    if f.is_a?(Token)
201:      fname= f.s
202:      vals=Array(Float64).new
203:      while args.is_a?(Node) && args.operator.s == "Apply"
204:        vals.unshift eval_expression(args.rhs, t)
205:        args = args.lhs
206:      end
207:      vals.unshift eval_expression(args, t)
208:      if fname == "sqrt"
209:        val = sqrt(vals[0])
210:      elsif fname == "exp"
```

```
211:        val = exp(vals[0])
212:      elsif fname == "log"
213:        val = log(vals[0])
214:      elsif fname == "exp"
215:        val = exp(vals[0])
216:      elsif fname == "pow"
217:        val = vals[0]**vals[1]
218:      end
219:    end
220:    val
221:  end
222:end
223:
224:struct  Nil
225:  def ppp(currentindent : Int64 = 0,  indent : Int64 = 4)
226:  end
227:end
228:
229:include NacsParser
230:
231:tokens= scan_expression_string( "1.5*m*(r[0]*v[1]-r[1]*v[0])")
232:
233:x= expression(tokens)
234:x.ppp
235:
236:
237:
238:token =
239:
240:x= expression(scan_expression_string "a(b,c,d)")
241:
242:x.ppp
243:
244:pp! x
245:exit
246:
247:
248:x= expression(scan_expression_string( "x(y,z)+f(xy)+r[0]*v[1]"))
249:x.ppp
250:
251:x= expression(scan_expression_string( "x(y,z)+f(xy)+r[0]*v[a]"))
252:x.ppp
253:
254:x= expression(scan_expression_string( "x(y,z)+f(xy)+r[0]*v[]"))
255:x.ppp
```

です。中身ですが、NacsParser というモジュールにして、まず Ttype ですね。これは enum というやつで、C 言語にもありますが、0, 1, 2 ... に人にわかりやすい名前つける、というだけのものです。Add が 0、Mul が 1 みたいな。で、これはトークン、つまり、a + b + c を ”a”, ”+”, ”b”, ”+”, ”c” みたいにバラバラにしたあとでそれぞれがどういう種類か、というものです。

その次は class Token で、トークンに対応する文字列と上の種類をもつクラスを作ってます。あと、

表示用の ppp という関数で、文字列と種類を表示します。enum を to_s でその名前の文字列にできるのは地味に便利ですね。この辺は Ruby はなくて Python はあるというか 3.4 からだから割合最近？

で、その次の scan_expression_string が、文字列をトークンの配列に変換する関数です。Lex とかのツール使うのが本当かもしれませんが、C のプログラムでてきても面倒だし、Crystal 用のはないっぽいので正規表現とか文字の比較で作りました。

まず、今回の文法ではスペースに意味がないので全部とります。なので、

```
f ( x y )
```

は

```
f(xy)
```

と同じみたいなだいぶアレな書き方もできます。

赤木 ：昔の FORTRAN みたいね。

学生 C：あ、そうだったんでしたっけ？

赤木 ：そう。昔の FORTRAN だと

```
      E N D
```

とか

```
      R E T U R N
```

とか書いてあるコードがたまにあるわ。

学生 C：できるからってしたらいいというものでもない気がしますが、、、

赤木 ：まあね。でもそうすると、先頭の空白だけとるほうがよくない？

学生 C：まあそうかもしれません。とりあえずこのまま説明すると、あと

```
      if s[0] == '('
        tokens.push(Token.new("(", Ttype::Open_Paren))
```

みたいなのが延々続きます。「()+-*/,」はこの文字が先頭にあったらそのトークンと決まってるので、それだけです。あ、「+-」は正規表現ですね。

```
      elsif /^([\+-])/ =~ s
```

// の中が正規表現、 =~ が正規表現が文字列にマッチするかどうかの演算子、「^」は文字列の先頭、[xy] は文字 x と文字 y のどっちかですが、 + は特別な意味があるので \+ とします。 - はしなくていいみたいです。「*/」も同様で、これは両方とも \ でエスケープです。あと、 $1 は、正規表現の中で括弧 () でくくった部分にマッチした文字列、というやつです。括弧が複数あれば順番に $1, $2 となるんだと思います。

赤木 ：

```
      elsif  /^([1-9][0-9]*\.[0-9]*e[+-]?[0-9]+)/ =~ s
```

は何？

学生 C : これは、[1-9] で 1 から 9 まで、次は [0-9]* ですが、ここでの * は前の文字が 0 回以上、次の \. は文字としての「.]、そのあとまた [0-9] が 0 回以上あって、「e] があって、[+-]? は + か - が、0 または 1 回、あとまた数字が、今度は + なので 1 回以上ですね。要するに

```
1.5e+10
```

みたいな、指数つきの浮動小数点数です。

赤木 ：指数なしの 1.5 とか、小数点がない 10 とか、小数点がないけど指数はあるとかは？

学生 C : まだその辺全部は作ってないです。指数なしの 1.5 は

```
elsif  /^([1-9][0-9]*\.[0-9]*)/  =~ s
```

かな。あと、今回の文法では a[1] みたいなのはもうトークンとして、という話だったので、

```
elsif /^([a-z_][a-z_0-9]*\[[0-9]+\])/ =~ s
```

で、文字または 「_」で始まって、文字または「_」または数字がきて、そのあと「[」+数字 (1 個以上)+「]」を Element として、「[]」がないのを Variable としてます。

赤木 ：定数の表現はまだちゃんとできてない、ということね。

学生 C : はい。とりあえずなんか動いて欲しかったので。

赤木 ：これの実行結果は？

学生 C : 例えば

```
print tokens.map{|t| "[#{t.s},#{t.t}]"}.join(", "),"\n"
```

で書くと

```
[1.5,Constant], [*,Mul], [m,Variable], [*,Mul], [(,Open_Paren],
[r[0],Element], [*,Mul], [v[1],Element], [-,Add], [r[1],Element],
[*,Mul], [v[0],Element], [),Close_Paren]
```

ですね (適当に改行いれました)

赤木 ：いい感じね。文法本体は？

学生 C : LL(1) パーサを、ということだったので再帰下降パーサを低レイヤを知りたい人のための *C* コンパイラ作成入門[1] の再帰下降構文解析のとこを参考にしながら作りました。

まず class Node ですが、operator, lhs, rhs、つまり演算子、左側、右側をもつとします。左側、右側はそれぞれ Node か Token か nil としてます。あと、印刷する関数を ppp で、再帰的に ppp を呼ぶ形で作りました。

赤木 ：もうちょっと格好よくグラフの表示とかにして欲しい気もするけど、テキストだけですむのはそれはそれで便利よね。パーサ本体は？

学生 C : まず expression ですね。

[1] https://www.sigbus.info/compilerbook#%E5%86%8D%E5%B8%B0%E4%B8%8B%E9%99%8D%E6%A7%8B%E6%96%87%E8%A7%A3%E6%9E%90

```
  def expression(token)
    signs = Array(Token).new
    while token[0].t== Ttype::Add
      signs.push  token[0]
      token.shift
    end
    n =  term(token)
    while signs.size >0
      op = signs.pop
      n = Node.new(op, nil, n)
    end
    while token.size >0 && token[0].t== Ttype::Add
      op= token.shift
      n=Node.new(op, n, term(token))
    end
    n
  end
```

LL(1) 文法に対する再帰下降パーサーって、基本的には、左側が一段おりたもの、この場合には最初の

```
      n =  term(token)
```

なんですが、この文法では expression は 符号+ expression というのがあるので最初のトークンが符号にあたる Ttype::Add だったら符号憶えておきます。複数符号あったら一応複数憶えておきます。これ、赤木さんが最初に書いたやつ全然間違っててちょっと苦労しました。

赤木 ：あら、そう？ごめんなさいね、、、

学生 C：赤木さんのは再帰みたいにしてたんですが、それだと最初の符号がそのあとの式全体にかかるので式の意味が変わっちゃってました。今のコードはこれで多分正しいと思いますが、最初にでてきた term に、符号があれば

```
    while signs.size >0
```

のループで符号つけて新しいノードにしてます。

で、最後に、

```
    while token.size >0 && token[0].t== Ttype::Add
```

のループで、次の term をその前にある + か - で追加してます。これで、沢山加算とか減算しても、ちゃんと正しく符号が考慮されます。

で、次が term で、これは符号がなくて演算子が */ だという以外は expression と同じで、但し term ではなくて factor を呼び出すことになります。その次が factor で、これは

```
  factor -> variable|element|constant|function| (expression)
```

と、色々な可能性があるので少し複雑です。まず、最初のトークンが Variable, Element, Constant のどれかの時ですが、まだ関数かもしれないわけです。関数かどうかは次が括弧かどうかでみるとので、これが 118 行目です。この時はさらに parameters を呼び出します。parameters m の呼び出しから帰ってくると次のトークンは「)」のはずなので、それをチェックします。

それ以外は、本当に Variable か Constant なので、そのトークン自体が factor の返り値になります。

そうでなければ最初が「(」のはずで、この時には expression のはずなので、ということですね。ここで、 1*(2+3) みたいなのが処理できるわけです。

parameters は「,」を演算子として引数は expression であるとして並べていく、という形で、これで引数が複数ある関数も書けます。

あと、このモジュールの外側ですが、 ppp が Nil に対して定義されてないと文句いわれたので定義してます。

赤木 ：まあ答がもっともらしいからとりあえず使えそうね。

学生 C：再帰下降パーサって、全部できて、いわゆる終端記号、要するにトークンですね、そこまで辿りつかないと何をするのか全然分からないのがなんか気持ち悪いというか、なんでこれで動くのかなという気がします。

動作をたどっていくと確かに動くんですが、、例えば f(x) みたいなのだと最初のトークンは「f」で、expression → term → factor ときて、ここで Variable なので、となって、次のトークンは「(」だからここではツリー構造としては Type::Fapply を生成して、「(」まで読んでから parameters を呼ぶ、そうするとまた expression を呼んで、ずーっとまた factor まできて、「x」なのでまた Variable で今度は次は「)」だからここで factor が「x」を返す、で、term に戻ってきて、次のトークンは「)」で */ じゃないので term も expression に戻って、次のトークンは「)」で */ じゃないので expression は parameters に戻って、ここでも次のトークンは「)」で「,」じゃないので factor にもどって、ここでちゃんと次は「)」で、これでちゃんと上手くできてもうトークンないので expression まで戻って抜ける、と確かにこうなるんですが、なんかすごい複雑ですよね。

```
expression
  term
     factor
        parameters
   expression
     term
        factor
```

みたいに、これだけ関数呼ばれるわけで、、、

赤木 ：そうねえ。これ、演算子の優先順位とあと括弧を表現するのに、expression, term, factor という 3 種類の規則を作って、それに対応して関数 3 個あるから、この 3 つの関数は必ずセットで呼ばれる、1 つのものみたいなところはあるわね。

でも、そうできた分、3 つの関数のそれぞれは簡単になってるわけ。

ここまでできたら、後は粒子データとこの構文木から値を計算するだけね。

学生 C：だけっていうのは簡単ですけどプログラム書くのは、、、まあやってみます。

(数時間後)

あれ、なんか簡単でした。YAML::Any と Node 渡して評価する関数が

```
def eval_expression(e : (Node|Token|Nil), t : YAML::Any)
  val=0.0
  if e.is_a?(Token)
    if e.t == Ttype::Constant
      val= e.s.to_f
    elsif e.t == Ttype::Variable
      val = t[e.s].as_f
    elsif  e.t == Ttype::Element
```

```
      /^([a-z_][a-z_0-9]*)\[([0-9]+)\]/ =~ e.s
      vname=$1
      index = ($2).to_i
      val = t[vname][index].as_f
    end
  elsif e.is_a?(Node)
    lhval=0.0
    rhval=0.0
    lhval = eval_expression(e.lhs, t) unless e.lhs.is_a?(Nil)
    rhval = eval_expression(e.rhs, t) unless e.rhs.is_a?(Nil)
    if e.operator.s== "+"
      val = lhval+rhval
    elsif e.operator.s== "-"
      val = lhval-rhval
    elsif e.operator.s== "*"
      val = lhval*rhval
    elsif e.operator.s== "/"
      val = lhval/rhval
    end
  end
  val
end
```

これだけです。これは再帰的に自分を呼び出すんですが、引数の e が Token だったら粒子データを見にいきます。といっても Constant だったら浮動小数点文字列のはずなので to_f するだけ、 Variable なら、 YAML::Any はハッシュなので、変数名で検索して返ってきた値を as_f で浮動小数点数にします。Element なら、変数名と添字の値を取り出してから、同様にハッシュから値を取り出します。

e がノードなら、演算子なわけで、あ、関数まだ書いてないので、左側と右側のノードを評価してから、演算子に応じて計算するだけです。

これ使って絵書く関数は

```
  1:require "grlib"
  2:require "clop"
  3:require "./nacsio.cr"
  4:require "./parser.cr"
  5:include Math
  6:include GR
  7:include Nacsio
  8:
  9:optionstr= <<-END
 10:  Description: Plot program for multiple nacs snapshot
 11:  Long description: Plot program for multiple nacs snapshot
 12:
 13:  Short name:-w
 14:  Long name:  --window-size
 15:  Value type:  float
 16:  Variable name:wsize
 17:  Default value:1.5
 18:  Description:Window size for plotting
 19:  Long description:
 20:    Window size for plotting orbit. Window is [-wsize, wsize] for both of
```

```
21:    x and y coordinates
22:
23:  Short name:-x
24:  Long name:  --x_expression
25:  Value type:  string
26:  Variable name:xexp
27:  Default value:r[0]
28:  Description:Expression to use as x coordinate
29:  Long description:     Expression to use as x coordinate
30:
31:  Short name:-y
32:  Long name:  --y_expression
33:  Value type:  string
34:  Variable name:yexp
35:  Default value:r[1]
36:  Description:Expression to use as y coordinate
37:  Long description:     Expression to use as y coordinate
38:
39:END
40:
41:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
42:options=CLOP.new(optionstr,ARGV)
43:include NacsParser
44:
45:pp! options
46:
47:xexp = expression(scan_expression_string(options.xexp))
48:yexp = expression(scan_expression_string(options.yexp))
49:
50:update_commandlog
51:ENV["GKS_DOUBLE_BUF"]= "true"
52:
53:wsize=options.wsize
54:setwindow(-wsize, wsize,-wsize, wsize)
55:setcharheight(0.05)
56:setmarkertype(4)
57:setmarkersize(1)
58:sp= CP(Particle).read_particle
59:while sp.y != nil
60:  pp=[sp.y]
61:  time = sp.p.time
62:  pp! time
63:  while (sp= CP(Particle).read_particle).y != nil && sp.p.time == time
64:    pp.push sp.y
65:  end
66:  clearws()
67:  box
68:  mathtex(0.5, 0.06, "x")
69:  mathtex(0.06, 0.5, "y")
70:  text(0.6,0.91,"t="+sprintf("%.3f", time))
71:  polymarker(pp.map{|p| eval_expression(xexp,p)},
72:             pp.map{|p| eval_expression(yexp,p)})
```

```
73:  updatews()
74:end
75:c=STDERR.gets
```

で、これは前の単に x,y プロットするのとの違いをみたほうが早いですね。

```
gravity> diff -C0 nacsplot1.cr nacsplot2.cr
*** nacsplot1.cr2020-05-24 18:52:29.933399329 +0900
--- nacsplot2.cr2020-05-26 00:43:36.661914327 +0900
***************
*** 3 ****
--- 4 ----
+ require "./parser.cr"
***************
*** 20 ****
--- 22,38 ----
+
+   Short name:-x
+   Long name:  --x_expression
+   Value type:  string
+   Variable name:xexp
+   Default value:r[0]
+   Description:Expression to use as x coordinate
+   Long description:     Expression to use as x coordinate
+
+   Short name:-y
+   Long name:  --y_expression
+   Value type:  string
+   Variable name:yexp
+   Default value:r[1]
+   Description:Expression to use as y coordinate
+   Long description:     Expression to use as y coordinate
+
***************
*** 24 ****
--- 43,49 ----
+ include NacsParser
+
+ pp! options
+
+ xexp = expression(scan_expression_string(options.xexp))
+ yexp = expression(scan_expression_string(options.yexp))
+
***************
*** 35,36 ****
!   pp=[sp.p]
!   time = pp[0].time
--- 60,61 ----
!   pp=[sp.y]
!   time = sp.p.time
***************
```

```
*** 39 ****
!     pp.push sp.p
--- 64 ----
!     pp.push sp.y
***************
*** 45,46 ****
!   text(0.6,0.91,"t="+sprintf("%.3f",pp[0].time))
!   polymarker(pp.map{|p| p.pos[0]}, pp.map{|p| p.pos[1]})
--- 70,72 ----
!   text(0.6,0.91,"t="+sprintf("%.3f", time))
!   polymarker(pp.map{|p| eval_expression(xexp,p)},
!              pp.map{|p| eval_expression(yexp,p)})
```

---

parser.cr を require して、オプションで式を与えるのを x, y の 2 個追加してます。それから、粒子クラスの sp.p じゃなくて、YAML のままの sp.y のほうを配列にいれます。で、最後、x 座標が p.pos[0] だったのが eval_expression(xexp,p)、y 座標も同様ですね。なんか思ったより簡単でした。

赤木 ：まあこれエラー処理とかしてないし、まだ関数ないし、速度もあんまり、みたいなところはあるわね。でも、よくできました。大変結構。

15.1 課題

1. トークン生成のところに、整数と 1e20 のような小数点以下がなくて指数がある浮動小数点数を認識する規則を追加せよ。
2. 関数 (例えば sqrt と exp) を追加せよ。
3. (オプションではなくてソースプログラムの中で) このインタプリタで使える関数を追加する方法を考えてみよ。
4. 複数の引数をもつ関数も実装できるようにせよ。

15.2 まとめ

1. 再帰下降型構文解析により、粒子データの物理量から新しい量を計算する関数を作り、それを画面表示できるようにした。
2. 構文解析には、トークン化 (字句解析) と文法解析がある。今回は、トークン化には正規表現を、文法解析には再帰下降型構文解析を実装することでおこなった。
3. is_a? 関数である変数の型をチェックできる。チェックしてそれが正しいブロックの中では、その型を引数にする関数を使える。

15.3 参考資料

低レイヤを知りたい人のための *C* コンパイラ作成入門[2]

Crystal の *class Regex*[3]

[2] https://www.sigbus.info/compilerbook#%E5%86%8D%E5%B8%B0%E4%B8%8B%E9%99%8D%E6%A7%8B%E6%96%87%E8%A7%A3%E6%9E%90
[3] https://crystal-lang.org/api/latest/Regex.html

pcre2pattern man page[4] Crystal が使っている正規表現ライブラリの文法解説

[4] http://www.pcre.org/current/doc/html/pcre2pattern.html

Chapter 16

今後の計画

赤木 ：ここまでで、数値計算、特に粒子法の基本的な考え方と、あと、そのためのソフトウェアパッケージのベースにあたるものを一応やったと思うのね。

- 粒子データを扱うクラス
- nacsio その汎用の入出力
- clop 色々な処理プログラムを作るためのコマンドライン解釈ライブラリ
- parser 粒子データを処理するためのインタプリタ「

こんな感じ。これで、N 体だけで、銀河みたいな無衝突系なら 100 万粒子くらいまではちょっと時間かかるけどできなくはないわ。

学生 C：じゃあこの辺で私お疲れさまで、、、、

赤木 ：でも、まだ、もっと粒子数大きいのを、と思うとツリーコードだというだけではやっぱり性能不足で、

- 並列計算機での MPI による並列化
- 1 台の中でのマルチコアによる並列化
- SIMD 演算器の利用

を考えないといけないし、惑星形成とか球状星団とかの計算に使おうと思うとハミルトニアン分割をいれたコードじゃないと使えないのね。で、例えば並列化は FDPS っていう作者氏のグループで開発しているフレームワーク使う方法があるし、その上でハミルトニアン分割するのは GPLUM とか PeTar とかの、それぞれ素晴らしいプログラムがあるのね。

でも、それ使って、っていうだけだとなんというか教育にならないじゃない？

学生 C：でもだから車輪の再発明もどうかと思います。

赤木 ：なので、その辺難しいところよね。あと、単に重力だけでなくて流体もいれたいなと。

学生 C：それって作者がやってみたいだけなんでは？

赤木 ：まあそうともいうわね。そういう意味ではやっておきたい/みたいこと (誰がよというのはちょっとおいて) を並べると

1. FDPS による並列化・高速化

2. FDPS によらない並列化・高速化
3. ハミルトニアン分割 (P3T スキーム) のコード
4. 球状星団ができるコード
5. 惑星形成が一応動くコード
6. 銀河中心 (ブラックホールあり) が動くコード
7. 流体いりコード

みたいな。

学生 C : それちょっと無理そうというか、FDPS 使わないで並列化はさすがにちょっとなくないですか？FDPS 的なものを全部書き直すんですか？

赤木 ：うーん、いつかというかそのうちというか時々しないといけないと思うんだけどねえ、、、

学生 C : まあでも今じゃないと思います。なのでまずは FDPS 使うのをでいいんじゃないですか？で、P3T は書いてみて的な。

赤木 ：そうねえ、当面の計画としてはそっちね。

学生 C : FDPS の Crystal 対応って作者がやりかけてなかったでしたっけ？えーと、*crystalfdps*[1] ですね。

赤木 ：あ、でも、これ、オフィシャルな C 言語インターフェースができる前に作りかけたやつで、Fortran インターフェース用のダミーコードを生成するのよね、、、まあそれベース直すのがいいかしら。

学生 C : と思います。

16.1 課題

この章は息抜きなので課題はありません。

16.2 まとめ

1. 次の章では FDPS を使ってみる。

16.3 参考資料

FDPS[2]

[1] https://github.com/jmakino/crystalfdps
[2] http://fdps.jmlab.jp/?lang=ja

Chapter 17

FDPS を Crystal から使う

赤木 ：というわけで FDPS を Crystal のプログラムから使う話ね。

学生 C : 一応読者のために FDPS とは何かってのをちょっと。

赤木 ： *FDPS*[1] は、作者のグループで開発・公開している、「大規模高性能粒子法プログラム開発のためのフレームワーク」というものね。どういう考え方かというと、

つまり、前章で書いたような

- 並列計算機での MPI による並列化
- 1 台の中でのマルチコアによる並列化
- SIMD 演算器の利用

をやってくれるものなの。

学生 C : MPI による並列化をやってくれるって、どういうことですか？あれだって自分がどの粒子をもってるかとか意識して、相互作用計算とかは自分のとこにない必要なデータはもらってきてとかしないといけないし、もらってくるっていっても実際にはデータもってるほうが送って、それをこちらが受け取るしで滅茶苦茶面倒くさいじゃないですか？

赤木 ：だから、そのへん全部 FDPS の関数がやってくれるわけ。ツリーコードで、相互作用計算ややこしいけど、関数としては

```
get_tree_acc
```

を呼んだらあと全部やってくれたじゃない？そういう感じ。

学生 C : でも、こちらで粒子クラスとか相互作用関数とか定義するわけでですよね、、、Crystal ならもちろん粒子クラスを渡されたところでコンパイラがなんかするからそれでいいですが、 C だとそんなのできないじゃないですか？

赤木 ：まあ C でもできなくはなくて、例えばこちらで定義した粒子クラスを読み込んでコンパイルするソースプログラムライブラリにすればいいんだけど、でもそうすると、2 種類の粒子を作るとかだとちゃんと違う名前の関数を生成しないといけないし、ちょっと面倒だしメンテナンスも大変かなあ。というわけで、 FDPS では C++ 使ってるの。C++ では、template という機能でクラスを渡すことができるの。 FDPS のサンプルプログラムの例だすわね。

[1] http://fdps.jmlab.jp/?lang=ja

```
template<class Tpsys>
void kick(Tpsys & system,
          const PS::F64 dt) {
    PS::S32 n = system.getNumberOfParticleLocal();
    for(PS::S32 i = 0; i < n; i++) {
        system[i].vel  += system[i].acc * dt;
    }
}
```

これ、なんだかよくわからないと思うけど、実際に使うとこは

```
PS::ParticleSystem<FPGrav> system_grav;
...
kick(system_grav, dt * 0.5);
```

こんな感じ。PS::ParticleSystem は FDPS のほうで用意した型なんだけど、これに FPGrav という、これはユーザー側で定義した粒子型なんだけど、それを渡して、FPGrav を知っている ParticleSystem 型、というのを作るわけ。で、そのインスタンスが system_grav であると。そうすると、 kick が呼ばれる行では system_grav はその型だ、とコンパイラが知っているから、

```
template<class Tpsys>
void kick(Tpsys & system,
```

のとこで、 system の型である Tpsys は PS::ParticleSystem¡FPGrav¿ だ、とコンパイラはわかって、なのでコンパイルできるのね。そのために、Tpsys はそういう、 kick 書いた時点では未定義で引数のほうの型を使う、ということをコンパイラに教えるのが

```
template<class Tpsys>
```

だということ。

学生 C：なんか無駄に複雑な感じがしますが、、、 Crystal だったら、、、多分

```
def kick(system, dt)
  system.getNumberOfParticleLocal.times{|i|
    system[i].vel  += system[i].acc * dt
  }
end
```

とか、 system がそもそも粒子の配列だったら

```
def kick(system, dt)
  system.each{|p| p.vel += p.acc * dt}
end
```

で、、、だいぶ短いし、本当に計算の中身を書いてるとこは結局 1 行で同じなんだから、それ以外のとこが簡単なほうがいいし、計算の中身も each とか使うと簡潔ですよね、、、

赤木 ：まあそれは、Crystal というかその文法の元になってる Ruby がそういうもので、そもそも型は実行時に決まることになっているから何も書かなくていいからね。 Crystal は文法は可能な限り Ruby だけどコンパイル時に全部型決めるので、コンパイラが実行をシミュレートするようなこ

として可能な限り型決めて、できなかったらエラーにする、となってるから。C++ は逆に、全部型宣言しないといけない C から出発したから、「型を渡すことができる」ということを書かないといけなくなっちゃうのね。これは多分、C 言語にあった、型が宣言されてない変数とか引数とか関数は全部 int 型、というのと矛盾しないようにかな？

まあでも、C とか C++ を使うしかなかったとこはあって、そもそも Crystal 言語って本当に新しくて、マルチコアの並列化がやっとちょっと動くけど、それ以前の問題として「京」コンピュータにははいらないから。富岳は Arm で LLVM ベースのコンパイラもあるから原理的にははいるんだけどね、、、FDPS の開発は 2013 年に始まってるから。C++ の機能も、新しいのはまだ「京」で動くコンパイラにはなかったりしたから、古い機能だけ使うとか色々苦労してるのよ。

学生 C : Crystal はまあ、我々がちょっと使うくらいはともかく、まだ言語の開発が途中だし、これからどうなるかも分からないですから、、、

赤木 ：そうなの。そういうわけで、「京」とか富岳とかのスパコンで、安心して使えるのはそもそも Fortran だけで、 C/C++ はまあ動くけどちゃんと性能出せるかどうかわからない、それ以外はそもそも動かない、みたいな感じだったから。「京」では Python でも苦労したみたい。

学生 C : 富岳というか FX700/FX1000 は大丈夫なんではないですか？

赤木 ：そうなるといいわねえ、、、まあ、ラズパイで Crstal コンパイラ動かして人もいるみたいだから Crystal も動くだけは動くわね。

話がそれたけどそういうわけで FDPS は C++ で書いてあって、使う側からみると、ちゃんとスパコンで動いて MPI での並列化とかもして、クラスとか渡せば並列化して面倒なことをやってくれる素敵ライブラリなんだけど、C++ で書いてあって C++ の機能使うから、ユーザープログラムも C++ で書かないといけなかったのね。

学生 C : クラスとか渡すんだからそうなるしかないんじゃないですか？

赤木 ：でもね、メモリレイアウトとしては C++ のクラスは C の構造体と互換性維持してるの。つまり、C++ の言語側で使うタグ「とか一切もってないのよ。あと、もうひとつ大事なのは、C++ のコードの側に C で作った、というか C で作ってなくても Fortran でもあと Crystal でもいいんだけど、 C から呼べるように書いてある関数であれば、C++ の中からでも呼べるのね。

で、FDPS の考え方としては、

- 粒子クラス
- 粒子クラス (と、あと場合によっては多重極子クラス) を受け取って相互作用計算する関数

を渡せればいいから、粒子構造体を C 言語で通るように定義して、それの struct と書いてあるところを class に書換えてやって FDPS に渡してあと相互作用関数も渡すと FDPS の側ではアプリケーションプログラムが使うライブラリ関数を全部生成できるから、それを C 言語から呼べるラッパーを経由して呼べばいいわけ。

学生 C : すごく複雑な気がしますが、、、

赤木 ：まあ、それでも呼べるだけましよ。

学生 C : まあそうかもですが、、、

赤木 ：この辺はだって、ユーザー側から見れば C のライブラリ関数みたいに見えるから、それで普通に使えるのね。あと、これの開発を始めた元々の同期が Fortran 言語のインターフェースを作る、ということだったから、Fortran のインターフェースも生成できるし Fortran の構造体書くと自動的に C++ のクラスに変換する機能もあるのよ。

学生 C : Fortran なんて今誰か使ってるんですか？なんか古代の言語と思ってました。

赤木 ：あら、スパコンでは随分使われてるわよ。日本メーカーだとわりと最近というか「京」ができてしばらくたっても C++ のコンパイラの出来があんまりそのだったりしたし。

学生 C：えええ、、そうなんですか。Fortran にもクラスとかあるんですか？

赤木 ：結構昔、Fortran 90 の時に derived data type という名前で C の構造体にあたるものが導入されて、Fortran 2003 でそれが C++ のクラスと大体同じように使えるようになった感じ。ただし、テンプレートはないの。Fortran 202Y には入れるとかの議論はあるみたい。

学生 C：はあ、、、

赤木 ：まあ、雑談はこれくらいにして、FDPS を呼び出す話ね。github の古い奴をみると、

```
convert_crystal_struct_to_f90.rb
convert_f90_struct_to_crystal.rb
convert_f90_if_to_crystal.rb
```

と ruby のプログラムが 3 個あって、その辺が基本的な変換するみたいね。これを Fortran90 じゃなくて C インターフェースにして欲しいのね。

学生 C：はあ、、、まあやりますが。Ruby のままですか？ Crystal にします？

赤木 ： うーん、Crystal にしてもいいけどまあそのままでも。

学生 C：了解です。

FDPS の公開バージョン[2]で、*user_defined.h*[3]をみてと。まずこれがでればいいはず。Fortran だすコードは これ[4]と。

入力は、、、

```
  1:#================================
  2:#   MODULE: User defined types
  3:#================================
  4:module FDPS_vector
  5:lib FDPS
  6:   struct Full_particle #!fdps FP,EPI,EPJ,Force
  7:      #!fdps copyFromForce full_particle (pot,pot) (acc,acc)
  8:      #!fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps) (pos,pos)
  9:      #!fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
 10:      id : Int64  #$fdps id
 11:      mass : Float64     #$fdps charge
 12:      eps : Float64
 13:      pos : Cvec_Float64 #!fdps position
 14:      vel : Cvec_Float64 #!fdps velocity
 15:      pot : Float64
 16:      acc : Cvec_Float64
 17:    end
 18:end
 19:end
 20:
 21:#   !**** Interaction function (particle-particle)
 22:include Math
 23:def calc_gravity(ep_i,n_ip,ep_j,n_jp,f)
 24:  n_ip.times{|i|
 25:    pi = (ep_i + i).value
```

[2] https://github.com/FDPS/FDPS
[3] https://github.com/FDPS/FDPS/blob/master/sample/c/nbody/user_defined.h
[4] https://github.com/jmakino/crystalfdps/blob/master/convert_crystal_struct_to_f90.rb

```
26:    eps2 = pi.eps*pi.eps
27:    xi = Vec_Float64.new(pi.pos)
28:    ai =  Vec_Float64.new(0)
29:    poti = 0_f64
30:    n_jp.times{|j|
31:      pj = (ep_j + j).value
32:      xj = Vec_Float64.new(pj.pos)
33:      rij = xi - xj
34:      r2 = rij*rij+eps2
35:      rinv = 1_f64/sqrt(r2)
36:      mrinv = pj.mass*rinv
37:      mr3inv = mrinv*rinv*rinv
38:      ai -= rij*mr3inv
39:      poti = poti - mrinv
40:    }
41:    pfi = (f+i)
42:    pfi.value.pot =  pfi.value.pot + poti
43:    pfi.value.acc =   Vec_Float64.new(pfi.value.acc)+ ai
44:  }
45:end
```

と。なら Ruby のプログラムをちょっと修正して

```
 1:#
 2:# convert_crystal_struct_to_c.rb
 3:#
 4:def print_header
 5:  print <<-EOF
 6:#pragma once
 7:/* Standard headers */
 8:#include <math.h>
 9:/* FDPS headers */
10:#include "FDPS_c_if.h"
11:EOF
12:end
13:
14:$type_conversion_table =<<-EOF
15:Int64         long long
16:Float64       double
17:Cvec_Float64  fdps_f64vec
18:Cvec_Float32  fdps_f32vec
19:EOF
20:
21:
22:$type_conversion_hash = $type_conversion_table.split("\n").map{|s|
23:  a= s.split
24:  [a[0], a[1..(a.size-1)].join(" ")]}.to_h
25:
26:open("crmain.cpp","w"){|f|
27:       f.print <<EOF
28:/* Standard headers */
```

```
29:#include <iostream>
30:#include <fstream>
31:/* FDPS headers */
32:#include <particle_simulator.hpp>
33:/* User-defined headers */
34:#include "FDPS_Manipulators.h"
35:extern "C"{
36:void crystal_init(void);
37:void crmain(void);
38:
39:}
40:int main(int argc, char *argv[])
41:{
42:
43:   //* Initialize fdps_manip
44:   FDPS_Manipulators::Initialize(argc,argv);
45:   //* Call Fortran main subroutine
46:   //   f_main_();
47:   crystal_init();
48:   crmain();
49:
50:
51:   return 0;
52:
53:}
54:EOF
55:     }
56:def print_fortran_member(s)
57:  a = s.split
58:  ss = "     "
59:  if a[0][0] != "#"
60:    # member variable line
61:    varname = a[0]
62:    typename = $type_conversion_hash[a[2]]
63:    raise "unparsable line: #{s}"    if a[1] != ":" || typename == nil
64:    ss += " "+typename + " "+ varname+";"
65:    a = a[3..(a.length)]
66:  end
67:  if a.length >0
68:    if a[0] == "#!fdps" || a[0] == "#$fdps"
69:      a[0] = "//$fdps"
70:    else
71:      a[0][0]= "//"
72:    end
73:    ss += " "+a.join(" ")
74:  end
75:  print ss,"\n"
76:end
77:
78:
79:
80:
```

```
  81:print_header
  82:
  83:while s=gets
  84:  if s =~ /struct (.*) #!fdps (.*)/
  85:#    print s
  86:    struct_name = $1.downcase
  87:    struct_types=$2
  88:    lines=[]
  89:    instruct=true
  90:    while instruct
  91:      s=gets
  92:      if s=~/^(\s*)end(\s*)$/
  93:        instruct =false
  94:      else
  95:        lines.push s.chomp
  96:      end
  97:    end
  98:    print "typedef struct #{struct_name}{  //$fdps #{struct_types}\n"
  99:    lines.each{|s| print_fortran_member(s)}
 100:    print "} #{struct_name.capitalize};\n"
 101:  end
 102:end
```

で、

```
gravity> ruby convert_crystal_struct_to_c.rb user_defined.cr
#pragma once
/* Standard headers */
#include <math.h>
/* FDPS headers */
#include "FDPS_c_if.h"
typedef struct full_particle{  //$fdps FP,EPI,EPJ,Force
      //$fdps copyFromForce full_particle (pot,pot) (acc,acc)
      //$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps) (pos,pos)
      //$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
      long long id; //$fdps id
      double mass; //$fdps charge
      double eps;
      fdps_f64vec pos; //$fdps position
      fdps_f64vec vel; //$fdps velocity
      double pot;
      fdps_f64vec acc;
} Full_particle;
```

と、大丈夫かなこれ？あと何をするんだっけ？C 言語サンプルの Makefile だと

```
 FDPS_c_if.h $(SRC_CXX): $(HDR_USER_DEFINED_TYPE) Makefile
         $(FDPS_C_IF_GENERATOR) user_defined.h --output ./
```

で色々なものが全部できるはずで、Crystal のだと

```
FDPS_cr_if.cr:   FDPS_ftn_if.cpp  convert_f90_if_to_crystal.rb
        ruby convert_f90_if_to_crystal.rb FDPS_ftn_if.cpp > FDPS_cr_if.cr
```

で Crystal のインターフェース作ってて、 FDPS の C インターフェースでも FDPS_ftn_if.cpp は作ってるからこのままでいいのかな。

これ実行してと、なんかできるけど

```
crystal FDPS_cr_if.cr
In FDPS_cr_if.cr:98:13

  98 | (*pfunc_comp : Bool) : Void
       ^
Error: expecting token 'CONST', not '*'
```

コンパイルエラーか。まあそうだよね。FDPS_cr_if.cr をみると、、、

```
fun sort_particle=fdps_sort_particle(psys_num : Int32,
           (*pfunc_comp : Bool) : Void
```

これ括弧も閉じてないし全然変だわ。えーと、これどうなって欲しいんだろ？pfunc_comp の型の宣言を生成するのに失敗していると。これって、比較関数を C++ の FDPS に渡すやつだから、ポインタも C++ の構造体へのポインタで、別に意味ないしなんでもいいはずかしら。とりあえず

```
fun sort_particle=fdps_sort_particle(psys_num : Int32,
          pfunc_comp : Pointer(T), Pointer(T) -> Bool) : Void
```

みたいなのがでればいいと。

その辺いいとして、、、あれ、 theta=0 近くでないと答あわないというか、ep-sp の計算壊れてるか。これ SPJmonopole でないといけないから？

あ、だから、

```
fdps_calc_force_all
fdps_calc_force_making_tree
fdps_calc_force_and_write_back
fdps_sort_particle
fdps_calc_force_all_and_write_back
```

とかの、粒子型とか関数型もらう奴は、とりあえず自動生成しないでユーザープログラムの中で宣言すればいいか。I/O に Nacsio 使うと、、、あ、ベクトル型が違う問題があるか。FDPS のは Vec_Float64 というのを作ってるのね。まあとりあえず見苦しいけどこのままで、粒子 I/O は Particle 型使って、それと Full_particle との間で変換かければいいか。

コマンドラインオプションに clop は使えるのかな？ MPI 使いたいから、

メインプログラムだけは C++ の関数があって、それから Crystal で書いた crmain っていうのを呼び出す形だから、ARGV とかないよね、、、

C++ のメインプログラムには argc, argv があるから、これを渡してなんとかすればいいか。

と、一応動くようになったかな、、、、エネルギーも保存するし。

OpenMP と、あと MPI も、お、動いてますね。

(ここまで 1 週間くらい)

なんかできた気がします。

赤木 ：え、本当に

学生 C : まあ作者氏が前に作ったやつが一応動くところまできてて、FDPS の側が公式に C 言語対応したのでその辺変えたのと、コマンドラインパーザとか Nacsio とか使うようにしたくらいなので。

赤木 ：エネルギー保存とかどんな感じ？

学生 C :

```
gravity> ./mkplummer -Y -n 1024 | env LD_LIBRARY_PATH=../../src/fdps-crystal/ ~/src/fdps-crystal/fdpsc
FDPS on Crystal test code
     //==================================\\
     ||                                  ||
     || ::::::: ::::::. ::::::. .::::::. ||
     || ::      ::    : ::    : ::       ||
     || ::::::  ::    : ::::::'  `::::. ||
     || ::      ::::::' ::      `.....' ||
     ||     Framework for Developing     ||
     ||        Particle Simulator        ||
     ||     Version 5.0g (2019/09)       ||
     \\==================================//

       Home   : https://github.com/fdps/fdps
       E-mail : fdps-support@mail.jmlab.jp
       Licence: MIT (see, https://github.com/FDPS/FDPS/blob/master/LICENSE)
       Note   : Please cite the following papers.
                - Iwasawa et al. (2016, Publications of the Astronomical Society of Japan, 68, 54)
                - Namekata et al. (2018, Publications of the Astronomical Society of Japan, 70, 70)

       Copyright (C) 2015
         Masaki Iwasawa, Ataru Tanikawa, Natsuki Hosono,
         Keigo Nitadori, Takayuki Muranushi, Daisuke Namekata,
         Kentaro Nomura, Junichiro Makino and many others
******** FDPS has successfully begun. ********
====================================
 Paralleization infomation:
   # of processes is 1
   # of thread is    4
====================================
options # => #<CLOP:0x7fe793a2df60
 @dt=0.0078125,
 @dt_dia=1.0,
 @dt_end=5.0,
 @dt_out=2.0,
 @eps=0.05,
 @help=false,
 @init_out=false,
 @n=0,
 @ofname="p1k-t10.yml",
```

```
 @tol=0.5,
 @x_flag=false>
Failed to raise an exception: END_OF_STACK
[0x7fe79326ca86] ???
[0x7fe793238be3] __crystal_raise +35
[0x7fe79325ffe5] ???
[0x7fe79328de58] ???
[0x7fe793294359] ???
[0x7fe7932933fd] ???
[0x7fe7932821fe] ???
[0x7fe79326c950] crmain +256
[0x561b6340b7af] main +31
[0x7fe7920cac87] __libc_start_main +231
[0x561b6340b83a] ???
[0x0] ???
Unhandled exception: Error writing file: Broken pipe (IO::Error)
  from /usr/share/crystal/src/io/evented.cr:82:13 in 'unbuffered_write'
  from /usr/share/crystal/src/io/buffered.cr:144:9 in 'write'
  from /usr/share/crystal/src/io.cr:471:7 in 'write_utf8'
  from /usr/share/crystal/src/string.cr:4961:5 in 'to_s'
  from /usr/share/crystal/src/io.cr:174:5 in '<<'
  from /usr/share/crystal/src/io.cr:188:5 in 'print'
  from /usr/share/crystal/src/kernel.cr:125:3 in 'print'
  from mkplummer.cr:128:20 in 'nacswrite'
  from mkplummer.cr:173:3 in '__crystal_main'
  from /usr/share/crystal/src/crystal/main.cr:110:5 in 'main_user_code'
  from /usr/share/crystal/src/crystal/main.cr:96:7 in 'main'
  from /usr/share/crystal/src/crystal/main.cr:119:3 in 'main'
  from __libc_start_main
  from _start
  from ???
```

```
gravity> ./mkplummer -Y -n 1024 | env LD_LIBRARY_PATH=../../src/fdps-crystal/ ~/src/fdps-crystal/fdpsc
FDPS on Crystal test code
     //==================================\\
     ||                                  ||
     || ::::::: ::::::. ::::::. .::::::. ||
     || ::      ::    : ::    : ::       ||
     || ::::::  ::    : ::::::'  `::::. ||
     || ::      ::::::' ::      `.....' ||
     ||     Framework for Developing     ||
     ||        Particle Simulator        ||
     ||     Version 5.0g (2019/09)       ||
     \\==================================//

       Home   : https://github.com/fdps/fdps
       E-mail : fdps-support@mail.jmlab.jp
       Licence: MIT (see, https://github.com/FDPS/FDPS/blob/master/LICENSE)
       Note   : Please cite the following papers.
                - Iwasawa et al. (2016, Publications of the Astronomical Society of Japan, 68, 54)
```

```
                - Namekata et al. (2018, Publications of the Astronomical Society of Japan, 70, 70)

       Copyright (C) 2015
         Masaki Iwasawa, Ataru Tanikawa, Natsuki Hosono,
         Keigo Nitadori, Takayuki Muranushi, Daisuke Namekata,
         Kentaro Nomura, Junichiro Makino and many others
******** FDPS has successfully begun. ********
====================================
 Paralleization infomation:
   # of processes is 1
   # of thread is    4
====================================
options # => #<CLOP:0x7f175a086f60
 @dt=0.0078125,
 @dt_dia=1.0,
 @dt_end=5.0,
 @dt_out=2.0,
 @eps=0.05,
 @help=false,
 @init_out=false,
 @n=0,
 @ofname="p1k-t10.yml",
 @tol=0.25,
 @x_flag=false>
Failed to raise an exception: END_OF_STACK
[0x7f17598c5a86] ???
[0x7f1759891be3] __crystal_raise +35
[0x7f17598b8fe5] ???
[0x7f17598e6e58] ???
[0x7f17598ed359] ???
[0x7f17598ec3fd] ???
[0x7f17598db1fe] ???
[0x7f17598c5950] crmain +256
[0x5641650567af] main +31
[0x7f1758723c87] __libc_start_main +231
[0x56416505683a] ???
[0x0] ???
Unhandled exception: Error writing file: Broken pipe (IO::Error)
  from /usr/share/crystal/src/io/evented.cr:82:13 in 'unbuffered_write'
  from /usr/share/crystal/src/io/buffered.cr:144:9 in 'write'
  from /usr/share/crystal/src/io.cr:471:7 in 'write_utf8'
  from /usr/share/crystal/src/string.cr:4961:5 in 'to_s'
  from /usr/share/crystal/src/io.cr:174:5 in '<<'
  from /usr/share/crystal/src/io.cr:188:5 in 'print'
  from /usr/share/crystal/src/kernel.cr:125:3 in 'print'
  from mkplummer.cr:128:20 in 'nacswrite'
  from mkplummer.cr:173:3 in '__crystal_main'
  from /usr/share/crystal/src/crystal/main.cr:110:5 in 'main_user_code'
  from /usr/share/crystal/src/crystal/main.cr:96:7 in 'main'
  from /usr/share/crystal/src/crystal/main.cr:119:3 in 'main'
  from __libc_start_main
  from _start
```

```
  from ???
```

```
gravity> ./mkplummer -Y -n 1024 | env LD_LIBRARY_PATH=../../src/fdps-crystal/ ~/src/fdps-crystal/fdpsc
FDPS on Crystal test code
     //==================================\\
     ||                                  ||
     || ::::::: ::::::. ::::::. .:::::::. ||
     || ::      ::    : ::    : ::       ||
     || ::::::  ::    : ::::::'  `:::::. ||
     || ::      ::::::' ::      `......' ||
     ||     Framework for Developing     ||
     ||        Particle Simulator        ||
     ||     Version 5.0g (2019/09)       ||
     \\==================================//

       Home   : https://github.com/fdps/fdps
       E-mail : fdps-support@mail.jmlab.jp
       Licence: MIT (see, https://github.com/FDPS/FDPS/blob/master/LICENSE)
       Note   : Please cite the following papers.
                - Iwasawa et al. (2016, Publications of the Astronomical Society of Japan, 68, 54)
                - Namekata et al. (2018, Publications of the Astronomical Society of Japan, 70, 70)

       Copyright (C) 2015
         Masaki Iwasawa, Ataru Tanikawa, Natsuki Hosono,
         Keigo Nitadori, Takayuki Muranushi, Daisuke Namekata,
         Kentaro Nomura, Junichiro Makino and many others
******** FDPS has successfully begun. ********
====================================
 Paralleization infomation:
   # of processes is 1
   # of thread is    4
====================================
options # => #<CLOP:0x7f1622364f60
 @dt=0.0078125,
 @dt_dia=1.0,
 @dt_end=5.0,
 @dt_out=2.0,
 @eps=0.05,
 @help=false,
 @init_out=false,
 @n=0,
 @ofname="p1k-t10.yml",
 @tol=0.1,
 @x_flag=false>
Failed to raise an exception: END_OF_STACK
[0x7f1621ba3a86] ???
[0x7f1621b6fbe3] __crystal_raise +35
[0x7f1621b96fe5] ???
[0x7f1621bc4e58] ???
[0x7f1621bcb359] ???
```

```
[0x7f1621bca3fd] ???
[0x7f1621bb91fe] ???
[0x7f1621ba3950] crmain +256
[0x5641f6b327af] main +31
[0x7f1620a01c87] __libc_start_main +231
[0x5641f6b3283a] ???
[0x0] ???
Unhandled exception: Error writing file: Broken pipe (IO::Error)
  from /usr/share/crystal/src/io/evented.cr:82:13 in 'unbuffered_write'
  from /usr/share/crystal/src/io/buffered.cr:144:9 in 'write'
  from /usr/share/crystal/src/io.cr:471:7 in 'write_utf8'
  from /usr/share/crystal/src/string.cr:4961:5 in 'to_s'
  from /usr/share/crystal/src/io.cr:174:5 in '<<'
  from /usr/share/crystal/src/io.cr:188:5 in 'print'
  from /usr/share/crystal/src/kernel.cr:125:3 in 'print'
  from mkplummer.cr:128:20 in 'nacswrite'
  from mkplummer.cr:173:3 in '__crystal_main'
  from /usr/share/crystal/src/crystal/main.cr:110:5 in 'main_user_code'
  from /usr/share/crystal/src/crystal/main.cr:96:7 in 'main'
  from /usr/share/crystal/src/crystal/main.cr:119:3 in 'main'
  from __libc_start_main
  from _start
  from ???
```

こんなふうで、Opening Angle である -T でのパラメータ小さくすると小さくなるので大丈夫じゃないですかね。

赤木 ：エネルギー保存は 1e-4, 1e-5, 2e-6 くらいなわけね。まあもっともらしいわね。あ、これ、4 スレッドなの？

学生 C : はい、 OpenMP 指定を有効にしたらちゃんと複数スレッドで動きました。

赤木 ：あら、2 年前にはまだガーベージコレクタに問題があって、複数スレッドになってると別に Crystal の側は 1 スレッドでも駄目だったんだけど、今は大丈夫なのね。ちゃんと速くなる?

学生 C : 4 スレッドで 4 倍にはなってないですが 3 倍くらいですかね。1 万粒子の時。

赤木 ：MPI は？

学生 C : 同じような感じでした。

赤木 ：出力どれくらいだしてるかとかによるから、もうちょっとちゃんと調べないといけないけど、とりあえずそんなところかしらね。FDPS 使えるといきなり OpenMP と MPI で並列化したプログラムが動くから、大規模計算でもできるかも。

学生 C : でも、スパコンに Crystal はいってないのでは？

赤木 ：x86 で Linux の機械ならはいるかもしれないし。運用部門とご相談とかね。

学生 C : 富岳とかは駄目ですか？

赤木 ：Crystal 開発側でまだ x86 以外のサポートできてないみたい。ラズパイとかで動かした人はいるみたいだから、いれれば動くと思うけど。

まあ世界のほとんどのスパコンは x86 だから、とりあえずはそれでなんとかね。

あとは、相互作用計算のところを、今のプログラムでは Crystal で普通に書いた関数を FDPS の側で呼ぶんだけど、そこを SIMD ユニット使って高速化したいわね。

これは、FDPS の開発グループで作ってるのがあるから、公開されたら使うことでいきましょう。

再帰下降パーサあるから作ってみてくれてもいいけど？

学生 C：それはさすがに車輪の再々発明ですよね。

赤木 ：まあそうね。

17.1 課題

1. *crystalfdps*[5] を動かして、1024 粒子のプラマーモデルを時間積分し、タイムステップ、ソフトニング、オープニングアングルをかえるととエネルギー保存がどう変わるか検討せよ。
2. OpeMP, MPI での速度を、粒子数を 1 万から 100 万程度までかえて調べよ。

17.2 まとめ

1. FDPS を使ってみた。動く気がする。

17.3 参考資料

FDPS[6]

crystalfdps[7]

[5] https://github.com/jmakino/crystalfdps
[6] http://fdps.jmlab.jp/?lang=ja
[7] https://github.com/jmakino/crystalfdps

Chapter 18

SIMD 命令利用による高速化 (まだ中身ないです)

赤木 ：というわけで SIMD 命令利用だけど、これは FDPS の開発グループで作ってるのを使うことで。

学生 C：じゃあ、まだ今日は何もなしですね?

赤木 ：そうしましょう。

Chapter 19

ちょっとしたツール

赤木 ：プラマーモデル作って時間積分だけではあんまり動きがないから、もうちょっと動くのを作ってみない？

学生 C : というと？

赤木 ：簡単なのはプラマーモデル 2 個の衝突ね。もっと沢山でもいいわ。

学生 C : 時間積分のプログラムに 2 つ読む機能とかつけますか？

赤木 ：そこは、処理速度としては問題あったりするけど、複数の単純なプログラムの組合せで実現する、っていう、まあ、Unix の精神でどうかしら？つまり、

- スナップショットの位置、速度をずらすプログラム
- 複数のスナップショットをまとめて 1 つにするプログラム

を作るわけ。位置、速度をずらすのは、パーサ使ってなんかもっと変なことをするのでもいいけど、まあ単純にベクトル与えてシフトでいいでしょう。複数のスナップショットは、文字列をコンマで区切って与えるのでどうかしら？

学生 C : とりあえずそれで 2 つとか 3 つぶつけるのができるわけですね。やってみます。

ずらすのは、hackcode1 から読むとこと書くとこ残して、あとオプションでもってきたベクトルを位置とか速度に足すのか。えーと、、、こんなかな。

まずずらすほうだけ。こんな感じです。

```
 1:require "clop"
 2:require "./nacsio.cr"
 3:include Math
 4:include Nacsio
 5:
 6:optionstr= <<-END
 7:  Description: program to shift the position and velocity of a snapshot
 8:  Long description: program to shift the position and velocity of a snapshot
 9:
10:  Short name:-x
11:  Long name:  --shift-pos
12:  Value type:  float vector
13:  Variable name:dx
```

```
14:  Default value:0.0,0.0,0.0
15:  Description:Shift value for position
16:  Long description:     Shift value for position
17:
18:  Short name:-v
19:  Long name:  --shift-vel
20:  Value type:  float vector
21:  Variable name:dv
22:  Default value:0.0,0.0,0.0
23:  Description:Shift value for velocity
24:  Long description:     Shift value for velocity
25:END
26:
27:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
28:options=CLOP.new(optionstr,ARGV)
29:
30:class Body
31:  YAML.mapping(
32:    pos: {type: Vector3, key: "r",default: [0.0,0.0,0.0].to_v,},
33:    vel: {type: Vector3, key: "v",default: [0.0,0.0,0.0].to_v,},
34:  )
35:end
36:
37:Nacsio.update_commandlog
38:Nacsio.repeat_on_snapshots(Body.from_yaml(""), true){|pp|
39:  pp.each{|p|
40:    p.p.pos += options.dx.to_v
41:    p.p.vel += options.dv.to_v
42:    p.print_particle
43:  }
44:}
```

赤木 ：じゃあまずは説明ね。

学生 C : はい。25 行目までは基本的にオプションの文字列で、 -r, -v で位置、速度のシフト量です。変数名は dx. dv です。

30 行目からは粒子型の宣言で、今回位置と速度だけ使うので、それだけの粒子にしてます。

37 行目は update_commandlog で、標準入力からスナップショットファイルのヘッダ部分読んで、自分のコマンド追加して書くものです。

38 行目の repeat_on_snapshots は、nacsio.cr に追加した関数で、これ自体は

```
1:  def repeat_on_snapshots(p = Particle.new, read_all = false)
2:    sp= CP(typeof(p)).read_particle
3:    no_time = (sp.y["t"] == nil )
4:    while sp.y != nil
5:      pp=[sp]
6:      time = sp.y["t"].as_f  unless read_all
7:      while (sp= CP(typeof(p)).read_particle).y != nil &&
8:            (read_all || no_time || sp.y["t"].as_f == time)
9:        pp.push sp
```

```
10:     end
11:     yield pp
12:   end
13: end
```

こういうものです。スナップショットを 1 個読んだら、というのは、時刻が変わったら次とみなすとして、なんかする、という処理一般に使える関数があると便利、ということで、 nacsplot2.cr とかであったループ構造を関数にしてます。

2 行目でまずは粒子 1 つ読み込みます。それから、一般的に使えるように、ということで、スナップショットの中に時刻データがなかったら、というのを 3 行目でチェックしてます。4 行目の while は、粒子が読める限り読むものですが、ここでは読んでないことに注意して下さい。ここでは、最初は sp には 2 行目で読んだものが入ってます。

while の下で配列 pp を作って、最初の要素をさっき読んだ sp にして、6 行目で時刻も設定します。但し、引数の read_all が真なら時刻無視して全部読むので時刻設定しません。7 行目からの while で、粒子読んで、時刻見る時は同じ時刻かどうかチェックします。で、チェックしないか、同じ時刻なら pp に粒子追加します。この while 抜けるのは、読んだ粒子の時刻が違うか、あるいは全部読んだ時ですね。そこで、11 行目の

```
11:     yield pp
```

が実行されます。この yield は、この関数呼び出し側の

```
Nacsio.repeat_on_snapshots(Body.from_yaml(""), true){|pp|
  pp.each{|p|
    p.p.pos += options.dx.to_v
    p.p.vel += options.dv.to_v
    p.print_particle
  }
}
```

のブロックの {} の中に「置き換えられる」ということのようです。但し、{} の中はこの関数の中というより呼び出し側の中で、呼び出し側の変数とか使えるみたいです。 a が配列だとして、

```
s=1
a.each{|x| x +=s}
```

とか書くのと同じ感じですね。repeat_on_snapshots は粒子と、 read_all の 2 つの引数ですが、粒子は CP(T) 型を T に型いれて実体化したもので、これは実は typeof(p) とするだけなので型だけあればよいので Body.from_yaml(””) を渡してます。 new でなくて from_yaml なのは、Body クラスの中身が YAML.mapping だけなので、from_yaml しか粒子作る方法がないからです。

で、 {|pp| と書いておくと、この pp に yield pp の中身が入ってそのあとの } までが実行されて、それが終わると repeat_on_snapshots の次の行に戻るわけです。

で、処理本体は

```
pp.each{|p|
  p.p.pos += options.dx.to_v
  p.p.vel += options.dv.to_v
  p.print_particle
}
```

で、粒子の、宣言した構造体になってる側で pos, vel にそれぞれコマンドラインオプションで指定された値を加えて、そのあとで yaml 側に残っているデータと合わせて出力、をくり返します。

赤木：なるほど。プログラム本体は repeat_on_snapshots の中身がほぼ全部ということね。わりと全体短くていい感じね。実行結果は？

学生 C：例えばこんな感じですね。

```
gravity> ./mkplummer -n 3 -Y -s 1
--- !CommandLog
command: ./mkplummer -n 3 -Y -s 1
log: Plummer model created
--- !Particle
id: 0
t: 0.0
m: 0.3333333333333333
r:
  x: 0.6959348429044219
  y: 0.5014594064614821
  z: -0.05509228685899097
v:
  x: -0.07083562690974746
  y: 0.369915247866214
  z: -0.4293037912428686
--- !Particle
id: 1
t: 0.0
m: 0.3333333333333333
r:
  x: -0.3818939282329637
  y: -0.10402804359047872
  z: 0.10433845010474857
v:
  x: 0.5625890492844403
  y: 0.14162824345185604
  z: 0.5622358681719519
--- !Particle
id: 2
t: 0.0
m: 0.3333333333333333
r:
  x: -0.31404091467145817
  y: -0.39743136287100317
  z: -0.04924616324575757
v:
  x: -0.49175342237469283
  y: -0.51154349131807
  z: -0.13293207692908335
```

```
gravity> ./mkplummer -n 3 -Y -s 1|./nacsshift -x 2,3,4
--- !CommandLog
```

```
command: './mkplummer -n 3 -Y -s 1

  ./nacsshift -x 2,3,4'
log: Plummer model created
Unhandled exception: Expected sequence, not YAML::Nodes::Mapping at line 7, column 3 (YAML::ParseExcep
  from /usr/share/crystal/src/yaml/nodes/nodes.cr:30:9 in 'raise'
  from /usr/share/crystal/src/yaml/from_yaml.cr:104:5 in 'new'
  from nacsio.cr:13:7 in 'initialize'
  from nacsio.cr:12:3 in 'new'
  from nacsshift.cr:45:3 in 'initialize'
  from nacsshift.cr:45:3 in 'new'
  from /usr/share/crystal/src/yaml/from_yaml.cr:12:3 in 'from_yaml'
  from nacsio.cr:108:30 in 'read_particle'
  from nacsio.cr:96:5 in 'read_particle'
  from nacsio.cr:149:5 in '__crystal_main'
  from /usr/share/crystal/src/crystal/main.cr:105:5 in 'main_user_code'
  from /usr/share/crystal/src/crystal/main.cr:91:7 in 'main'
  from /usr/share/crystal/src/crystal/main.cr:114:3 in 'main'
  from __libc_start_main
  from _start
  from ???
```

座標の数字が 2,3,4 増えてるのがわかるかと。

赤木 ：なかなか素晴らしいわね。コマンドも何を実行したかが残るのね。

学生 C : そうです。次は複数スナップショットですが、、、これ、今の nacsio ではできないですよね？そもそもスナップショットを標準入力からしか読まないじゃないですか？

赤木 ：そうねえ、まあ、だから、複数ファイルをつなげて標準入力から読んで 1 つに、でもいいんだけど、一応複数ファイルを、でやってみて。なので、 nacsio を色々な修正していいから。

学生 C : わかりました。

といっても、えーと、どうするんだ？ read_particle はファイル引数にして、今までのが動くようにするには STDIN がデフォルト値ならいいか。これは簡単だな。

update_commandlog は、、、これは今、入力からログ読んで、書くまでやっちゃってるけど、そうじゃなくて読んでその値が入ってるインスタンス変数が返るようにすればいいと。じゃあそういう関数を read_commandlog で作って、で、 read もファイルをもらって、今の配列に追加、、、するようにもうなってるか。だから

```
c = CommandLog.new
fnames.each{|fn|
  File.open(fn,"r"){|f|
    c1 = read_commandlog(f)
    read(body,ybody,f)
    c.command +=  "\n"+c1.command
    c.log += "\n"+c1.log
  }
}
```

こんなので、粒子もコマンドログも追加するだけかな。

というわけで、、、なんかできたかな。

こんなふうです。

```
 1:require "clop"
 2:require "./nacsio.cr"
 3:include Math
 4:include Nacsio
 5:
 6:optionstr= <<-END
 7:  Description: program to add multiple snapshots to create one file
 8:  Long description: program to add multiple snapshots to create one file
 9:
10:  Short name:-i
11:  Long name:  --input-files
12:  Value type:  string
13:  Variable name:fnames
14:  Default value:none
15:  Description:comma-separated list of input files
16:  Long description:     comma-separated list of input files
17:
18:  Short name:-r
19:  Long name:  --reset-id
20:  Value type:  bool
21:  Variable name:reset_id
22:  Description:if true, reset id of particles
23:  Long description:     if true, reset id of particles
24:
25:END
26:
27:clop_init(__LINE__, __FILE__, __DIR__, "optionstr")
28:options=CLOP.new(optionstr,ARGV)
29:def write(body, ybody)
30:  body.size.times{|i|
31:    CP.new(body[i], ybody[i]).print_particle
32:  }
33:end
34:def read(body,ybody, f)
35:  while (sp= CP(typeof(body[0])).read_particle(f)).y != nil
36:    body.push sp.p
37:    ybody.push sp.y
38:  end
39:end
40:
41:class Body
42:  YAML.mapping(
43:    id: {type: Int64, default: 0i64,},
44:  )
45:end
46:
47:body = Array(Body).new
48:ybody= Array(YAML::Any).new
49:
50:c = CommandLog.new
```

```
51:options.fnames.split(",").each{|fn|
52:  File.open(fn,"r"){|f|
53:    c1 = read_commandlog(f)
54:    c.command +=  "\n"+c1.command
55:    c.log += "\n"+c1.log
56:    read(body,ybody,f)
57:  }
58:}
59:print c.add_command.to_nacs
60:body.each_with_index{|b,i| b.id=i.to_i64} if options.reset_id
61:write(body,ybody)
```

赤木 ：あら、割合いい感じ？では「解説お願いね。

学生 C : まずオプションですが、 -i, -r の 2 つで、 -i でファイルネーム、-r は、これいれると id を連続番号にします。

29 行目からの write はさっきと同じですが、 34 行目からの read は、update_commandlog をなくして、入力ファイルをを引数にしてそこから読んでます。

41 行目からの Body は、 id だけですね。あとのデータさわらないので。

50 行目で、空のコマンドログを作っておきます。

で、 body, ybody を初期化して、 fnames を配列にしてそれぞれから読むのが 51 行目です、52 行目の File.open(fn, "r") のあとに {|f| ...} がくるのは、

```
f= File.open(fn, "r")
...
```

と変わらないですが、このブロック抜けたらファイル閉じるとかがあります。で、53 行目で コマンドログを読んで、54,55 行目で最初空のコマンドログに今読んだのを追加していきます。それから 56 行目で粒子を読んで配列に追加です。

ファイル読むのが終わったら、59 行目で、コマンドログに現在のコマンド名を追加して出力です。で、オプションで指定があれば 60 行目で id ふりなおして、最後に write で粒子を出力でおしまいです。

赤木 ：2048 粒子のプラマーモデルちょっと離したところからぶつけるとかしてみて。

学生 C : コマンドはこんなのですかね。

```
./mkplummer -n 2048 -Y -s 1 > p2k.yml
nacsshift < p2k.yml > tmp0 -x -3,0,0 -v 0.3,-0.2,0
nacsshift < p2k.yml > tmp1 -x 3,0,0 -v -0.3,0.2,0
nacsadd -r -i tmp0,tmp1 > 4k.yml
env LD_LIBRARY_PATH=../../src/fdps-crystal/ ~/src/fdps-crystal/fdpscr -O 4k-out.yml -t 20 -d 1 -T 0.5 
env GKS_WSTYPE=jpg nacsplot3 < 4k-out.yml -w 10
```

結果の絵はこんな感じです。

赤木 ：まあそれっぽいわよね。

19.1 課題

この章は息抜きなので課題はありません。

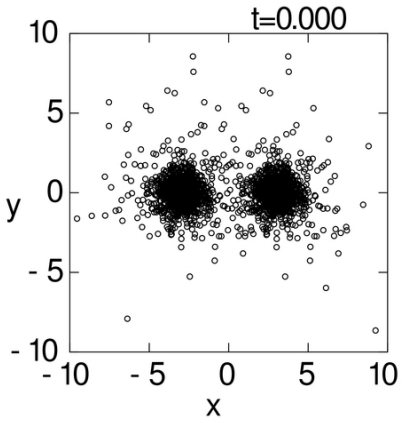

Figure 19.1: 初期条件

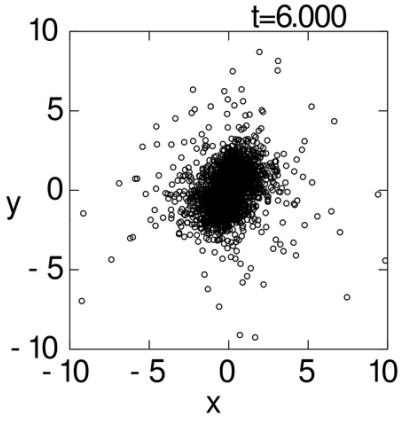

Figure 19.2: t=6

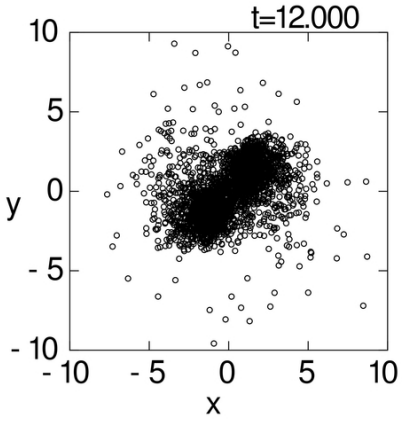

Figure 19.3: t=12

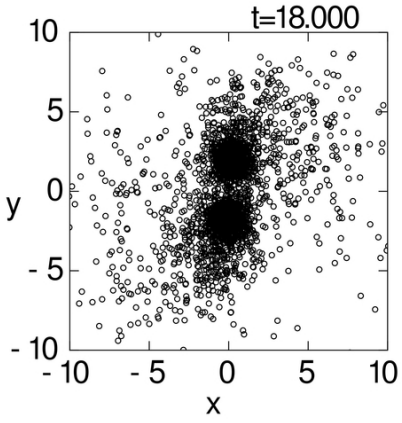

Figure 19.4: t=18

19.2 まとめ

1. スナップショットファイル加工するいくつかのツールを作ってみた。

19.3 参考資料

特になし。この文書で作っているプログラムの *github*[1]

[1] https://github.com/jmakino/numerical-calculation-with-crystal

Chapter 20

多次元配列

赤木 ：色々忙しいと思わず逃避行動にでているんだけど (だれが？)、Crystal で多次元配列ってどうなのかしら？

学生 C : どうなのかしら？といわれましても、、、ないですよね？ Ruby にないし。もちろん、配列の配列つくれば多次元配列だから、速度とか気にしなければそれでいいし、Ruby はそのレベルで速度とか気にするなら使えないし。

赤木 ：でもほら、田中さんの narray とか、あと Python の numpy とか、ライブラリ使えば速いみたいなのがあるじゃない?そういう感じのが Crystal にもあってもいいと思わない?

学生 C : あればいいんでしょうけど、えーと、具体的に何が欲しいと?

赤木 :要するに、普通に多次元配列宣言して、その要素にアクセスしたいわけ。例えば Fortran なら

```
        subroutine matmul(a,b,c,n)

integer*4 n
        real*8 a(n,n), b(n,n), c(n,n)
        ...
end
```

みたいな感じで配列かけて、多次元配列のサイズを関数に渡せるし、 C も C99 ならできるじゃない? C++ は未来永劫これができそうにないけど。

学生 C : C++ なら、サイズと、実際のデータエリアへのポインタもった構造体とか作りますよね？

赤木 ：まあそうね。

学生 C : で、Crystal でどうしたいと?

赤木 ：どっちいうと C++ でやるみたいにサイズと、実際のデータエリアへのポインタもったクラス作って、でも、なるべくそれを自然に使うみたいな。例えば Ruby の NArray だと

```
  x =Narray.float(10)
  xy =Narray.float(3,3)
```

とかで、長さ 10 の 1 次元配列とか、3x3 の 2 次元配列ができて、2 次元配列のほうは

```
  xy[1,2] = 3
  pp! xy[1,2]
```

みたいな感じで、 [1][2] みたいな書き方じゃなくてコンマ区切りでいいのと、あと、Fortran や C のプログラムに配列のポインタを渡せる、言い換えると、内部で Ruby の Array を使ってなくて、連続アドレス領域をとってデータがおいてあるわけ。だから、これ使うと色々なライブラリが簡単に使えるのね。

学生 C : えー、でも、Array じゃないんだとどうするんですか？全部 C で書くとか？そんなのできませんよ。

赤木 ：あ、Crystal その辺便利で、Pointer ってのがあるの。それで C みたいに malloc とかできるの。で、面白いのは、 free はないのね。プログラムで使わなくなったらガーベージコレクタが自動的にケアするから。なので、こういうの使っても C/C++ みたいなややこしいバグとかは起こらないの。

学生 C : まあそれはよさそうですが、、、多次元配列を a[2,3] みたいに書くって、 [] を 2 つ引数とる関数として定義するみたいなのがいりますよね？それできるんですか？

赤木 ： def [](i,j) とか def[]=(i,j,val) でできるみたいよ。

学生 C : それならなんとかなりそうですね。Ruby の narray とか Python の numpy と違って、ちゃんとコンパイラだから演算自体は Crystal で書いて別に遅くないはずだし。ちょっとかいてみます。

学生 C : えーと、Pointer というのがあるんですね。で、赤木さんがいってたみたいに malloc できると。あれ、Slice というのもあると。

```
While a pointer is unsafe because no bound checks are performed
when reading from and writing to it, reading from and writing to a
slice involve bound checks. In this way, a slice is a safe
alternative to Pointer.
```

えーと、範囲チェックが必要ならこっちを使えということですね。でも、いつもチェックしてたら遅い気がします。コンパイラオプションで切換えるとかですかね？まずはそういうの考えないで最低限だと、2 次元までなら

```
 1:  class Narray_F64
 2:    property :data
 3:    def initialize(nx : Int32,ny : Int32 = 1)
 4:      @data = Pointer(Float64).malloc(nx*ny)
 5:      @nx=nx
 6:      @ny=ny
 7:    end
 8:    def [](i)   @data[i] end
 9:    def [](i,j) @data[i*@ny+j]  end
10:    def []=(i,x)  @data[i]=x    end
11:    def []=(i,j,x)   @data[i*@ny+j]=x  end
12:  end
```

3 次元以上にしたければ @nz とかもつければいいと。initialize 最初は配列の次元数毎に書いたんだけど、中身は要するに使わない次元を 1 にするだけだったのでデフォルト引数ですむ。一応これ呼ぶプログラム作ると、

```
1:require "./narray0.cr"
2:
3:x =Narray_F64.new(10)
```

```
 4:x[1]=2.0
 5:pp! x
 6:xy =Narray_F64.new(4,4)
 7:16.times{|i| xy[i]=i.to_f64}
 8:xy[2,3]=-1.0
 9:4.times{|i|
10:  4.times{|j|print " ",xy[i,j]}
11:  print "\n"
12:}
13:pp! xy
```

これ実行してみると

```
gravity> crystal narray0test.cr
x # => #<Narray_F64:0x7f0a4d05ee60
 @data=Pointer(Float64)@0x7f0a4d069f60,
 @nx=10,
 @ny=1>
 0.0 1.0 2.0 3.0
 4.0 5.0 6.0 7.0
 8.0 9.0 10.0 -1.0
 12.0 13.0 14.0 15.0
xy # => #<Narray_F64:0x7f0a4d05e940
 @data=Pointer(Float64)@0x7f0a4d06dea0,
 @nx=4,
 @ny=4>
```

値もはいっているしそれっぽいですね。

赤木 ：そうね。でもこれだと Float64 しか使えないから、Float32 も使えるようにできる？

学生 C : 2 つ書けばよくないですか？

赤木 ：まあ、そういうことをしなくていいための機能が C++ とかでいうところのテンプレートで、 Crystal だと Generics というのね。多項式のところで、 Array から新しいクラス作るとか、あと MathVector でも使ってるけど、

```
class Foo
```

の代わりに

```
class Foo(T)
```

として、別に T でなくてもいいはずだけど、クラスが引数として型を持つようにできるのね。Array も

```
class Array(T)
```

で、 new は

```
Array(Float64).new(...)
```

とかするのと同じわけ。

学生 C : そうすると、

```
class Narray(T)
```

で、

```
  def initialize(nx : Int32,ny : Int32 = 1)
    @data = Pointer(T).malloc(nx*ny)
    @nx=nx
    @ny=ny
  end
```

ですね。これを narray1.cr にして

```
  1:require "./narray1.cr"
  2:
  3:x =Narray(Float64).new(10)
  4:x[1]=2.0
  5:pp! x
  6:xy =Narray(Float32).new(4,4)
  7:16.times{|i| xy[i]=i.to_f32}
  8:xy[2,3]=-1.0_f32
  9:4.times{|i|
 10:  4.times{|j|print " ",xy[i,j]}
 11:  print "\n"
 12:}
 13:pp! xy
```

これ実行してみると

```
gravity> crystal narray1test.cr
x # => #<Narray(Float64):0x7f8676555e60
 @data=Pointer(Float64)@0x7f8676560f60,
 @nx=10,
 @ny=1>
 0.0 1.0 2.0 3.0
 4.0 5.0 6.0 7.0
 8.0 9.0 10.0 -1.0
 12.0 13.0 14.0 15.0
xy # => #<Narray(Float32):0x7f86765559a0
 @data=Pointer(Float32)@0x7f8676566f50,
 @nx=4,
 @ny=4>
```

なるほど、大丈夫ですね。ちゃんと x と xy で違う型ですね。あとなんでしたっけ？範囲チェックですね？

赤木 ：そう。Slice 使ってみて。

学生 C : 単に使うだけなら

```
  def initialize(nx : Int32,ny : Int32 = 1)
    @data = Slice(T).new(Pointer(T).malloc(nx*ny), nx*ny)
    @nx=nx
    @ny=ny
  end
```

ですかね。これで @data は Pointer じゃなくて Slice になるんですよね？

赤木 ：そのはず。なんかテスト書いて範囲外アクセスとかしてみて。

学生 C：ではこんなので

```
  1:require "./narray2.cr"
  2:
  3:x =Narray(Float64).new(10)
  4:x[1]=2.0
  5:pp! x
  6:xy =Narray(Float64).new(4,4)
  7:16.times{|i| xy[i]=i.to_f64}
  8:xy[2,3]=-1.0
  9:4.times{|i|
 10:  4.times{|j|print " ",xy[i,j]}
 11:  print "\n"
 12:}
 13:pp! x[11]
```

```
gravity> crystal narray2test.cr
x # => #<Narray(Float64):0x7f4a9a70af00
 @data=Slice[0.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 @nx=10,
 @ny=1>
 0.0 1.0 2.0 3.0
 4.0 5.0 6.0 7.0
 8.0 9.0 10.0 -1.0
 12.0 13.0 14.0 15.0
Unhandled exception: Index out of bounds (IndexError)
  from /usr/share/crystal/src/indexable.cr:903:8 in '[]'
  from narray2.cr:8:17 in '[]'
  from narray2test.cr:13:1 in '__crystal_main'
  from /usr/share/crystal/src/crystal/main.cr:115:5 in 'main_user_code'
  from /usr/share/crystal/src/crystal/main.cr:101:7 in 'main'
  from /usr/share/crystal/src/crystal/main.cr:127:3 in 'main'
  from /lib/x86_64-linux-gnu/libc.so.6 in '__libc_start_main'
  from /home/makino/.cache/crystal/crystal-run-narray2test.tmp in '_start'
  from ???
x[11] # =>
```

学生 C：おお、ちゃんと x[11] は範囲外だとでますね。narray1.cr を使うと

```
 1:require "./narray1.cr"
 2:
 3:x =Narray(Float64).new(10)
 4:x[1]=2.0
 5:pp! x
 6:xy =Narray(Float64).new(4,4)
 7:16.times{|i| xy[i]=i.to_f64}
 8:xy[2,3]=-1.0
 9:4.times{|i|
10:  4.times{|j|print " ",xy[i,j]}
11:  print "\n"
12:}
13:pp! x[11]
```

```
gravity> crystal narray2test1.cr
x # => #<Narray(Float64):0x7fd0ab8bee60
 @data=Pointer(Float64)@0x7fd0ab8c9f60,
 @nx=10,
 @ny=1>
 0.0 1.0 2.0 3.0
 4.0 5.0 6.0 7.0
 8.0 9.0 10.0 -1.0
 12.0 13.0 14.0 15.0
x[11] # => 0.0
```

なんか知らないけど 0 が戻りますね。

赤木 ：これ、Slice にしておくと、範囲チェックができる他に色々いいことがあるのね。ドキュメントみてもらうとわかるけど、Slice には Array にあるような便利な関数がほとんどあるから、each とか map! とか使えるようになるわけ。map もあるけどこれは配列になっちゃうからそのままではちょっとね。

学生 C : sort! とかもできるわけですか？

赤木 ：できるわよ。ただ、今のだと、 x.@data.sort! とかでちょっと見苦しいわね。method_missing 使って

```
macro method_missing(call)
  @data.{{call}}
end
```

としておけば、 Narray に対して知らない関数がきたら @data に投げる、とできるから、これで sort! とか map とかなんでも使えるわね。

学生 C : ちょっと試しに、、、

```
1:require "./narray2.cr"
2:
3:xy =Narray(Float64).new(4,4)
4:16.times{|i| xy[i]=i.to_f64}
```

```
 5:xy[2,3]=-1.0
 6:4.times{|i|
 7:  4.times{|j|print " ",xy[i,j]}
 8:  print "\n"
 9:}
10:xy.sort!
11:4.times{|i|
12:  4.times{|j|print " ",xy[i,j]}
13:  print "\n"
14:}
15:
```

を実行すると

```
gravity> crystal narray2test2.cr
 0.0 1.0 2.0 3.0
 4.0 5.0 6.0 7.0
 8.0 9.0 10.0 -1.0
 12.0 13.0 14.0 15.0
 -1.0 0.0 1.0 2.0
 3.0 4.0 5.0 6.0
 7.0 8.0 9.0 10.0
 12.0 13.0 14.0 15.0
```

おおお、確かに実行できてソートもされますね。2 次元配列こんなふうにソートすることはあんまりないとは思いますが、map! とかはいいかもですね。

赤木：で、これでいいんだけど、「コンパイルオプションによって Pointer か Slice か切換える」「でも、 Slice の機能は使えるようにする」ってできる？

学生 C：まずポインタで領域とって、それに対して Slice 作成だから、添字でのアクセスでどっち使うかですね。コンパイルオプションで選ぶのは、少なくともクラス全体をマクロにすればよかったような、、、なので、こんなのですね。

```
 1:# narray.cr
 2:#
 3:# multi-dimensional array similar to NUMO:Narray
 4:#
 5:# Copyright 2020- J. Makino
 6:#
 7:macro use_narray(bound_check = false)
 8:class Narray(T)
 9:  property :data
10:  def Narray.range_check?
11:    {% if bound_check %}
12:      true
13:    {% else %}
14:      false
15:    {% end %}
16:  end
17:  def initialize(@nx : Int32, @ny : Int32 = 1, @nz : Int32 = 1)
```

```
  18:    n=@nx*@ny*@nz
  19:    @data = Slice(T).new(Pointer(T).malloc(n), n)
  20:    @datasl = @data
  21:    {% if bound_check %}
  22:      @data  = @datasl
  23:    {% end %}
  24:  end
  25:  def [](i)      @data[i] end
  26:  def [](i,j)    @data[i*@ny+j]  end
  27:  def [](i,j,k)  @data[(i*@ny+j)*@nz+k]   end
  28:  def []=(i,x)   @data[i]=x    end
  29:  def []=(i,j,x) @data[i*@ny+j]=x  end
  30:  def []=(i,j,k, x)  @data[(i*@ny+j)*@nz+k]=x   end
  31:  macro method_missing(call)
  32:    @datasl.\{{call}}
  33:  end
  34:  def each_index
  35:    @nx.times{|i|
  36:      @ny.times{|j|
  37:        @nz.times{|k|
  38:          yield i,j,k
  39:        }
  40:      }
  41:    }
  42:  end
  43:  def each_index(idim)
  44:    if idim == 0
  45:      @nx.times{|i| yield i}
  46:    elsif  idim == 1
  47:      @ny.times{|j|yield j}
  48:    elsif  idim == 2
  49:      @nz.times{|k|yield k}
  50:    end
  51:  end
  52:end
  53:end
  54:
  55:{% if flag?(:range_check) %}
  56:  use_narray(true)
  57:    {% else %}
  58:    use_narray(false)
  59:{% end %}
```

まず、クラス定義全体を

```
macro use_narray(bound_check = false)
...
end
```

で囲って、

```
use_narray(true)
```

で範囲チェックあり、 false か引数なしならチェックなしのコードがコンパイル時に生成されるようにします。そのなかでは

```
{% if bound_check %}
   foo
{% else %}
   bar
{% end %}
```

でどっちのコードになるか選べるわけです。チェックするかどうかは、initialize のほうで

```
{% if bound_check %}
  @data = Slice(T).new(Pointer(T).malloc(nx*ny*nz), nx*ny*nz)
  @datasl = @data
{% else %}
  @data = Pointer(T).malloc(nx*ny*nz)
  @datasl = Slice(T).new(@data,  nx*ny*nz)
{% end %}
```

として、チェックするなら data も datasl も Slice、しない時は data は Pointer のままで、としました。

```
@data = Pointer(T).malloc(nx*ny*nz)
@datasl = Slice(T).new(@data,  nx*ny*nz)
{% if bound_check %}
  @data = @datasl
{% end %}
```

のほうが格好よいですがまあ。あと、range_check? というクラス関数も作ってみました。で、最後に

```
{% if flag?(:range_check) %}
  use_narray(true)
    {% else %}
    use_narray(false)
{% end %}
```

で、コンパイル時に -Drange_check されるかどうかでどっちかのマクロが生成されます。

```
 1:require "narray"
 2:x =Narray(Float64).new(10)
 3:x[1]=2.0
 4:xy =Narray(Float64).new(4,4)
 5:16.times{|i| xy[i]=i.to_f64}
 6:xy[2,3]=-1.0
 7:4.times{|i|
 8:  4.times{|j|print " ",xy[i,j]}
 9:  print "\n"
10:}
11:xy.sort!
12:pp! xy
13:pp! Narray.range_check?
14:pp! x[11]
```

を範囲チェックありで実行すると

```
gravity> crystal run -Drange_check narray_test.cr
 0.0 1.0 2.0 3.0
 4.0 5.0 6.0 7.0
 8.0 9.0 10.0 -1.0
 12.0 13.0 14.0 15.0
xy # => #<Narray(Float64):0x7fc6e6727680
 @data=
  Slice[-1.0,
   0.0,
   1.0,
   2.0,
   3.0,
   4.0,
   5.0,
   6.0,
   7.0,
   8.0,
   9.0,
   10.0,
   12.0,
   13.0,
   14.0,
   15.0],
 @datasl=
  Slice[-1.0,
   0.0,
   1.0,
   2.0,
   3.0,
   4.0,
   5.0,
   6.0,
   7.0,
   8.0,
   9.0,
   10.0,
   12.0,
   13.0,
   14.0,
   15.0],
 @nx=4,
 @ny=4,
 @nz=1>
Narray.range_check? # => true
Unhandled exception: Index out of bounds (IndexError)
  from /usr/share/crystal/src/indexable.cr:903:8 in '[]'
  from lib/narray/src/narray.cr:56:1 in '[]'
  from narray_test.cr:14:1 in '__crystal_main'
```

```
   from /usr/share/crystal/src/crystal/main.cr:115:5 in 'main_user_code'
   from /usr/share/crystal/src/crystal/main.cr:101:7 in 'main'
   from /usr/share/crystal/src/crystal/main.cr:127:3 in 'main'
   from /lib/x86_64-linux-gnu/libc.so.6 in '__libc_start_main'
   from /home/makino/.cache/crystal/crystal-run-narray_test.tmp in '_start'
   from ???
x[11] # =>
```

なしだと

```
gravity> crystal run narray_test.cr
 0.0 1.0 2.0 3.0
 4.0 5.0 6.0 7.0
 8.0 9.0 10.0 -1.0
 12.0 13.0 14.0 15.0
xy # => #<Narray(Float64):0x7f43c0f5aed0
 @data=Pointer(Float64)@0x7f43c0f5ef30,
 @datasl=
  Slice[-1.0,
   0.0,
   1.0,
   2.0,
   3.0,
   4.0,
   5.0,
   6.0,
   7.0,
   8.0,
   9.0,
   10.0,
   12.0,
   13.0,
   14.0,
   15.0],
 @nx=4,
 @ny=4,
 @nz=1>
Narray.range_check? # => false
x[11] # => 0.0
```

で、ちゃんとコンパイルオプションで @data の型が変わって、範囲チェックの結果も変わってるのがわかります。

赤木 ：なるほど。最低限のことはこれでできる感じね。あとは、C/C++ に比べて速度は、とか、SIMD 化はどうするの？とかまだ色々あるけど、それはおいおいね。

20.1 課題

この章は息抜きなので課題はありません。

20.2 まとめ

1. 多次元配列を作った。

20.3 参考資料

特になし。Crystal 言語の Pointer, Slice, Macro 等をみておこう。

Chapter 21

単位系ふたたび

赤木 ：重力多体問題の単位系の話、だいぶ前に話をしたんだけど、今一つ色々な場合に使えるようになってない気がしたから整理するのにちょっとつきあって？

学生 C：太陽がとか木星がとかの話でしたっけ？

赤木 ：そう、それ。まずはは太陽と地球の話からね。

学生 C：了解です。

21.1 太陽と地球

赤木 ： もちろん、普通に考えると、太陽の質量が SI 単位系で¡$2\times 10^31 \rm kg> で、重力定数がいくつで、とかで計算できるんだけど。無駄に桁が多くわかりにくいので、太陽の質量を 1、1 天文単位 (1AU) を 1、重力定数を 1 にする単位系で考える、という話はしたわね？

学生 C：はい。で、そうすると、地球の軌道周期はその単位系で 2π になるので、逆にいうと時間の単には $1/(2\pi)$ 年ということになるわけですね。

赤木 ：そうそう。なので、惑星形成のシミュレーションや太陽系の安定性のシミュレーションだとこの単位系を使うことが多いわね。

学生 C：まあこれはいいですよね。

21.2 遠くの惑星

赤木 ：じゃあ、順番に、ということで、木星とか、外側の惑星なら？

学生 C：うーん、単位 1AU のままででも問題ないですよね？

赤木 ：まあそうね。1000AU のところを回ってる TNO なら？

学生 C：そんなのありましたっけ？

赤木 ：まああってもいいでしょう。この時に、軌道長半径が 1 になるような単位系にしたら時間の単位はどうなるかしら？

学生 C：軌道周期がその時間単位で 2π になるのは同じですね。軌道周期は 1000AU ならケプラーの第三法則から $10^{4.5}$ 年ですから、時間単位は $10^{4.5}/(2\pi)$ 年、5032 年とかですね。

赤木 ：多分それであってると思う。その辺になると、シミュレーションもやっぱり軌道長半径が 1

とかの単位系のほうが無難よね。誤差推定とか時間刻み決めるのとかがちゃんと無次元化されてないと滅茶苦茶な答になったりするし。

学生 C : それはちゃんと無次元化できてないコード書くのが悪くないですか？

赤木 ：そうだけど、そうなっちゃってることもあるのよ。

学生 C : まあそうかもしれないですが、、、

赤木 ：というわけでここまでの話を整理するわね、私達は基本的に系 (惑星系なら中心星) の質量が $M = 1$dir、系の大きさ (惑星系ならどれかの惑星の軌道長半径くらいのもの、地球なら天文単位) $L = 1$ 重力定数 $G = 1$ の単位系を考えますと、そうすると、時間の単位は自動的に決まって、これは L のところでの軌道の周期が 2π になることからわかると。なので、ここでは 時間の単位は $T = 1/(2\pi)$ 年となると。

学生 C : はい。

赤木 ：では、距離の単位を 1AU じゃなくて aAU にしたら？

学生 C : さっきの話で軌道周期が $a^{3/2}$ なので、時間の単位は $T = a^{3/2}/(2\pi)$ 年ですね。

赤木 ：そう。これで話の半分はきたわけ。

学生 C : あとの半分ってなんですか？

赤木 ：ここまでで長さの単位と時間の関係がでたから、今度は質量の単位ね。

学生 C : どんな場合ですか？

21.3 系外惑星

赤木 ：一つは、中心星が太陽じゃない場合ね。例えば系外惑星。中心星の質量が m 太陽質量で、まず長さの単位が 1AU だと時間の単位は？

学生 C : 地球と同じような円軌道を考えると速度が $\sqrt{m}$ 倍だから、周期が $1/\sqrt{m}$ になって、時間の単位は $T = 1/(2\pi\sqrt{m})$ 年ですね。

赤木 ：そうね。じゃあ長さの単位を aAU にしたら？

学生 C : この場合でもやっぱりケプラーの第三法則で $a^{3/2}$ で長くなるので、

$$T = a^{3/2}/(2\pi\sqrt{m})\mathrm{yr} \tag{21.1}$$

ですね。

赤木 ：大変結構。これで、太陽質量と天文単位で考えてる限り、一般的な時間の単位の計算式、ということになるわね。ただ、天文学や宇宙物理では、スケールが大きくなるとパーセク使うから、こっちの式もだしときましょう。

21.4 パーセク

学生 C : パーセクってなんでしたっけ？

赤木 ：高校の地学でならうんだけど、、、とってないわよね、、、パララックスが 1 秒になる距離。

学生 C : 秒って角度の秒ですね？ 1 度の 1/3600?

赤木 ：そう。なので、

$$1\mathrm{pc} = 360 \cdot 60 \cdot 60/(2\pi)\mathrm{AU} \tag{21.2}$$

というのが定義になって、数値としては

$$1\mathrm{pc} = 2.0626 \times 10^5 \mathrm{AU} \tag{21.3}$$

となるわね。

学生 C : なるほど。

赤木 ： で、まずは、質量の単位が太陽質量で長さの単位が 1 パーセクの時に時間の単位は？

学生 C : それはつまり長さの単位が 2.0626×10^5AU ということで、その 3/2 乗を 2π で割って、1.49093×10^7 年ですね。

赤木 ：じゃあ a パーセクと m 太陽質量なら？

学生 C :

$$T = 1.49093 \times 10^7 a^{3/2}/\sqrt{m}\mathrm{yr} \tag{21.4}$$

となるだけですよね？

赤木 ：そう。天文単位の時の式とパーセクの時の式で、同じ式なのに係数違って気持ち悪いから、それぞれ単位が明確な式にしてみるわね。あと記号は軌道長半径みたいな a, m は止めて L,M にして

$$\left(\frac{T}{\mathrm{year}}\right) = \frac{1}{2\pi}\left(\frac{L}{\mathrm{AU}}\right)^{3/2}\left(\frac{M}{M_\odot}\right)^{-1/2} \tag{21.5}$$

$$\left(\frac{T}{\mathrm{year}}\right) = 1.49093 \times 10^7 \left(\frac{L}{\mathrm{pc}}\right)^{3/2}\left(\frac{M}{M_\odot}\right)^{-1/2} \tag{21.6}$$

これで $G = 1$ の時の単位系の換算は完璧、なはず。

学生 C :

赤木 ：

学生 C :

赤木 ：

学生 C :

赤木 ：

学生 C :

赤木 ：

学生 C :

赤木 ：

学生 C :

赤木 ：

学生 C :

赤木 ：

学生 C :
赤木 :
学生 C :
赤木 :

21.5 課題

この章は息抜きなので課題はありません。

21.6 まとめ

1. 単位系について整理しなおした。

21.7 参考資料

特になし。

Chapter 22

MPIの全てを5分で理解する (嘘)

学生 C : 今回これなんですか？

赤木 ：何かそういうものがあったほうがいいという神の声が聞こえたとか。

学生 C : はあ。

赤木 ：まあ全てっていうのはもちろん嘘で、考え方と、FDPS とか使うのに必要な最低限くらいね。

学生 C : まあそれなら、、、

赤木 ：君 MPI でプログラム書いたことある？

学生 C : ないです。

赤木 ：じゃあまあちょうどいい機会だから、付き合って。

学生 C : はい、、、

22.1 MPI の考え方

赤木 ：まず考え方ね。MPI って、基本的な考え方として、ネットワークでつながった沢山の計算機でそれぞれ勝手にプログラムが走って、それらがお互いに通信できる、というだけなの。例えば、メイルを送るプログラムと受け取るプログラムは通信してるし、ブラウザとウェブサーバーも通信しているわけね、MPI でも、基本的にはそれと同じで、いくつかの計算機で走ってるプログラム同士が通信するだけ。

学生 C : でもなんか色々な、MPI の関数呼んで初期設定したり、通信も MPI の関数使うんじゃないですか？

赤木 ：それはそう。普通のネットワークでの通信だと、相手は IP アドレスとか名前で指定するじゃない？でも並列プログラムって、スパコンで実行したり、研究室のサーバーで実行したり、あるいは手元のノートで実行したりするわけで、その時に IP アドレスとか名前とかをどう変更するかを自分で管理するのは面倒なわけで、MPI の初期化関数はその辺の面倒をみてくれるわけ。

初期化というか、最初のプログラムの実行開始の時ね。普通だと

```
mpirun -np 16 your_program [options of your program]
```

みたいな感じで、スパコンセンターだと色々センター固有のコマンドとかあるけど、ここでは -np 16 で 16 個の your_program を走らせる、と指定してるわけ。ここでは 16 個が全部同じ your_program

ね。色々すると例えば 8 個は program1 で残りは program2 なんてこともできるけど、普通しないと思う。

で、この 16 個がどの計算機でどう走るかは、MPI のインストールがちゃんとできていればどっかにある設定ファイルを使ってなんかするのね。基本的は特に指定しなければ 1 台の計算機の上で複数プロセスが走るはず。スパコンセンターとかだと「ジョブスクリプト」というのを書いて、それで使う計算機の台数とか、1 台の中でプロセス何個動かすかとか指定するの。

で、そうすると、MPI のプログラムの中では、プロセスに番組がふられて、これランクっていうのが MPI の中でのきまりだけど、自分のランクが何番かわかる (そういう関数がある) し、全部で何プロセスあるかとかもわかる、あと、通信も相手をランクで指定できるわけ。

なので、MPI 使って書いておけば、研究室のサーバーでも富岳とかのスパコンでも同じように使えるわけ。

学生 C : そういわれるとなんか素晴らしいものみたいに聞こえますが、そのわりには赤木さんいつも「MPI はクソ」「MPI は滅びるべきである」とかいってませんか？

赤木 : そうね、それは、MPI のやってくれることが本質的にわりと今いったこと、つまり、名前のかわりに番号で通信できるのと、あと自分の番号と全プロセスの数がわかる、とこれだけで、あとは全部自分でしないといけないからなの。

学生 C : え、でも、並列プログラムって、何台かの計算機とか、複数のプロセスとかで 1 つの計算するわけだから、どうしてもそうなるんじゃないですか？それぞれのプロセスが自分が k 計算するところを計算して、必要なデータを他からもらってくる必要があれば通信するわけですよね？

赤木 : 20 年くらい前までは並列プログラムってそうじゃなかったのよ。例えば、High Performance Fortran っていう言語があって、これは Fortran 90 の配列操作をまあ自動的に並列実行してくれる、というものだったわけ。大きな配列を宣言して、それをどう分割するかくらいを指定すれば、あとは自分の分担から外れた隣にある配列要素のアクセスとかしてもコンパイラが面倒みてくれたの。

学生 C : それは確かに素晴らしいように聞こえますが、、、でもそんなのが動く計算機がもうないからみんな MPI 使ってるんですよね？あればそっち使いますよね？

赤木 : まああるっちゃあるんだけど、、、、NEC の SX とか、、、ただ、配列操作を並列化、というのは最近のキャッシュがある計算機では絶対性能でないから、そういう計算機で性能だそうと思うとなんでもできる MPI 使うわけ。並列計算って大抵は速くしたいからするわけだから、性能でないと話にならないのね。

学生 C : なんか話が愚痴になってません？本題は MPI だったような気も。

赤木 : そうね、じゃあまず初期設定から

22.2 初期設定

赤木 : 言語は C ね。C++ でも他の言語でも普通 C のインターフェース使うから。

```
#include "mpi.h"
```

はしたとするとと。

```
int myid, numprocs;
MPI_Init(&argc,&argv);                    //初期化
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);//プロセス数を numprocs に入れる
MPI_Comm_rank(MPI_COMM_WORLD,&myid);     //自分の番号を myid に入れる
```

みたいな感じ。このあと色々やって、あとプログラムの終了の時には

```
MPI_Finalize
```

を呼ぶんだったような気がするわ。

学生 C： MPI_COMM_WORLD ってなんですか？

赤木 :これ、「コミュニケータ」というもので、通信する範囲を指定するのに使えるの。MPI_COMM_WORLD はデフォルトで、起動した MPI プロセス全部。

学生 C : 通信相手番号で指定するなら、それだけでよくないですか？何に使うんですかこれ？

赤木 ：あ、相手が 1 つならいいけど、MPI では 放送とか、逆に総和とかできるのね。あるプロセスのデータを他のプロセスに放送とか、逆に全プロセスのある変数の値の合計をあるプロセスにとか、あるいは全プロセスにとか。で、これ、全プロセスじゃなくて、グループに分けられると便利なこともある、というか、分ける必要があるプログラムがあるので、そのための機能ね。まああんまり使わないんだけど、、、

学生 C : わけるって、どんなふうにですか？

赤木 ：例えば LINPACK っていう連立 1 次方程式を解くプログラムだと、プロセスを 2 次元格子に並べて行列を分割してもたせるわけね。そうすると、縦方向の放送とか、横方向の放送とかがあると便利なの。

学生 C : なるほど。粒子法とかだと MPI_COMM_WORLD そのままですか？

赤木 ：大抵はそうだと思うわ。あと、1 対 1 通信と集合通信の話をすればいいんじゃないかな。というわけでまずは 1 対 1 通信ね。

22.3 1 対 1 通信

学生 C : えーと、今までの話だと、単に送るほうは送り先指定して送って、受け取るほうはどこからきたかを指定して受け取るだけですよね？

赤木 ：と思うわけだけど、実際はそうでもないのね。送る関数と受け取る関数はそれぞれこんな感じ:

```
int MPI_Send(const void *buf,
             int count, MPI_Datatype datatype,
     int dest,
     int tag,
     MPI_Comm comm);

int MPI_Recv(void *buf,
             int count,
     MPI_Datatype datatype,
     int source,
     int tag,
             MPI_Comm comm,
     MPI_Status * status)
```

学生 C : buf, count, datatype は、、、これなぜ単純にバイト数渡すんじゃなくて、さらには C のいろんな関数みたいに sizeof を渡すんでもなくて型そのものを渡すんですか？

赤木 ：まあ send/receive なら絶対これいらないわね。話だけでてきたけど総和とかしたいと型がいるから、そういう関数に合わせたんじゃないかしら？これ設計者のセンスが悪いと私は思うわ。

学生 C： dest, source はそれぞれ送り先と送信元のランクですね？ tag ってなんですか？

赤木 ：これが一致してないと受け取らないの。

学生 C : 何故そんな面倒なことを？

赤木 ：これは、ネットワークとかネットワークコントローラの性質を考えたからだと思うわ。ネットワークのメッセージって、送った準備に届く、という保証がないの。処理のなんかの都合で順番変わったりするわけ。そうすると、単純に順番でみてるととデータが入れ替わっても分からないから、それが起きないようにアプリケーションの側でちゃんと番号かなんかつけてね、としてるの。

学生 C : それ、MPI のライブラリ側でもできますよね？送るほうは順番数えて送る時にそれつけて送って、受け取るほうもちゃんと今の呼び出しが何回目か憶えてればいいだけじゃないですか？

赤木 ：ヨイトコロニキガツキマシタネ。私もそう思うわ。プロセスがマルチスレッドでプロセス a のスレッド i からプロセス b のスレッド j に送る、みたい場合だとそれでは上手くいかなく、、、ないか。これも別にスレッドが全部プロセスみたいなものと思って数えればいいわね。

この辺が MPI のアレなところなわけ。なんかユーザー側でしなくてもいいことをしないといけなくてそれが間違いの元になりえる、という。あと、受け取る側で受け取る数を指定しないといけないのもわりとウザいところね。C のいろんな入力関数みたいに、ちゃんと十分なバッファサイズを用意して、あふれたらあふれたと返す、とかなぜできないの?みたいな。

文句ばっかりいってもしょうがないわね。1 対 1 通信であと知っておいて欲しいのは、 Send/Recv の他に Isend/Irecv というのがあって、こっちは非同期に動く、つまり、この関数呼んでもすぐに戻ってきて別の処理にうつることができるんだけど、いつ通信が終わるかはわからなくて終わるのを待つ関数が別にあるの。逆にいうと Send/Recv は関数から戻ってきたら処理が終わってるのね。でも、 Send はユーザープログラムが送り終わった、というだけでネットワークのどこまでメッセージがいってるかはわからないの。

これ、2 つのプロセスがメッセージを交換するのも実は簡単ではない、ということなのね。例えば両方が MPI_Send したら、相手が受け取らないので Send が終わらなくてプログラムがハングアップするかもしれないわけ。で、なのでその時には MPI_Sendrecv を使うか、 Isend/Irecv 使うかどっちかにしないといけないの。

学生 C： Isend と Recv の組合せではいけないんですか？

赤木 ： あ、そうね。理屈ではいいような気がするわ。ちゃんと仕様調べて実験もしてみて？

学生 C： えー、なんか大変そう、、、

赤木 ： まあ気がむいたらでいいわ。あと集団通信ね。

22.4 集団通信

赤木 ：一杯あるんだけど基本的なものだけ。まず MPI_Barrier.

```
int MPI_Barrier( MPI_Comm comm );
```

これは、全プロセスがこれ呼ぶまで先に呼んだものは待つ、というだけ。こういうのバリア同期というのね。これだけが必要なことはあんまりないはずだけど、デバッグの時にはすごく使うかも。これもコミュニケータ引数なので、もしも COMM_WORLD でないのを使うとその範囲だけが止まるわけ。

学生 C : これはタグとかないんですか？

赤木 ：これはいらない、といってもさっきの話だと他のもいらないんだけど、これはとにかくいらないということみたい。2 つのバリアの順番が入れ替るとかはないから。

学生 C : なるほど。

赤木 : で、次、

```
int MPI_Bcast(void *buffer,
              int count, MPI_Datatype datatype,
              int root,
              MPI_Comm comm);
```

これは放送ね。ランク root のプロセスが同じデータを他の全プロセスに送る。

```
int MPI_Reduce(const void *sendbuf,
               void *recvbuf,
int count,
MPI_Datatype datatype,
               MPI_Op op,
int root,
MPI_Comm comm);
```

こっちは総和とか。これ、型と操作と両方指定するのなんか変な気がするけどまあ仕様はそうなってると。あと Allreduce というのもあってこっちは結果を全プロセスが受け取るの。

あとまだすごく一杯色々な関数あるけどそういうのが必要になったらあるかどうかは調べればいいと思うわ。それぞれが複数のところに一度に送るとかそれをまた合計するとかそういうの。

学生 C : FDPS 使うプログラムの中ではこの辺の MPI の関数直接呼ぶのはどうするんですか？

赤木 : COMM_WORLD 使ってれば、というか普通使ってるしそうでない時は PS::CommInfo::setCommunicator とか使って設定しているはずだからそのコミュニケータ使って直接よべばいいけど、プロセス数とか自分の番号とか総和とか放送とかは良く使うので関数準備してあるの。例えば総和は

```
template <class T>
T PS::Comm::getSum(const T val);
```

ね。C++ のテンプレート使ってるから、MPI が知っている型なら MPI_Allreduce みたいに 6 個も引数並べなくてもよしなにやってくれるみたい。もちろん中では MPI_Allreduce 呼んでるだけ。

学生 C : 配列の各要素の和とかはどうすればいいんですか？

赤木 : そこですごく遅くなるとかでなければ要素毎に getSum 呼んでもいいし、性能が気になるなら直接 MPI_Allreduce 呼んでもいいし、なんなら配列版の getSum 作ってプルリクだしてくれれば、、、

学生 C : あー、そういうことですね。わかりました、、、

22.5 課題

1. 適当な 1 変数関数の台形公式による数値積分を区間分割して並列に行なう MPI プログラムを作って、ちゃんと動いていること、台形公式の刻みが十分小さいならちゃんと並列化で高速になることを確認してみよ。
2. 本文にあった、2 プロセスがメッセージ交換するプログラムをいくつか作ってみて、メッセージサイズが非常に大きい時の動作を確認せよ。

22.6 まとめ

1. MPI の概要を学んだ
2. MPI では、それぞれのプロセスは普通のプログラムで、それが MPI の関数を使って通信する。
3. 通信の基本的関数は Send/Recv (Isend/Irecv) である。
4. バリア同期、総和、放送のための関数もある。
5. FDPS から直接これらの関数を呼べる。また、同期、総和、放送は便利な関数が用意されている。

22.7 参考資料

特になし。