Ruby 初級者向けレッスン第 6 回

かずひこ@株式会社 ネットワーク応用通信研究所

2006 年 3 月 11 日

わかりやすい Ruby のコードを書こう

- 文字コードの問題
- 環境依存を避ける
- 変数名
- 空白とインデント
- ループの読みやすさ
- DRY (Don't Repeat Yourself)
- コーディング規約

今日のお題（小波ゼミの課題より）

- 運動方程式を 2 次の Leap-Frog 法で解く

運動方程式

- a（加速度）, v（速度）, x（位置）, t（時刻）とすると、以下が成立する
  - a = dv / dt
  - v = dx / dt

Leap-Frog 法（2 次）

- 「蛙跳び」という意味の近似法
- 最初の半歩先の値は Euler 法で求める
  - x1 = x0 + f(x0, t0) * dt
- 現在の値と半歩先の値から一歩先の値を求める
  - x2 = x0 + 2 * f(x1, t1) * dt
- 半歩先の値と一歩先の値から一歩半先の値を求める
  - x3 = x1 + 2 * f(x2, t2) * dt
- 同様に繰り返す

課題のパラメータ

- a = -9.8（重力加速度）
- v_start = 9.8（初速度）
- x_start = 0（初期位置）
- t_start = 0（開始時刻）
- t_end = 2.2（終了時刻）
- dt = 0.01（半歩分の時間）

変更前のコード

- fall-lf.rb

```
#!/usr/local/bin/ruby
#fall-lf.rb
a      = -9.8
vstart =  9.8
xstart =  0.0
tstart =  0.0
tend   =  2.2
dt     =  0.01

t = tstart
v0 = vstart
x0 = xstart
printf("%12.9f %12.9f\n",t,x0)#最初のゼロの状態表示
dt2 = dt*2#半歩から一歩作成
v1 = v0 + a*dt
x1 = x0 + v1*dt
imax = ((tend - tstart)/dt2).to_i
for i in 1 .. imax
  t = tstart + i*dt2 #表示させるときは一歩ずつ
  x2 = x0 + 2*v1*dt #計算は半歩ずつ
```

```
  v2 = v1 + a*dt #同上
  printf("%12.9f %12.9f\n",t,x2)#一歩先を表示
  x3 = x1 + 2*v2*dt#出力の更に半歩先を求めて、次の出力にそなえる
  v3 = v2 + a*dt#同上
  x0 = x2
  v1 = v3
  x1 = x3
  v2 = v3 + a*dt #半歩先のｖを求めて入れ替える。上では求められていないから。
end
```

ソースの文字コード

- 答案の文字コードは ISO-2022-JP（いわゆる JIS コード）だった
- ソースの中に全角文字（文字列リテラルなど）がある場合、Ruby インタプリタがソースを読み込む前に'-K' で指定する必要がある
- 特に Shift-JIS は、全角文字の 2 バイト目に'\' などの Ruby の制御文字が表れることがあるので、'-Ks' をつけて ruby を起動しないとはまる
- ISO-2022-JP は'-K' の選択肢にないので、文字単位で正規表現で扱えない
- というわけで、EUC-JP か UTF-8 がお薦め

Ruby インタプリタの起動

- 1 行目

```
#!/usr/local/bin/ruby
```

- /usr/local/bin/ruby のように書くと環境依存になる
- ちなみに、この起動方法を指定する 1 行目の書き方を shebang という
- 直接 ruby のパスを書かずに/usr/bin/env 経由で起動する

```
#!/usr/local/bin/ruby
↓
#!/usr/bin/env ruby
```

- env は「コマンドラインで指定したように環境を変更して、引数で指定したプログラムを実行する」コマンド
- とはいえ、今回のようなプログラムなら'ruby filename.rb' のように起動すればいいのであまり気にしなくてもいいかも

変数名

- 4 行目〜

```
vstart =  9.8
xstart =  0.0
tstart =  0.0
tend   =  2.2
```

- 'tend' のように単語として実在する変数名は意味を理解しにくい
- 'tend' →'t_end' のように、適宜'_' を挿入する方が意味を理解しやすい

```
vstart =  9.8
xstart =  0.0
tstart =  0.0
tend   =  2.2
↓
v_start =  9.8
x_start =  0.0
t_start =  0.0
t_end   =  2.2
```

空白とインデント

- 13 行目〜

```
printf("%12.9f %12.9f\n",t,x0)#最初のゼロの状態表示
dt2 = dt*2#半歩から一歩作成
v1 = v0 + a*dt
x1 = x0 + v1*dt
```

- 演算子やコメントの回りに空白がないと読みにくい
- 適宜、空白を入れる

```
printf("%12.9f %12.9f\n",t,x0)#最初のゼロの状態表示
dt2 = dt*2#半歩から一歩作成
v1 = v0 + a*dt
x1 = x0 + v1*dt
↓
printf("%12.9f %12.9f\n", t, x0) # 最初のゼロの状態表示
dt2 = dt * 2                     # 半歩から一歩作成
v1 = v0 + a * dt
x1 = x0 + v1 * dt
```

不要な代入の削除

- 18 行目〜（抜粋）

```
for i in 1 .. imax
  v2 = v1 + a * dt # 同上
  v3 = v2 + a * dt # 同上
  ...
  v2 = v3 + a * dt # 半歩先の v を求めて入れ替える。上では求められていないから。
end
```

- ループの上で v2 に代入しているから末尾の v2 の代入は不要
- 18 行目〜（抜粋）

```
x1 = x0 + v1*dt
...
for i in 1 .. imax
  printf("%12.9f %12.9f\n", t, x2) # 一歩先を表示
  x3 = x1 + 2 * v2 * dt # 出力の更に半歩先を求めて、次の出力にそなえる
  x1 = x3
  ...
end
```

- よく見ると x1 と x3 は他で使われていないから不要？

ループの読みやすさ

- 17 行目〜

```
imax = ((t_end - t_start) / dt2).to_i
for i in 1 .. imax
  t = t_start + i * dt2 #表示させるときは一歩ずつ
```

- 「t_start から t_end まで dt2 ずつ t を進めていく」という意図がわかりにくい
- while で書く方が読みやすい？

```
while t <= t_end
  ...
  t += dt2 # 時刻を一歩先に進める
end
```

- ただし、Float を足していくと誤差がありえるので注意


```
a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1  + 0.1 + 0.1 #=> 0.8
a == 0.8 #=> false
a - 0.8 #=> -1.11022302462516e-16
```

- 1/2、1/4、1/8... のように「誤差なしで表現できる」Float なら誤差は出ない

```
a = 0.25 + 0.25 + 0.25 + 0.25 + 0.25 + 0.25  + 0.25 + 0.25 #=> 2.0
a == 2.0 #=> true
```

- Float#step を使えば、誤差は大丈夫

```
t_start.step(t_end, dt2) do |t|
  ...
end
```

DRY (Don't Repeat Yourself)

- 13 行目〜（抜粋）

```
printf("%12.9f %12.9f\n", t, x0) # 最初のゼロの状態表示
v1 = v0 + a * dt
for i in 1 .. imax
  v2 = v1 + a * dt
  printf("%12.9f %12.9f\n", t, x2) # 一歩先を表示
  v3 = v2 + a * dt
  v1 = v3
end
```

- 似た printf がループ内外にある
- 出力はすべてループの中で処理する

```
v1 = v0 + a * dt
# t_end に達するまで dt2 間隔で繰り返す
t_start.step(t_end, dt2) do |t|
  printf("%12.9f %12.9f\n", t, x0)
  x2 = x0 + 2 * v1 * dt
  v2 = v1 + a * dt
  v3 = v2 + a * dt
  v0 = v2
  v1 = v3
  x0 = x2
end
```

コメントなど

- 関連する処理や代入をまとめて、必要に応じてコメントを書く

```
# パラメータの設定
a       = -9.8
v_start =  9.8
x_start =  0.0
t_start =  0.0
t_end   =  2.2
dt      =  0.01   # 半歩の長さ
dt2     =  dt * 2 # 一歩の長さ
```

- 関連する処理や代入をまとめて、必要に応じてコメントを書く

```
# 最初の状態
t  = t_start
v0 = v_start
x0 = x_start
```

ここまでのコード

- fall-lf.rb

```
#!/usr/bin/env ruby

# パラメータの設定
a       = -9.8
v_start =  9.8
x_start =  0.0
t_start =  0.0
t_end   =  2.2
dt      =  0.01   # 半歩の長さ
dt2     =  dt * 2 # 一歩の長さ

# 最初の状態
t  = t_start
v0 = v_start
x0 = x_start

# Euler 法で半歩先の状態を計算する
v1 = v0 + a * dt
```



```
# t_end に達するまで dt2 間隔で繰り返す
t_start.step(t_end, dt2) do |t|
  printf("%12.9f %12.9f\n", t, x0)
  x2 = x0 + 2 * v1 * dt # 現在の位置と半歩先の速度から一歩先の位置を計算する
  v2 = v1 + a * dt      # 半歩先の速度と加速度から一歩先の速度を計算する
  v3 = v2 + a * dt      # 一歩先の速度と加速度から一歩半先の速度を計算する
  v0 = v2               # 次の v0 を代入
  v1 = v3               # 次の v1 を代入
  x0 = x2               # 次の x0 を代入
end
```

ちょっと待って...

- Leap-Frog 法は「現在の値と半歩先の微分係数から一歩先の値を求める」だよね？
- だったら、v1 から v2 を求めるのって反則じゃない？
- 半歩ずつ時刻を進めるなら一歩半先を求めなくていいよね？

実装の見直し

- ちゃんと Leap-Frog 法で実装しなおす

```
v1 = v0 + a * dt
x1 = x0 + v0 * dt
# t_end に達するまで dt 間隔で繰り返す
t_start.step(t_end, dt) do |t|
  printf("%12.9f %12.9f\n", t, x0)
  v2 = v0 + 2 * a * dt
  x2 = x0 + 2 * v1 * dt
  v0 = v1
  x0 = x1
  v1 = v2
  x1 = x2
end
```

変更後のコード

- fall-lf.rb

```
#!/usr/bin/env ruby

# パラメータの設定
```

```
a       = -9.8
v_start =  9.8
x_start =  0.0
t_start =  0.0
t_end   =  2.2
dt      =  0.01   # 半歩の長さ

# 最初の状態
t  = t_start
v0 = v_start
x0 = x_start

# Euler 法で最初の半歩先の状態を計算する
v1 = v0 + a * dt  # 半歩先の速度
x1 = x0 + v0 * dt # 半歩先の位置

# t_end に達するまで dt 間隔で繰り返す
t_start.step(t_end, dt) do |t|
  printf("%12.9f %12.9f\n", t, x0)
  v2 = v0 + 2 * a * dt  # 現在の速度と半歩先の加速度から一歩先の速度を計算する
  x2 = x0 + 2 * v1 * dt # 現在の位置と半歩先の速度から一歩先の位置を計算する
  v0 = v1               # 次の v0 を代入
  x0 = x1               # 次の x0 を代入
  v1 = v2               # 次の v1 を代入
  x1 = x2               # 次の x1 を代入
end
```

コーディング規約

- コーディング標準は XP（eXtreme Programming）のプラクティスの一つ
- 一つの規約に従ってコードを書くことで、コード共有を助ける
- Ruby のコーディング規約は前田さんのが有名
  - http://shugo.net/ruby-codeconv/codeconv.rd

まとめ

- 「一ヶ月後の自分」は他人、他人にもわかるコードを書こう
- 「よりよい書き方」を目指すことで、自然とコーディング能力が上達する
- 「変だな？」と思ったら落ち着いて考え直そう

参考文献

『Rubyist Magazine - あなたの Ruby コードを添削します』
http://jp.rubyist.net/magazine/?0010-CodeReview http://jp.rubyist.net/magazine/?0011-CodeReview http://jp.rubyist.net/magazine/?0013-CodeReview

『小波ゼミ - 変化を追う：進化，物体の運動，状態の変化』
http://www.cs.kyoto-wu.ac.jp/konami/Semi-34/evolution/

『Ruby コーディング規約』
http://shugo.net/ruby-codeconv/codeconv.html

今後の情報源

公式 Web サイト
http://www.ruby-lang.org/

リファレンスマニュアル
http://www.ruby-lang.org/ja/man/

日本 Ruby の会
http://jp.rubyist.net/

Rubyist Magazine
http://jp.rubyist.net/magazine/

ふぇみにん日記
http://kazuhiko.tdiary.net/

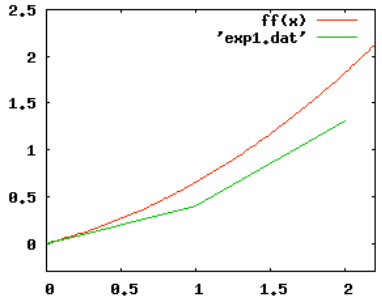
このページを正しく読むためには MathML の表示のための数式フォントが必要です． Firefox, Netscape (6.0以降)など Moziila 系ブラウザであれば，次の手順に従ってフォントをインストールしてみてください． → Firefox のインストールとその他の設定の手順
なお，IE の場合には MathPlayer というのも組み込まないとハングするようです．

Euler 法を改良する — Leap-Flog 法

ここではまず Euler 法の問題点を調べて，それを改善するためのより高い近似法である Leap-Frog 法を使ってみます．その後，物体の落下の問題を取り扱います．これらは力学的なシミュレーションの基礎となるものです．

Euler 法の問題点

Euler 法の区間の刻みを粗くしてみる



上の図を見てください．赤はある微分方程式を解析的に正確に解いて得られた曲線です．その下側にある折れ線は，非常に大きな間隔を使って Euler 法で同じ微分方程式の解を求めたものです．正しい曲線に比べて，Euler 法では解が小さな値をとってしまっていることが分ります．その原因はどこにあるのでしょうか．

グラフの左下は，計算の出発点です．本来の解と，Euler 法による解とは，出発点での向きは一致しています．Euler 法による解は，そのまま直線で延びているのに，正しい解の法は，そこから上向きにカーブしています．この違いが問題だということは，見ればわかります．つまり，Euler 法は，

- 傾きを使って次の点を予測する
- カーブによるずれは計算に入れられない

という性質の近似なのです．傾きは1次微分で曲がりは2次微分ですから，つまり1次微分だけを考えるというやり方ともいえます．

Euler 法の詳しい検討

下の図を使って，そのことを詳しく説明してみましょう．解かれるべき微分方程式は

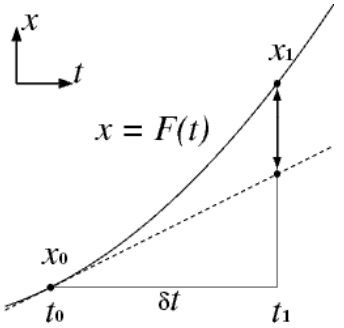
$$\frac{dx}{dt} = f(x, t)$$

です．これを解いて得られる正確な解は

$$x = F(t)$$

であるとします．

このグラフを見るとき，横軸は t，縦軸はx になっていることに注意してください(グラフはいつでも横軸が x，縦軸が y だと思わないように！)． t は時刻， x は位置ということにして説明します．



いま，時刻t_0 で位置が x_0 にあったとしましょう．この点でのグラフの傾きは，定義によって $f(x_0, t_0)$ です．図の点線の傾きがちょうどそれになっていることは見ればわかるでしょう．この時点からある短い時間 δt が過ぎた時刻 t_1 での位置を予測したいのです．その計算方法は簡単で，

上昇の幅＝傾き×水平のずれ

なのですから，

$$f(x_0, t_0)\delta t$$

がグラフの上昇分に相当します．これを元の位置 x_0 に加えてやれば， t_1 での位置が得られます．つまり，

$$x_1 = x_0 + f(x_0, t_0)\delta t$$

として，次の位置 x_1 を求めます． これが Euler 法の考え方です．

ところが，こんなふうにして得られた点は，本当の解の曲線上には乗ることができず，ずれが生じてしまいます．図の矢印は真の解における位置 $x_1 = F(t_1)$ と，上の計算で得られた答とのずれを示しています．このように，Euler 法は解となる曲線が曲がっていることを考慮しない計算法なので，そのことによるずれは避けられません．

もっともこれでも間隔を十分に細かく取って計算してやれば，正確な解にいくらでも近い結果を得ること

が，原理的にはできるはずです．なぜなら，自然は瞬間瞬間の傾きから，その次の無限に近い点を決めて動いているといってもよいのですから．

ところが，このような差分法を非常に細かい区切りで実行することには，次のように2つの問題点があります．

- 回数が莫大になってしまって，計算時間が非常に長くなる．
- 膨大な数を足しあわせていくことになり，誤差の蓄積によって計算精度が落ちてしまう．

そこで，曲線のカーブの部分に目をつけた微分方程式の数値的解法がいくつも提案されています．次に紹介する Leap-Frog 法もそのひとつで，最近よく使われるようになっています．なお，微分方程式の数値的解法としては Runge-Kutta 法も非常によく使われていますが，ここでは取り上げません．

Leap-Frog 法

Leap は「跳ぶ」，Frog はもちろん「蛙」です．つまり「蛙跳び法」という名前の方法です．どうしてこんな奇妙な名前が付いているのかは，その仕組みをみると分ります．

Euler 法の原理を微分形で表す

時間 t とともに変化する量 x が次の微分方程式に従うものとします．

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

Euler 法ではこれを素朴に差分化したわけですが，ここではちょっとした仕掛けを設けて，別の形の差分を導きます．

まずtの 関数$F(t)$ の微分の定義というのは，通常次のように表されます．

$$\frac{dF(t)}{dt} = \lim_{h \to 0} \frac{F(t+h) - F(t)}{h}$$

Euler 法というのは，この式で無限小として扱われる h を有限の大きさδt として，次のように差分の形で扱ったものです．

$$\frac{F(t+\delta t) - F(t)}{\delta t} = f(x)$$

この扱いの意味とその問題点については，すでに述べました．

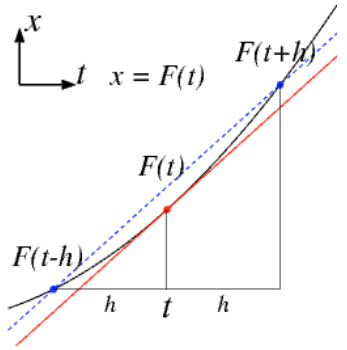
Leap-Frog 法の原理

さて，ここで微分の定義を次のようにひとひねりして書いてみます．

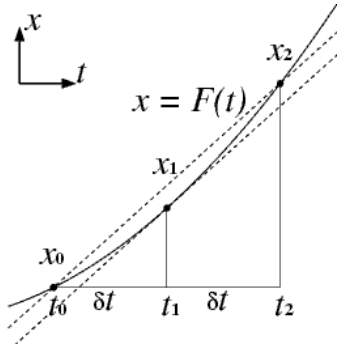
$$\frac{dF(t)}{dt} = \lim_{h \to 0} \frac{F(t+h) - F(t-h)}{2h}$$

本来の微分の定義における幅 h を，接線の傾きを求めたい点の前後にとって，それに合わせて増分 h も $2h$ にするわけですから，これでも構わないわけです．下の図を見てください．上の式による微分の定義は，2つの青い点を，h を小さくすることによって，中央の赤い点に近づけて行ったときに得られる赤い

線が，t におけるこの関数の傾き（微分）であることが，図から理解できるでしょう．



Leap-Frog 法は上記のような微分形を差分化したものと考えることができます．hを有限な刻み δt に置き換え，さらに $t - h \to t_0$, $t \to t_1$, $t + h \to t_2$, $F(t-h) \to x_0$, $F(t) \to x_1$, $F(t+h) \to x_2$ という置き換えを行ったのが下の図です．



これを使った差分方程式の形は次のようになります．

$$\frac{x_2 - x_0}{2\delta t} = f(t_1)$$

これを変形すると，次の式が得られます．これが Leap-Frog 法の基礎となる式です．

$$x_2 = x_0 + 2f(x_1, t_1)\delta t$$

この式の意味は次のように解釈できます．上の図を見てください．

$t = t_1$ における接線の傾きは $f(t_1)$ です．この接線を上にずらしてやって，ちょうど x_0の点を通るようにしてやります．その点から $2\delta t$ だけ進んだところでの上昇分 $2f(t_1)\delta t$ を x_0 に加えると，x_2 の値に近い値が得られるはずです．

このやり方では，$t = t_0$ からの増加分を求めるのに，その時点での微分係数 $f(t_0)$ を用いずに，目的とする点までのちょうど半分のところでの微分係数を使っています．ですから，曲線の曲がりの効果も計算に含まれることになり，計算の精度は改善されることになります． Leap-Frog 法の精度についての説明は下に示してありますから，参考にしてください．

Leap-Frog 法は初期値のとり方に工夫がいる

実際に Leap-Frog 法をプログラムに組み込もうとすると， Euler 法にはなかった問題点にぶつかります．それは，初期値が1つではすまないということです．つまり，Euler 法では最初の時刻 $t = t_0$ での $x = x_0$ の値さえ分れば，それらを $f(x, t)$ に代入して $f(x_0, t_0)$ を得て，それから傾きを計算することができました(お湯の冷却問題では， $f(x, t)$ は t には依存しないので，単に $f(x)$の形になっています)．ところが Leap-Frog 法では，次の点の値を予測するために必要な傾きは δt だけ進んだ時点 $t = t_1$ での傾きなのです．その瞬間のx の値は $x_1 = F(t_1)$ なのですから，そもそも $F(t)$ を求めようというのに，先にその解を知らないといけないという自己矛盾が発生してしまいます．

上記の難点を切り抜けるためには，とりあえず Euler 法のやり方で x_1 を求めてしまおうというやり方を取ることにします．つまり，$t = t_0$ の時の傾き $f(x_0, t_0)$ を使って，次のように x_1 を求めるのです．

$$x_1 = x_0 + f(x_0, t_0)\delta t$$

これはすでに出てきた Euler 法のやり方そのものです．そのために生じる誤差は我慢するしかありませんが，こうやって得た x_1 さえあれば，その後は順次計算を進めることが可能です．それでは実装のための段取りを考えましょう．

Leap-Frog 法を実装する

Leap-Frog 法を実装するために，その手続きを書き並べてみます．

- 最初に与えられているもの： 初期値 t_0, x_0，求めるべき関数 $F(t)$ の変化率を規定する関数 $f(x, t)$
- 決めておくべき定数： 時間刻み δt と計算すべき t の範囲．ただし，計算すべき範囲と，それをいくつに分割するかという，区間の数を決めておいてもよい．その場合，δt は割り算で決めることになる．
- $x_1 = x_0 + f(x_0, t_0)\delta t$ を使って x_1 を求める．
- x_0, x_1 を $x_2 = x_0 + 2f(x_1, t_1)\delta t$ に代入して， x_2 を得る．
- x_1, x_2 を $x_3 = x_1 + 2f(x_2, t_2)\delta t$ に代入して， x_3 を得る．
- 以下必要な回数だけ繰り返す．

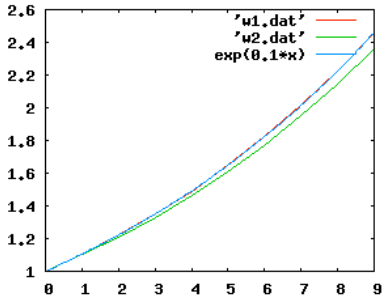
以上を Ruby で実装したソースを下に示します． for ループの中での時刻のとり方や，ある変数から計算によって得られた値を再び前の変数に代入して計算を繰り返しているところなどに注意しましょう．

```
#!/usr/local/bin/ruby
def f(x)       # f(x,t), ただし, この場合には f は t に依存しない.
  return 0.1 * x
end
tstart = 0.0   # 始まりの時刻
tend = 10.0    # 終了時刻
xinit = 1.0    # x の初期値
nstep = 10     # 分割区間の数
dt = (tend-tstart)/nstep
dth = dt/2     # dx の半分
t = tstart
x0 = xinit
```



```
x1 = xinit + dth * f(x0)    # 最初に x1 を求めておく.
for i in 1 .. nstep
   printf("%12.9f %12.9f¥n",t,x0)
   printf("%12.9f %12.9f¥n",t+dth,x1)
  x2 = x0 + dt * f(x1)
  x3 = x1 + dt * f(x2)
  t = tstart + i * dt
  x0 = x2
  x1 = x3
end
```

このソースを使って実際に計算した結果を下に示します．赤が Leap-Frog 法による結果，青は解析的な厳密解，緑は Euler 法による結果です．Euler 法に比べて格段に精度が向上していることが分ります．



Leap-Frog 法の精度

この図で，(t_0, x_0) と(t_2, x_2) の2点を結んだ点線が，もしも(t_1, x_1) における$x = F(t)$ の接線と平行であるなら， Leap-Frog 法の近似は厳密に正しいことになるはずです．なぜならそのとき，
(t_0, x_0) を通って，この接線と平行な直線は， (t_2, x_2) を通ることになるから．
この条件が成立するのはどういうときでしょうか．それは$F(t)$ が t の2次までの項しか含まない多項式で表されるとき，つまり
$F(t) = at^2 + bt + c$
で表されるときということになります(この式を上の記述に従っていじってみれば，簡単に証明できます)．逆に3次以上の高次の項を含んでいるときには，ここで使っている Leap-Frog 法では誤差を生じます．もちろんそれでも Euler 法よりはずっとよい精度の結果が得られるのですが，もっとよい結果をほしいときには，さらに次数の高い項を含められるように改良された Leap-Frog 法を使うことになります．

元に戻る

小波秀雄，2005/10/19