常微分方程式 (Ordinary Differential Equation) の数値解法

森下 博和*

平成 20 年 11 月 17 日

1 はじめに

上微分方程式は変数をうまく用い入ることで 1 階微分方程式にすることで解ける。
ちなみにうまく用いないと不連続な振舞をするかも。
常微分方程式を解くための問題として境界条件の性質がありそれは以下の 2 つとなる。

- 初期値問題
  - 式中の全ての初期値を与えておくことで、最終状態 x_f あるいはその値の変化をある間隔で観測することを目的とする
- 2 点境界問題
  - 2 点以上の初期値が存在。一般には x_s, x_f が与えられる。

本章では初期値問題を扱う。この問題を特には

- Runge-Kutta method
  - 基本的には何でも解けるが高速とは限らない
- Richardson extrapolation, 特別なステップと有理関数補外により Bulirsch-Stoer method
  - 予測子修正子法より良い事が多い。
- predictor-corrector method
  - 値を戻しながら実行するが、Runge-Kutta より効率は良い。洗練しないと Bulirsch-Stoer method に勝てない

これらを構成するものは 3 層からなる。

- アルゴリズムルーチン
  - 新しい計算点の算出
- ステッパルーチン
  - アルゴリズムルーチンに適切な刻み幅を渡し、精度を保証

*morisita@am.ics.keio.ac.jp

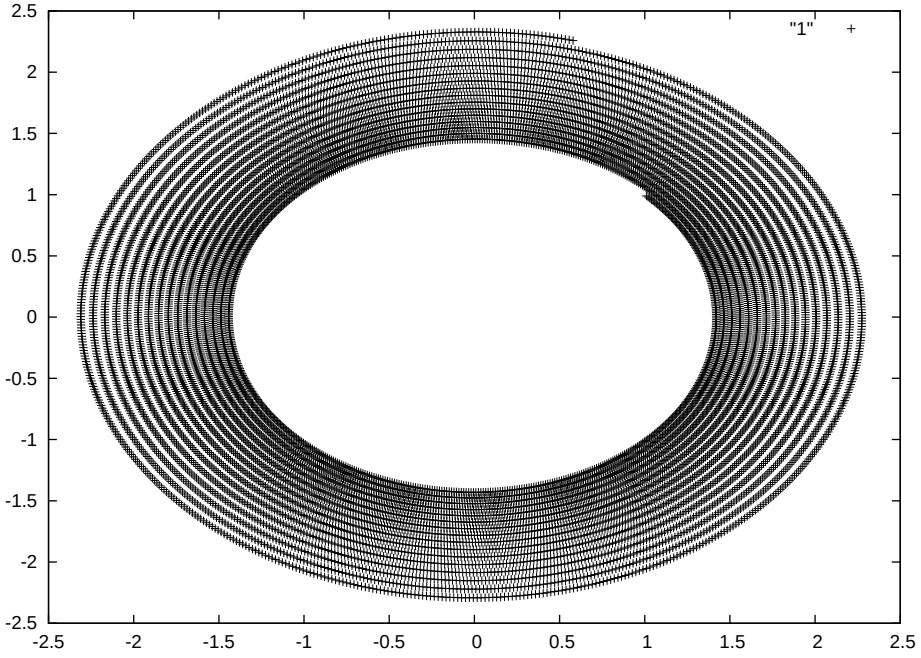

図 1: Euler 法

- ドライバルーチン
  - 利用者とのインタフェース、計算開始終了など。

2 Runge-Kutta method

Euler 法は出発点を与えられた後次の点を補外することで進展する。こいつの弱点は

- 同じ刻み幅でも他よりへぼい
- 安定でない

まずここで単振動の実験をしてみる。

$$m\frac{d^2x}{dy^2} = -kx$$

これを一階の微分方程式にするには

$$\frac{dv}{dt} = -x$$

$$\frac{dx}{dt} = v$$

とおけばよい。
$m = 1$、$k = 1$ とすると次のような結果になる。

そこで次の点の前にその間の仮ステップを挿入することで 2 次ルンゲクッタ法となる (ここではそのうちの 1 つ中点法を紹介)。
この" 仮のステップ" を計算するのにテイラー展開を行う。a の近傍に置けるテイラー展開は

$$f(x) = f(a) + f'(a)(x-a) + \frac{f^{(2)}(a)}{2}(x-a)^2 + \frac{f^{(3)}(a)}{6}(x-a)^3 + \frac{f^{(4)}(a)}{24}(x-a)^4 + ...$$

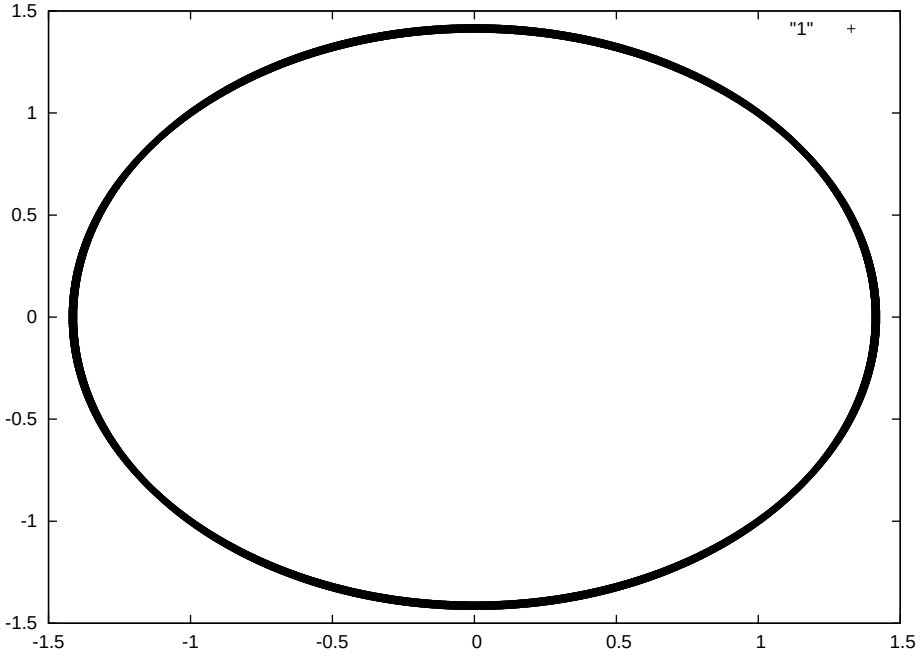

図 2: 中点法

と示せる。これを 2 次まで考慮すると

$$f_n(x) = f(x+h) = f(a) + f'(a)h + \frac{f^{(2)}(a)}{2}h^2(\text{ただし } f(a) \text{ は既知})$$

$$= f(a) + h(Af'(a) + Bf'(a+h)h)(\text{ラグランジュの平均値の定理を使っている})$$

ここでこれを満たす A,B,h の値を任意に定めることができる。そのうちの一つが中点法で、$A = 0, B = 1, h = \frac{h}{2}$ とすれば,

$$k_1 = hf'(x_n, y_n)$$

$$k_2 = hf'(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$

$$y_{n+1} = y_n + k_2 + O(h^3)$$

が得られる。この計算結果は次の通り。特によく使われるのが式の 4 次ルンゲクッタ法。定義については長くなるそうなのでとりあえず以下を使うべし。

$$k_1 = hf'(x_n, y_n)$$

$$k_2 = hf'(x_x + \frac{h}{2}, y_n + \frac{k_1}{2})$$

$$k_3 = hf'(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$

$$k_4 = hf'(x_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)$$

次はルンゲクッタの計算例。でも高精度でいうと、Bulirsch-Stoer 法や予測子修正子法の方が役立ち、ルンゲクッタは駄馬だ（原文より)。

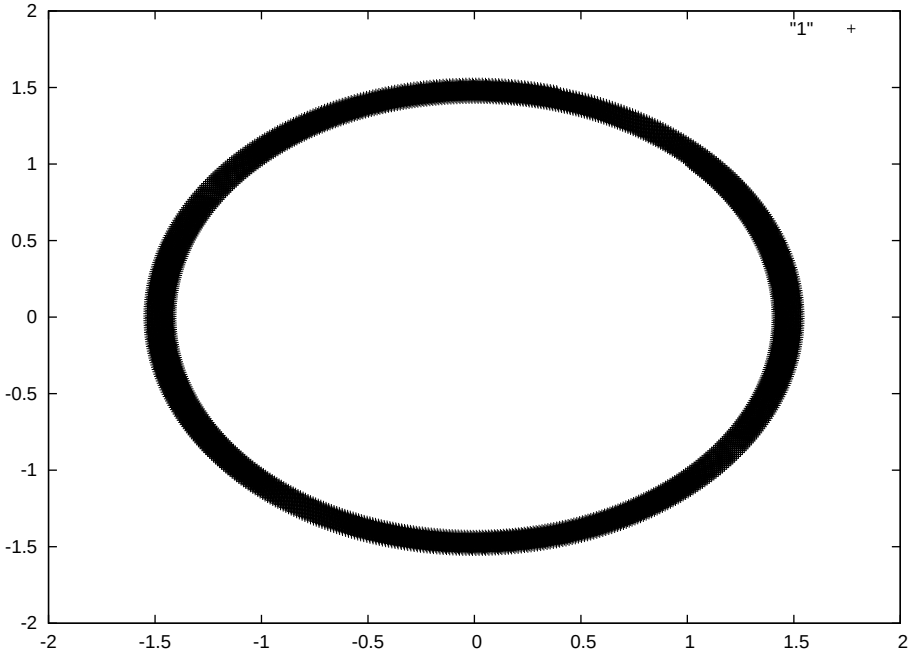

図 3: ルンゲクッタ法

本文のソースは次のようになっている。次の結果は導関数を定義していないと駄目なのだが、そこはユーザー定義である。なぜ計算しない場合があるかというと、刻み幅が大きすぎたときのやりなおしの可能性があるからだ。これを適応刻み幅制御 (adaptive stepsize control) という。次はドライバを示す。ドライバにより初期値、関数、ステップ数の制御、結果の表示などが可能となる。

3 Runge-Kuttaに対する適応刻み幅制御

この方法を使うと、効率は 10〜100 倍に効くらしい。精度の検証で逆に計算が遅くなりそうだが、投資の価値は十分にある。全体の 4 次ルンゲクッタのステップ幅を半分にすれば、計算回数は合計 11(スタートが一つは一緒) でこの半分の刻み幅と比較すればたったの 1.375 倍労力で、精度は同じ。
計算をして所望の精度が得られたら次に進み、駄目なら細かくするというもの。実装の際には、細かすぎる精度を要請すると、マシンイプシロンを下回り、計算が先に進まなくなることがあるので注意。それに対する警告を出力すればいいが、すると計算機から離れなくなり結局自動化に向かず。

4 修正中点法

時間ステップ $h = H/n$ として h ステップで進む以下の式の方法。

$$z_0 \equiv y(x)$$

$$z_1 = z_0 + hf'(x, z_0)$$

$$z_{m+1} = z_{m-1} + 2f'(x + H, z_n)$$

$$y(x + H) \approx y_n \equiv \frac{1}{2}(z_n + z_{n-1} + hf'(x + H, z_n))$$

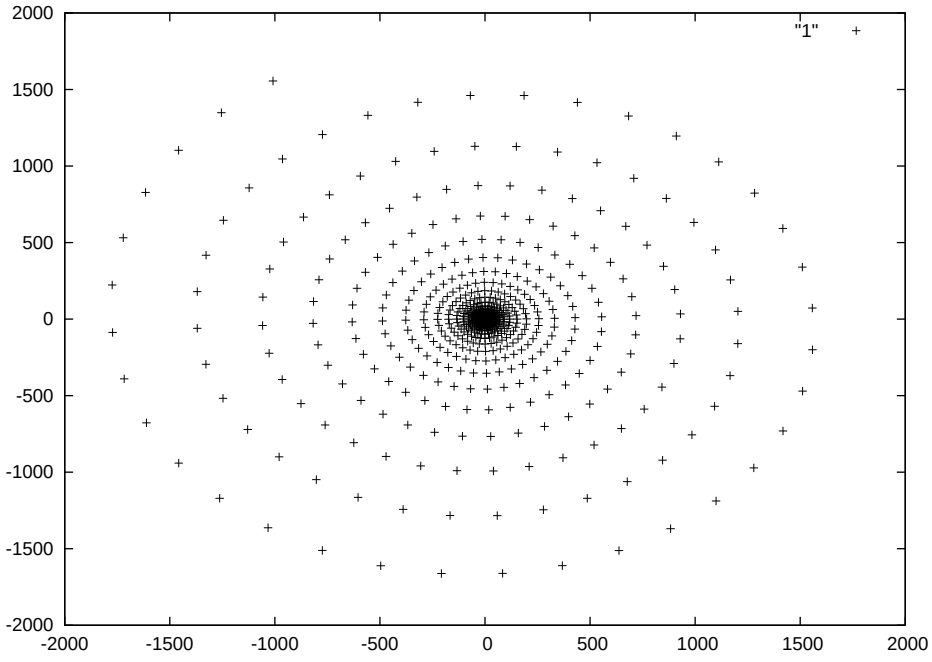

図 4: 修正中点法

2 次ルンゲクッタ法 (中点法) よりも計算量が少ないが、適応刻み幅制御を施した 4 次ルンゲクッタには及ばない。この方法は Bulirsch-Stoer 法に採用されるため紹介している。これを n-¿n/2 にすると 4 次精度の計算量の少ない Rungekutta 法となるが、まぁ慌てるな。まずはこれでも見て落ち着こう。

5 Richardson 補外と Bulirsch-Stoer 法

これらの方法は

- なめらかな関数であること
- 特異点がないこと

において有効でルンゲクッタよりは高速だが、例えば値を補間しながら計算をすすめるのであればあまりおすすめできない方法だ。とはいえ、これにあてはまらなければ、高精度高速なものだぜ。
これの基本的な三つの考えは以下の通り。

- Richardson の極限遅延接近
  - 最終的な補外の値を細かな刻み幅 h で得られた場合、その補外値を答えとして扱う
- 有理関数補外
  - 有理関数がうまく補外できる可能性が高い事
- 有理関数補外を h ではなく h^2 で行うこと

そして、計算には修正中点法、有理関数補外を行う。これがうまくいかないのは、先の条件の他に、有理関数が極を持ち、0 で除算される場合だが、そのためのチェックはプログラム内で気をつければよい。

他にも刻み幅の設定が下手だとつまづきかねない。
具体的には大きな区間 H に対してサブステップ n を用意して

$$n = 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, ...$$

のように計算を行っていく。このステップの変更に伴い新たな補外値と推定誤差を求める。この方法は最適とは言えないが、何が最適なのかは分かっていないらしい。ただし、それでもうまくいかない場合は、以下のような変更、

- 部分的にルンゲクッタ法を使う方法
- 有理関数補外が駄目ならそこだけ多項式補外を用いる

が考えられる。巧く使えばこの方法が最高精度、最高速度だと考えられている。

6　予測子・修正子法

使いやすさの適応幅刻み制御のルンゲクッタ、精度速度の Bulirsch-Stoer、その中間的性質を持つのがこいつ。歴史が長いため古くからのプログラムにはよくある。各点のプロファイルをとっておき、多項式補外によって次のステップを得る（予測子ステップ)。なんと右辺を計算しない。
積分では独立変数 x により y が一意に決まるのに対して、偏微分ではそうもいかないが、Simpson 方の積分を行い、右辺を計算するのが修正子ステップで、積分程度のオーダーなら問題なし。
有名な方法として以下の式の Adams-Bashforth-Moulton(アダムス・バッシュフォース・モールトン) 法が挙げられる。

$$\text{予測子 } P : y_1 = y_0 + \frac{h}{24}(55y'_0 - 59y'_1 + 37y'_2 - 9y'_3)$$

導関数 y を求める:E

$$\text{修正子 } E : y_1 = y_0 + \frac{h}{24}(9y'_1 + 19y'_0 - 5y'_1 + y'_2)$$

このステップを $P(EC)^m$ 続け、最後は E で終えるのが良いようだ。弱点としては、初期計算に何点か計算を行わなければいけない、幅や精度を適応的に変えなければ他の方法より優れた結果とならないなどがある。

7　硬い連立方程式

本章で扱った数値解法では、誤差の影響により計算が合わなくなる硬い方程式に出くわし兼ねない。“硬さ” とは、ある時間刻みが発散しない範囲でとりうる最大値のことである。

$$s = \frac{max|R\lambda|}{min|R\lambda|}$$

およそ 10^4 を越えると硬いと言われるが、これは実用上の問題である。

例えば、

$$y = Ae^{-10x} + Be^{10x}$$

$$y(0) = 1, y'(0) = -10$$

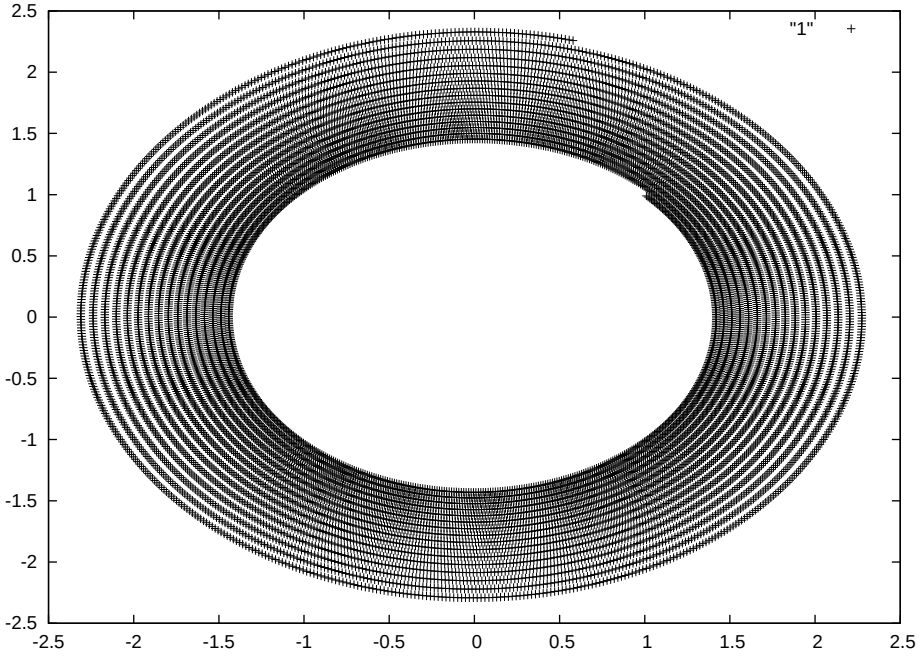

図 5: 後退オイラー法

とすると、その解は

$$y = e^{-10x}$$

だが、これに誤差が入ると、B の値に何らかの値が入ることと等価となる。

1 つの方程式

$$y' = -cy(c > 0)$$

を解くのに、刻み幅 h、陽的（前進）Euler 法を用いると

$$y_{n+1} = y_n + hy'_n = (1 - ch)y_n$$

として解ける。ただしこれは時間ステップ $h <= 2/c$ を満たさなければならない。

一方新しい点から前の点を算出する陰的 (後退)Euler を用いると

$$y_{n+1} = y_n + hy'_{n+1}$$

から解ける。これは本質的に安定だが、各ステップ毎に逆行列を計算しなければならず、計算量が増える。

高次の陰解法を開発するのは大変で、現在も研究中だそうです。最後に陰的一時オイラーのプログラムを見てくださいませ。

8 付録

あんまり綺麗なソースではないので参考にはならないかもしれませんが。

単振動

$$m\frac{d^2x}{dt^2} = -kx(m = 1, k = 1)$$

を例題としています。

8.1 陽的オイラー法

```
import std.cstream;
import std.math;

int main(char[][] args){

   real x, y, v;
   real dt, t, tmax;
   real dvdt, dxdt;

   x = 1.0;
   v = 1.0;
   dt = 0.01;
   tmax = 100.0;

   for ( t =0.00; t < tmax; t+= dt){

      dvdt = dt*(-x);
      dxdt = dt*v;

      x += dxdt;
      v += dvdt;

      dout.writefln(x," ",v);

   }

   return 0;
}
```

8.2 陽的 2 次ルンゲクッタ (中点法)

```
import std.cstream;
import std.math;

int main(char[][] args){

   real x, y, v;
   real dt, t, tmax;
   real dvdtFirst, dxdtFirst;
   real dvdtCenter, dxdtCenter;

   x = 1.0;
```

```
   v = 1.0;
   dt = 0.01;
   tmax = 100.0;

   for ( t =0.0; t < tmax; t+= dt){

      dvdtFirst = dt*functionForCoodination( x, v);
      dxdtFirst = dt*functionForVelocity( x, v);

      dvdtCenter = dt*functionForCoodination( x+dxdtFirst/2, v+dvdtFirst/2);
      dxdtCenter = dt*functionForVelocity( x+dxdtFirst/2, v+dvdtFirst/2);

      x += dxdtCenter;
      v += dvdtCenter;

      dout.writefln(x," ",v);

   }

   return 0;
}

real functionForCoodination( real x, real v){
   return -x;
}

real functionForVelocity( real x, real v){
   return v;
}
```

8.3 陽的4次ルンゲクッタ法

```
import std.cstream;
import std.math;

int main(char[][] args){

   real x, y, v;
   real dt, t, tmax;
   real k1ForX, k1ForV;
   real k2ForX, k2ForV;
   real k3ForX, k3ForV;
   real k4ForX, k4ForV;
```

```
   x = 1.0;
   v = 1.0;
   dt = 0.01;
   tmax = 100.0;

   for ( t =0.0; t < tmax; t+= dt){

      k1ForX = dt*functionForCoodination( x, v);
      k1ForV = dt*functionForVelocity( x, v);

      k2ForX = dt*functionForCoodination( x+k1ForX/2, v+k1ForV/2);
      k2ForV = dt*functionForVelocity( x+k1ForX/2, v+k1ForV/2);

      k3ForX = dt*functionForCoodination( x+k2ForX/2, v+k2ForV/2);
      k3ForV = dt*functionForVelocity( x+k2ForX/2, v+k2ForV/2);

      k4ForX = dt*functionForCoodination( x+k3ForX/2, v+k3ForV/2);
      k4ForV = dt*functionForVelocity( x+k3ForX/2, v+k3ForV/2);

      x += k1ForX/6 + k2ForX/3 + k3ForX/3 + k4ForX/6;
      v += k1ForV/6 + k2ForV/3 + k3ForV/3 + k4ForV/6;

      dout.writefln(x," ",v);

   }

   return 0;
}

real functionForCoodination( real x, real v){
   return v;
}

real functionForVelocity( real x, real v){
   return -x;
}
```

8.4 陽的修正中点法

```
   real[10] zForX;
   real[10] zForV;
```

```
   int splitTheStep = 10; /+ Corresponding to "n" +/
   int i;

   x = 1.0;
   v = 1.0;
   H = 0.1;
   tmax = 100.0;
   dt = H / cast(real)splitTheStep; /+ TimeStep +/

   /+ Start at Time 0.0 +/
   for ( t=0.0; t < tmax; t+= H) {

      /+ First two step+/
      zForX[0] = x;
      zForV[0] = v;

      zForX[1] = zForX[0] + dt*functionForCoodination( x, v);
      zForV[1] = zForV[0] + dt*functionForVelocity( x, v);

      for ( i = 1; i < splitTheStep - 1; i ++){
         zForX[i+1]=zForX[i-1] +
                  2*dt*functionForCoodination( x+zForX[i], v+zForV[i]);

         zForV[i+1]=zForV[i-1] +
                  2*dt*functionForVelocity( x+zForX[i], v+zForV[i]);
      }

      /+ Last Step +/

      x = ( zForX[9] + zForX[8] + dt*functionForCoodination( x+zForX[9], v+zForV[9]))/2;
      v = ( zForV[9] + zForV[8] + dt*functionForVelocity( x+zForX[9], v+zForV[9]))/2;

      dout.writefln(x," ",v);
   }

   return 0;
}

real functionForCoodination( real x, real v){
   return v;
}

real functionForVelocity( real x, real v){
   return -x;
```

```
}
```

8.5 陰的オイラー法

```
import std.cstream;
import std.math;

int main(char[][] args){

   real x, y, v;
   real dt, t, tmax;
   real dvdt, dxdt;
   real preX, preV;

   x = 1.0;
   v = 1.0;
   dt = 0.01;
   tmax = 100.0;

   for ( t =0.00; t < tmax; t+= dt){
      preX = x;
      preV = v;

      x = preX + dt * preV / ( 1 + dt*dt);
      v = preV - dt * preX / ( 1 - dt*dt);

      dout.writefln(x," ",v);

   }

   return 0;
}
```