# エルゴノミクスコンピューティング実習
# 物理シミュレーション

井村 誠孝 (m.imura@kwansei.ac.jp)

2018 年 12 月 4 日

バーチャル世界のリアリティを高めるためには，妥当性のあるシミュレーションが必要である．本演習では，運動方程式に基づいて物体の挙動をシミュレーションする手法のうち，粘弾性体の表現に利用されるバネ-ダンパ-質点について学習する．

**■■■ 演習: この項目が演習** 各自，実施してください．発展と書かれている項目は，任意です．

本課題は 3 回にわたって実施されます．

1. 物理シミュレーションの基礎
2. クラスの設計と導入，インタラクション
3. バネ-ダンパ-質点モデルによる粘弾性体の表現

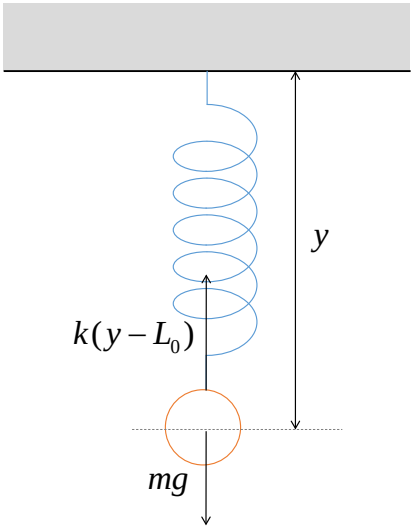


図 1　バネにより単振動する物体

# 1　単振動する物体

早速，最初の物理シミュレーションとして，バネによって定点に接続されている物体の単振動を扱ってみましょう．

## 1.1　単振動する物体の運動方程式

まず物体に働く力を書き出して，運動方程式を立てましょう．

簡単のため，物体は直線上を運動することにします．鉛直下向きを +y 方向とします．バネ定数 $k$，自然長 $L_0$ のバネの一端が固定されており，バネのもう片方に重さ $m$ の物体が接続されているものとします．従って物体にはバネの伸びに比例した力が働きます．また物体には重力が鉛直下向きに働くものとします．状況を図 1 に示します．

この物体の運動を支配する運動方程式を書き下してみましょう．時刻 $t$ における位置を $y(t)$ とすると，Newton の運動方程式は次のようになります．

$$m\frac{d^2y(t)}{dt^2} = -k\{y(t) - L_0\} + mg \tag{1}$$

質量×加速度 =力ですので，右辺に力を集めます．符号に注意しましょう．バネが伸びているとき，すなわち，$y(t) - L_0 > 0$ のときに，バネの復元力が-y 方向に働く，という常識的な事実との整合性が取れているか確認しましょう．

微分方程式の数値解を求めるためには，必ず初期条件 (や境界条件) が必要です．二階常微分方程式の初期値問題を解くためには，初期条件は二つ必要です．時刻 $t = 0$ でバネの長さを $L$ とします (条件 1)．また物体の初速度を 0 とします (条件 2)．

## 1.2　方程式の離散化

現在の計算機は連続量を扱うことができません．微分方程式の数値解を得るためには，ある一定の時間刻み $\Delta t$ で離散化し，差分方程式として順次数値解を求めていきます．

二階微分方程式は，2 変数の連立一階微分方程式に帰着されます．

$$v(t) = \frac{dy(t)}{dt} \tag{2}$$

と，新たに変数 $v(t)$ を定義すると，

$$m\frac{dv(t)}{dt} = -k\{y(t) - L_0\} + mg \tag{3}$$

$$\frac{dy(t)}{dt} = v(t) \tag{4}$$

と書くことができます．$v(t)$ は物体の速度で，下向きが正です．

次にこれらの方程式の微分項を離散化します．最初に離散化の最も単純な方法を示します．

まず，離散的な時刻 $t_i = i\Delta t$ を考えます．時刻 $t_i$ における $y(t)$ および $v(t)$ を，それぞれ $y_i = y(t_i)$，$v_i = v(t_i)$ と置きます．

時刻 $t_i$ までの数値解が既知 (すなわち，$y_i$ と $v_i$ の値はわかっている) の場合に，時刻 $t_{i+1}$ での数値解 $y_{i+1}$ と $v_{i+1}$ を求めることを考えます．

時間微分を差分で近似します．

$$\frac{dy(t)}{dt} \to \frac{y_{i+1} - y_i}{\Delta t} \tag{5}$$

$$\frac{dv(t)}{dt} \to \frac{v_{i+1} - v_i}{\Delta t} \tag{6}$$

この置き換えは最も単純な差分近似の一つで，前進差分と呼ばれます．前進差分を用いた方程式の数値解法を前進 Euler 法と呼びます．前進差分によって，微分方程式は次の差分方程式になります．

$$m\frac{v_{i+1} - v_i}{\Delta t} = -k\{y_i - L_0\} + mg \tag{7}$$

既知の値を右辺に集めると，未知の値 $v_{i+1}$ は，

$$v_{i+1} = v_i + \left[-\frac{k}{m}\{y_i - L_0\} + g\right]\Delta t \tag{8}$$

で得られることがわかります．一般性のある形で書けば，

$$v_{i+1} = v_i + \frac{F_i}{m}\Delta t \tag{9}$$

です．同様にして，$y_{i+1}$ は，

$$y_{i+1} = y_i + v_i\Delta t \tag{10}$$

となります．

表 1　単振動シミュレーションのパラメータ

| パラメータ | 記号 | 値 |
|---|---|---|
| バネ定数 | $k$ | 100.0 |
| 自然長 | $L_0$ | 0.2 |
| 初期状態のバネの長さ | $L$ | 0.2 |
| 重力加速度 | $g$ | 9.8 |
| 質量 | $m$ | 0.5 |
| 時間刻み | $\Delta t$ | 0.01 |

時間微分を差分で近似するとは

先程気軽に「時間微分を差分で近似」しましたが，見方を変えてみます．まず微分方程式を考えます．独立変数まで丁寧に書くと，

$$\frac{dx(t)}{dt} = f(x(t), t) \tag{11}$$

これを，時刻 $t_i$ から $t_{i+1}$ まで積分します．

$$x_{i+1} - x_i = \int_{t_i}^{t_{i+1}} f(x(t), t)dt \tag{12}$$

右辺の積分を近似します．時刻 $t_i$ での値が，$t_i \leq t \leq t_{i+1}$ の間継続しているものとすると，

$$x_{i+1} - x_i = f(x_i, t_i)\Delta t \tag{13}$$

となりますから，

$$\frac{x_{i+1} - x_i}{\Delta t} = f(x_i, t_i) \tag{14}$$

これはまさに，時間微分を差分で近似した式と同じです．つまり，あまり深く考えずに行った差分による近似は，数値積分を 0 次近似することと同じだったのです．そう考えると，一見妥当で無害に見える差分による近似ですが，かなりの誤差を含んでいそうだと思えてこないでしょうか．

## 1.3　実装してみよう

単振動シミュレーションを実装してみましょう．式 (9) と式 (10) に従って，計算を進めます．プログラムの作成にはこれまでと同様 Processing を用います．

実装には具体的な値が必要です．単位系を MKS 単位系 (長さを m，質量を kg，時間を s(秒) で表す) とします．ここでは，表 1 の値を用います．バネの片方は $y = 0$ に固定されているものとします．物体の初期位置は $y = 0.2$ とします．また計算の時間刻みは物理定数ではありませんが非常に重要なパラメータです．ここでは $\Delta t = 0.01$ とします．

位置 $y$ は変数 `y` に，速度 $v$ は変数 `vy` に，力 $f$ は変数 `fy` に格納します．

**■■■ 演習: 単振動の物理シミュレーション**　以下のソースコードを入力し，実行してみよ．

リスト 1　単振動その 1

```
1 // oscillation1
2
3 float m;
```

```
float y;
float vy;
float fy;

float radius;

float k = 100.0;
float l0 = 0.2;
float gy = 9.8;

float dt = 0.01;

float viewingSize = 0.5;

int xOffset;
int yOffset;
float viewingScale;

void setup() {
  size(512, 512);
  frameRate(60);
  smooth(4);

  xOffset = width / 2;
  yOffset = 0;
  viewingScale = width / viewingSize;

  m = 0.5;
  y = 0.2;
  vy = 0.0;
  fy = 0.0;

  radius = 0.05;
}

void strokeWeightScaled(float s) {
  strokeWeight(s / viewingScale);
}

void keyPressed() {
  if (key == ESC || key == 'q') {
    exit();
  }
}

void draw() {
  fy = -k * (y - l0) + m * gy;
  float yNew = y + vy * dt;
  float vyNew = vy + fy / m * dt;
  y = yNew;
  vy = vyNew;

  ellipseMode(RADIUS);
  background(255);

  translate(xOffset, yOffset);
  scale(viewingScale);

  fill(224);
```

```
63    stroke(128);
64    strokeWeightScaled(1.0);
65
66    line(0, 0, 0, y);
67    translate(0, y);
68    ellipse(0, 0, radius, radius);
69  }
```

■物理シミュレーション部　変数を適宜宣言し，setup() 中で初期化します．位置と速度の更新は，draw() 中の 50 行から 54 行で行われます．

■一時変数の使用　計算は draw() 内で行っています．次の時刻での位置および速度である $y_{i+1}$ および $v_{i+1}$ の値を格納するために，変数 yNew および vNew を確保しています．これらを使わずに直接変数 y および v を更新した場合，後に更新された方 (ここでは $v_{i+1}$ の計算) では，$v_i$ と $y_i$ ではなく $v_i$ と $y_{i+1}$ の値が使われてしまいます．これは好ましくないと思われるので，変数を分けて回避します．

■シミュレーションのスケールとウィンドウとの対応　シミュレーションはおおむね 0.1 から 1 程度のスケール (10cm から 1m に相当) で行われます．この座標値をそのまま描画すると，1 ピクセルよりも小さくなってしまうために識別不可能です．よって画面に描画する際にはスケールの変換が必要です．

本節では，画面の幅 (viewingScale に保持) が 0.5 になるようにスケールを変換します．画面の幅を使う必然性は無いのですが，ウィンドウのスケールの代表値として使うにあたって特に不満はありません．viewingScale にスケール変換の係数を設定します．

またウィンドウの上辺中央を座標系の原点としています．ウィンドウにおける座標系原点の座標を，xOffset および yOffset で保持しています (27 行および 28 行)．

座標の対応関係を図 2 に示します．この座標変換は，描画開始時に translate() と scale() を適用することで実現しています．後の描画はウィンドウの座標系を意識することなく，行うことができます．しかし一点配慮が必要な箇所は，描画する線の太さを設定する関数 storkeWeight() も scale() の影響を受けてしまうことです．ここでは，strokeWeightScaled() を定義して，設定時にスケーリングの影響を相殺するように倍率を変更します．

■物体の描画　物体は ellipse() で描いています．ellipse() の各引数の意味は ellipseMode() で変更することができます．ここでは draw() で描画を開始する前に，第 3 引数と第 4 引数を半径 (より正確には，幅や高さの半分) と解釈するように ellipseMode(RADIUS) をあらかじめ実行しています．

## 1.4　恐ろしい結末

さて，何となくうまく動いているように見えますが，時間が経つと，明らかに振幅が大きくなっていき，破綻します．これは正しくありません．減衰の無い単振動では，振幅は変化しないはずです．

原因は数値解析につきものの計算誤差です．微分を差分に置き換える方法に問題があります．

次は，この問題を解決しましょう．

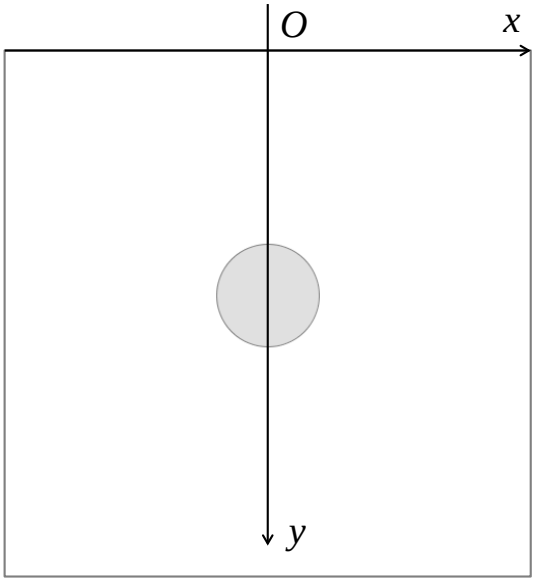


図 2　座標の対応

# 2　Adams-Bashforth 法による数値積分の改善

数値積分の方法を変更することにより，安定性を改善することができます．様々な方法が提案されていますが，ここでは導入が比較的容易な Adams-Bashforth 法を使ってみましょう．

## 2.1　Adams-Bashforth 法

物体の位置を $y(t)$，物体に働く力を $F(t)$ とします．Newton の運動方程式は次のようになります．

$$m\frac{d^2y(t)}{dt^2} = F(t) \tag{15}$$

また速度 $v(t)$ は，

$$v(t) = \frac{dy(t)}{dt} \tag{16}$$

です．

単振動の方程式を一階連立微分方程式に直すのと同様にして，

$$\frac{dv(t)}{dt} = \frac{F(t)}{m} \tag{17}$$

$$\frac{dy(t)}{dt} = v(t) \tag{18}$$

どちらも一階常微分方程式になったので，一般的な一階常微分方程式を考えます．すなわち，

$$\frac{dy(t)}{dt} = f(y,t) \tag{19}$$

素直に，両辺を $t_i$ から $t_{i+1}$ まで積分すると，

$$y_{i+1} - y_i = \int_{t_i}^{t_{i+1}} f(y,t)dt \tag{20}$$

いま，時刻 $t_i$ の状態が求まっているとします．しかし，右辺の積分の被積分関数は，$t_i < t \leq t_{i+1}$ で未知であるため，積分を実行することができません．仕方がないので，過去の値を用いて外挿します．最も簡単な例は，$f_i$ の値をそのまま使うというもので，これが先程行った方法 (前進 Euler 法) に相当します．次に簡単な方法は線形近似で，$t_{i-1}$ と $t_i$ での値 $f_{i-1}$ と $f_i$ を用いて，線形に外挿します．すなわち，

$$f(y,t) - f_i = \frac{f_i - f_{i-1}}{\Delta t}(t - t_i) \tag{21}$$

よって，

$$y_{i+1} - y_i = \left[ f_i t + \frac{1}{2}\frac{f_i - f_{i-1}}{\Delta t}(t - t_i)^2 \right]_{t_i}^{t_{i+1}} = f_i\Delta t + \frac{1}{2}\frac{f_i - f_{i-1}}{\Delta t}(\Delta t)^2 = \left(\frac{3f_i}{2} - \frac{f_{i-1}}{2}\right)\Delta t \tag{22}$$

これは 2 次の Adams-Bashforth 法と呼ばれます．

一般的に，過去の複数の時刻での値を用いる場合を多段階法と呼び，$n$ 時刻分用いるものを $n$ 段階法と呼びます．

さて，Newton の運動方程式に適用してみましょう．

$$y_{i+1} = y_i + \left(\frac{3v_i}{2} - \frac{v_{i-1}}{2}\right)\Delta t \tag{23}$$

$$v_{i+1} = v_i + \left(\frac{3F_i}{2} - \frac{F_{i-1}}{2}\right)\frac{\Delta t}{m} \tag{24}$$

$$\tag{25}$$

従って，過去 2 時刻分の $y_i$，$v_i$，$F_i$ から，新たな時刻での数値解が計算可能であることがわかります．

## 2.2 実装してみよう

先程のプログラムを改良して，2 次の Adams-Bashforth 法を実装してみましょう．過去 2 時刻分の情報を保持しておく必要あることから，yPrev，vPrev，fPrev を準備し，1 時刻前の位置，速度，力を保管するために使います．

多段階法の場合，計算開始直後に，過去の値として何を使うかを決める必要があり，やや厄介です．2 次の Adams-Bashforth 法の場合，$f_i$ と $f_{i-1}$ が等しいとすると，前進 Euler 法になります．よって初期状態と同じ値を使います．

■■■ **演習: 数値積分の改良** 前節のプログラムを変更し，Adams-Bashforth 法を実装せよ．具体的には，

- float 型の変数 yPrev, vyPrev, fyPrev を宣言する．
- setup() での初期化の際には，上記 3 変数は初期値と同じ値を設定する．
- draw() で yNew および vyNew を更新する式を変更する．式 (24) と式 (23) に従って，計算を進める．
- draw() の計算処理の最後に，yPrev, vyPrev, fyPrev に現在の値を保存しておく．

テンプレートを示すので，変更点を確認して手元のソースコードに追加した上で，空欄になっている部分を埋めよ．添字が $i$ のものは今の値を使う．また添字が $i-1$ は 1 時刻前の値で変数名 *Prev に格納されている．

実行してみて，安定性が改善していることを確認せよ．

■■■ **演習: 物理定数の変更 1** バネ定数 k の値を変更し，挙動の変化を確認せよ．

■■■ **演習: 物理定数の変更 2** 初期位置 l0 の値を変更し，挙動の変化を確認せよ．

■■■ **演習: 時間刻みの変更** 時間刻み dt の値を徐々に大きくしていくと何が起こるだろうか．確かめよ．

■■■ **演習: [発展] 周期の検出** 位置が極大となった際に，時刻を出力せよ．また前回極大となった時刻から経過した時間，すなわち単振動の周期を算出し出力せよ．

リスト 2　単振動その 2

```
1  // oscillation2
2
3  float m;
4  float y;
5  float vy;
6  float fy;
7
8  float radius;
9
10 float yPrev;
11 float vyPrev;
12 float fyPrev;
13
14 float k = 100.0;
15 float l0 = 0.2;
16 float gy = 9.8;
17
18 float dt = 0.01;
19
20 float viewingSize = 0.5;
21
22 int xOffset;
```

```
int yOffset;
float viewingScale;

void setup() {
  size(512, 512);
  frameRate(60);
  smooth(4);

  xOffset = width / 2;
  yOffset = 0;
  viewingScale = width / viewingSize;

  m = 0.5;
  y = 0.2;
  vy = 0.0;
  fy = 0.0;

  radius = 0.05;

  yPrev = y;
  vyPrev = vy;
  fyPrev = fy;
}

void strokeWeightScaled(float s) {
  strokeWeight(s / viewingScale);
}

void keyPressed() {
  if (key == ESC || key == 'q') {
    exit();
  }
}

void draw() {
  fy = -k * (y - l0) + m * gy;
  float yNew = ___________________________________;
  float vyNew = ___________________________________;
  yPrev = y;
  vyPrev = vy;
  fyPrev = fy;
  y = yNew;
  vy = vyNew;

  ellipseMode(RADIUS);
  background(255);

  translate(xOffset, yOffset);
  scale(viewingScale);

  fill(224);
  stroke(128);
  strokeWeightScaled(1.0);

  line(0, 0, 0, y);
  translate(0, y);
  ellipse(0, 0, radius, radius);
}
```

# 3 ダンパの導入

ダンパを追加することにより，粘性 (この場合は空気抵抗) を表現できます．

## 3.1 ダンパとは

ダンパとは，運動エネルギーを減衰させるもの全般を指します．速度に比例した抵抗を導入することによって，バネの運動が徐々に減衰する現象が表現できます．

これまでは，質点に働く力は，バネの復元力と重力のみでした．

$$F(t) = -k\{y(t) - L_0\} + mg \tag{26}$$

これにダンパを追加すると，

$$F(t) = -k\{y(t) - L_0\} - dv(t) + mg \tag{27}$$

となります．$d$ はダンパの減衰係数です．

**■■■ 演習: ダンパの追加** 前節のプログラムにダンパを追加し，挙動の違いを確認せよ．具体的には，

- 物理定数 (`k` や `l0` など) が定義されている箇所の最後に，`float` 型の変数 `d` を宣言し，値を 0.5 として初期化する．
- y 軸方向に働く力 `fy` を計算している箇所を，ダンパによる減衰項を加えた式 (27) の内容に変更する．

**■■■ 演習: ダンパの強さ** ダンパの減衰係数を適宜変更し，挙動の変化を確認せよ．

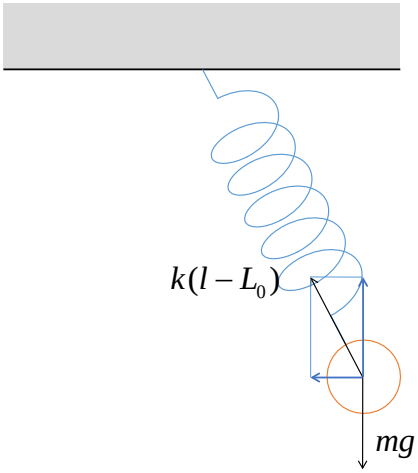


図 3　バネにより単振動する物体: 2 次元

# 4　2 次元化

バネによる単振動は安定してシミュレーションできるようになったが，このままではあまり面白いことは起こりそうにありません．拡張の方向として，

- 次元を上げて 2 次元 (あるいは 3 次元) とする．
- 物体およびバネを増やす．

などが考えられます．ここではまず，次元を増やしてみましょう．

## 4.1　1 次元から 2 次元に

1 次元から 2 次元になっても，計算の本質には変更はありません．ただし，力や運動を，図 3 に示すように，各軸方向の成分に分けて考える必要があります．

- 弾性力については，バネの固定点と質点を結ぶ方向の単位ベクトル $\boldsymbol{e} = (e_x, e_y)$ を考えます (図 4)．バネによる弾性力の方向は $\boldsymbol{e}$ で，バネの長さを $l$ とすると弾性力の大きさは符号を含めて $-k(l - L_0)$ になります．よって，弾性力 $\boldsymbol{F}_s$ の各軸方向成分は，

$$F_{sx} = -k(l - L_0)e_x \tag{28}$$
$$F_{sy} = -k(l - L_0)e_y \tag{29}$$

  となります．
- 粘性力は，各軸方向の速度に減衰係数を乗じたものになります．
- 重力は，$y$ 軸方向にしか働きません．

以上を踏まえると，物体の位置を $(x, y)$ として，各軸方向に働く力の計算手順は以下のようになります．

1. バネの長さ $l = \sqrt{x^2 + y^2}$ を計算します．
2. バネの単位ベクトルの成分 $e_x = x/l, e_y = y/l$ を計算します．

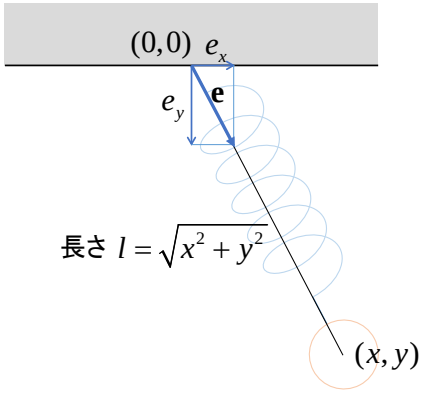


図 4　バネによる弾性力が働く方向のベクトル

3. 質点に働く力の x 軸方向成分 $f_x$ および y 軸方向成分 $f_y$ は以下の式で与えられます．

$$f_x = -k(l - L_0)e_x - dv_x \tag{30}$$
$$f_y = -k(l - L_0)e_y - dv_y + mg \tag{31}$$

## 4.2　実装

下記に一部を欠損させたソースコードを示します．物体の初期位置は $(x, y) = (0.05, 0.2)$ とし，その他の条件は 1 次元の場合と同一とします．

**■■■ 演習: 2 次元への拡張**　ソースコードの穴空き部分を埋めて，バネのシミュレーションを 2 次元に拡張せよ．

リスト 3　単振動 2 次元

```
// oscillation4

float m;
float x, y;
float vx, vy;
float fx, fy;

float radius;

float xPrev, yPrev;
float vxPrev, vyPrev;
float fxPrev, fyPrev;

float k = 100.0;
float l0 = 0.2;
float d = 0.5;
float gx = 0.0;
float gy = 9.8;

float dt = 0.01;

float viewingSize = 0.5;

int xOffset;
int yOffset;
```

```
26 float viewingScale;
27
28 void setup() {
29   size(512, 512);
30   frameRate(60);
31   smooth(4);
32
33   xOffset = width / 2;
34   yOffset = 0;
35   viewingScale = width / viewingSize;
36
37   m = 0.5;
38   x = 0.05;
39   y = 0.2;
40   vx = ________;
41   vy = 0.0;
42   fx = ________;
43   fy = 0.0;
44
45   radius = 0.05;
46
47   xPrev = ________;
48   yPrev = y;
49   vxPrev = ________;
50   vyPrev = vy;
51   fxPrev = ________;
52   fyPrev = fy;
53 }
54
55 void strokeWeightScaled(float s) {
56   strokeWeight(s / viewingScale);
57 }
58
59 void keyPressed() {
60   if (key == ESC || key == 'q') {
61     exit();
62   }
63 }
64
65 void draw() {
66   float l = _____________________________;
67   float ex = _____________________________;
68   float ey = _____________________________;
69   fx = _____________________________;
70   fy = _____________________________;
71   float xNew = _____________________________;
72   float yNew = _____________________________;
73   float vxNew = _____________________________;
74   float vyNew = _____________________________;
75
76   xPrev = ________;
77   yPrev = y;
78   vxPrev = ________;
79   vyPrev = vy;
80   fxPrev = ________;
81   fyPrev = fy;
82
83   x = ________;
84   y = yNew;
```

```
85    vx = ________;
86    vy = vyNew;
87
88    ellipseMode(RADIUS);
89    background(255);
90
91    translate(xOffset, yOffset);
92    scale(viewingScale);
93
94    fill(224);
95    stroke(128);
96    strokeWeightScaled(1.0);
97
98    line(0, 0, x, y);
99    translate(x, y);
100   ellipse(0, 0, radius, radius);
101 }
```

**■■■ 演習: [発展] 風が吹いている** この空間に周期的に風が吹いているとして，その時刻・その場所での風速に比例した外力を物体に加えてみよ．

**■■■ 演習: [発展] 二重振り子** 物体とバネ (-ダンパ) をもう一つ増やし，二重振り子を作成せよ．2 個目の物体の初期位置等は適当に設定せよ．

**■PVector の導入について** 2 次元になったので，Processing が提供しているベクトルを扱うクラスである `PVector` を導入したくなりますが，`PVector` は扱いに注意が必要な点がありますので，ここでは敢えて導入しません．興味のある人は各自実施してください．

組み込み型の変数と違いがあって留意しなければいけない点を以下に挙げます．

- 普通の型ではなくクラスなので，宣言後に必ず `new PVector()` が必要．
- 参照なので，通常の代入は浅いコピー (参照元は同一)．代入先を書き換えると代入元も書き変わるが，この挙動は組み込み型と異なるため混乱すること必定．深いコピー (実体を別にする) はメソッド `copy()` を使用する必要がある．
- 演算子オーバーロードがなされていない (そもそも Processing の元となっている Java には無いんですよね) ので，計算式の記述が面倒．
- 同一名の (インスタンス) メソッドとクラスメソッドが存在する．インスタンスメソッドはインスタンスの内容を書き換えることを把握していないと，気付かないうちに値が変化している現象に頭を悩ますことになる．

この後，クラスの使用，多数の質点，スレッド化処理，と続きます．