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)

2016 年 12 月 13 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

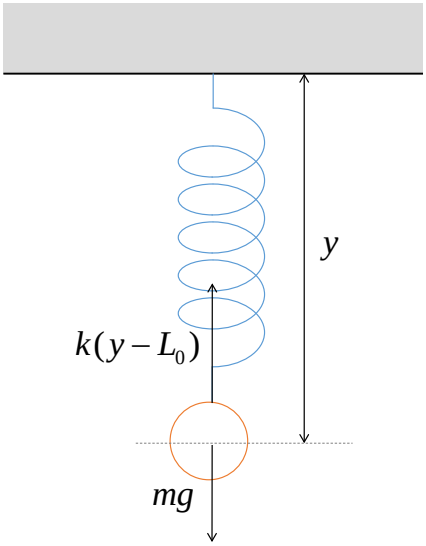

図 1　バネにより単振動する物体

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 Δt 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．

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，$t_i \le t \le t_{i+1}$ 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．

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	Δt	0.01

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

```
// oscillation1

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;
```

```
32    y = 0.2;
33    vy = 0.0;
34    fy = 0.0;
35
36    radius = 0.05;
37  }
38
39  void strokeWeightScaled(float s) {
40    strokeWeight(s / viewingScale);
41  }
42
43  void keyPressed() {
44    if (key == ESC || key == 'q') {
45      exit();
46    }
47  }
48
49  void draw() {
50    fy = -k * (y - l0) + m * gy;
51    float yNew = y + vy * dt;
52    float vyNew = vy + fy / m * dt;
53    y = yNew;
54    vy = vyNew;
55
56    ellipseMode(RADIUS);
57    background(255);
58
59    translate(xOffset, yOffset);
60    scale(viewingScale);
61
62    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

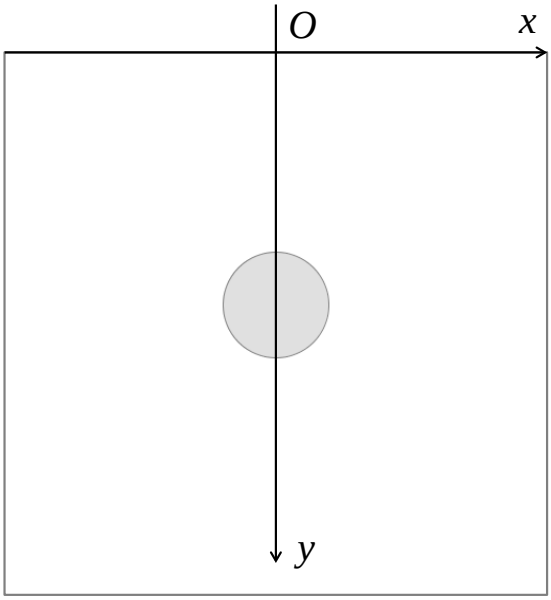

図 2　座標の対応

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．

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;
23 int yOffset;
24 float viewingScale;
25
26 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．

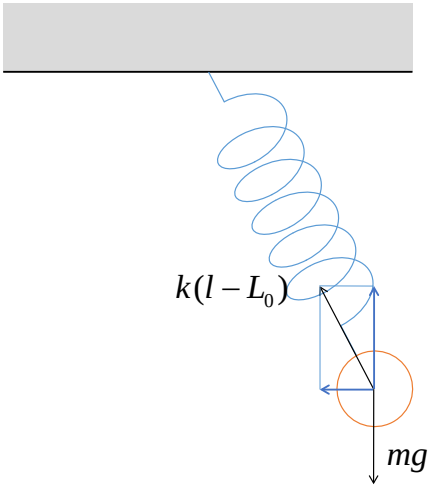

図 3　バネにより単振動する物体: 2 次元

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．
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．

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

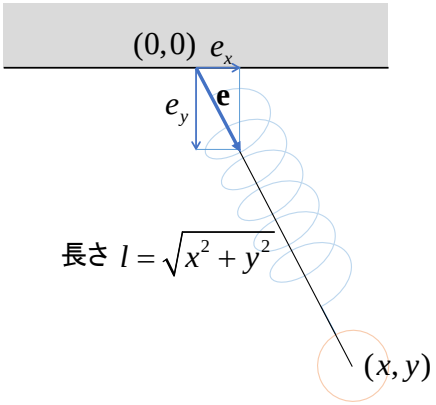

図 4　バネによる弾性力が働く方向のベクトル

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;
float viewingScale;

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

  xOffset = width / 2;
  yOffset = 0;
  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);
```

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

■■■ 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もあるので，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．

リスト 4　単振動 2 次元 (完成版)

```
// 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;
float viewingScale;

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

  xOffset = width / 2;
  yOffset = 0;
```

```
  viewingScale = width / viewingSize;

  m = 0.5;
  x = 0.05;
  y = 0.2;
  vx = 0.0;
  vy = 0.0;
  fx = 0.0;
  fy = 0.0;

  radius = 0.05;

  xPrev = x;
  yPrev = y;
  vxPrev = vx;
  vyPrev = vy;
  fxPrev = fx;
  fyPrev = fy;
}

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

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

void draw() {
  float l = sqrt(x * x + y * y);
  float ex = x / l;
  float ey = y / l;
  fx = -k * (l - l0) * ex - d * vx + m * gx;
  fy = -k * (l - l0) * ey - d * vy + m * gy;
  float xNew = x + (3 * vx - vxPrev) * 0.5 * dt;
  float yNew = y + (3 * vy - vyPrev) * 0.5 * dt;
  float vxNew = vx + (3 * fx - fxPrev) * 0.5 * dt / m;
  float vyNew = vy + (3 * fy - fyPrev) * 0.5 * dt / m;

  xPrev = x;
  yPrev = y;
  vxPrev = vx;
  vyPrev = vy;
  fxPrev = fx;
  fyPrev = fy;

  x = xNew;
  y = yNew;
  vx = vxNew;
  vy = vyNew;

  ellipseMode(RADIUS);
  background(255);

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

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

```

```
98    line(0, 0, x, y);
99    translate(x, y);
100   ellipse(0, 0, radius, radius);
101 }
```

5 クラス化

ここまでのプログラムでは，物体の数は 1 個，バネの数も 1 本と想定されていました．今後，複数の質点 (粒子)，複数のバネによって物体を表現するにあたって，このまま拡張を続けるのは得策ではありません．プログラムを整理し，より拡張が容易な形にします．

5.1 整理 1: シミュレーション部分の整理

最初に，シミュレーション部分を関数化します．シミュレーションに関しては，大きく分けると，

- 初期化
- 計算
- 描画

の 3 パートになります．それぞれの関数を `simulationInit()`，`simulationCalc()`，`simulationDraw()` とし，各関数にシミュレーションに関連する処理を集約します．

5.2 整理 2: 物体とバネのクラス化

クラスとは，ある対象に関する変数 (フィールド) と関数 (メソッド) を一括して扱うための構造です．C 言語で言えば，構造体とその構造体に関連する関数を一緒にしたものです．

変数と対比すると，クラスは変数の型に相当します．変数の実体に相当するものを，Processing ではオブジェクトと呼んでいます．オブジェクト指向プログラミングではインスタンスと呼ばれることが一般的です．

現在のプログラムには，物体とバネ (ダンパ) が登場します．これらに関する情報を整理します．

物体の方は比較的簡単です．物体に関する情報を集めて整理します．なお今後，物体ではなく粒子と呼び，クラス名は `Particle` とします．またバネの方は `Spring` クラスとします．

■Particle クラスの解説　フィールド `boolean isFixed` は，この粒子が動かない粒子であれば `true` になります．これは，バネの一端が壁面に固定されている場合に，壁面クラスを別途定義するのではなく，位置が固定された粒子を配置することで表現するために使用します．

クラス名と同じ名前の関数 `Particle()` はコンストラクタと呼ばれます．コンストラクタはインスタンスが作成された際に自動的に呼び出される特別な関数です．フィールドの初期値設定など，必ず行わなければならないインスタンス生成時の処理を記述します．[*1]

メソッドとしては以下を用意し，2 から 5 を繰り返します．

1. `init()`: 初期化します．
2. `clearForce()`: 力を 0 にします．
3. `addForce()`: 力を加えます．必要であれば何度も呼ばれます．
4. `move()`: 移動します．
5. `draw()`: 描画します．

このうち `addForce()` を他の関数から呼び出すことによって，粒子に働く力を合算します．

■Spring クラスの解説　バネは片方が (0,0)，片方が (x,y) と固定されていましたが，複数のバネへの一般化を考えます．両端は Particle として，Particle への参照を保持します．

メソッド `calc()` の中では，両端の粒子の位置と速度に基づいて，バネが発揮する力を計算し，両端の粒子

[*1] Processing は Java の流れを汲んでいるため，デストラクタはありません．

(フィールド particles[] に保持されています) のメソッド addParticle() を呼び出して，力を作用させます．

クラス化が完了したソースコードは次のようになります．

リスト 5　単振動 2 次元 (クラス化)

```
// oscillation5c

// physical parameters

float x0 = 0.05;
float y0 = 0.2;

float m = 0.5;

float radius = 0.05;

float k = 100.0;
float l0 = 0.2;
float d = 0.5;

float gx = 0.0;
float gy = 9.8;

float dt = 0.01;

// viewing parameters

float viewingSize = 0.5;

int xOffset;
int yOffset;
float viewingScale;

// objects

int nParticles = 2;
int nSprings = 1;

Particle[] particles;
Spring[] springs;



// particle ////////////////////////////////////////////////////////////////

class Particle {
  float x, y;
  float vx, vy;
  float fx, fy;

  float m;
  boolean isFixed;

  float radius;

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

  Particle(float x0, float y0, float vx0, float vy0, float m0,
           boolean f, float r) {
    m = m0;
    x = x0;
```

```
    y = y0;
    vx = vx0;
    vy = vy0;
    isFixed = f;
    radius = r;
  }

  void init() {
    fx = 0.0;
    fy = 0.0;

    xPrev = x;
    yPrev = y;
    vxPrev = vx;
    vyPrev = vy;
    fxPrev = fx;
    fyPrev = fy;
  }

  void clearForce() {
    fx = 0.0;
    fy = 0.0;
  }

  void addForce(float x, float y) {
    fx += x;
    fy += y;
  }

  void move(float dt) {
    if (isFixed) {
      return;
    }

    float xNew = x + (3 * vx - vxPrev) * 0.5 * dt;
    float yNew = y + (3 * vy - vyPrev) * 0.5 * dt;
    float vxNew = vx + (3 * fx - fxPrev) * 0.5 * dt / m;
    float vyNew = vy + (3 * fy - fyPrev) * 0.5 * dt / m;

    xPrev = x;
    yPrev = y;
    vxPrev = vx;
    vyPrev = vy;
    fxPrev = fx;
    fyPrev = fy;

    x = xNew;
    y = yNew;
    vx = vxNew;
    vy = vyNew;
  }

  void draw() {
    if (radius <= 0.0) {
      return;
    }

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

    pushMatrix();
    translate(x, y);
```

```
122     ellipse(0, 0, radius, radius);
123     popMatrix();
124   }
125 };
126 
127 // spring ////////////////////////////////////////////////////////////////////
128 
129 class Spring {
130   Particle[] particles;
131   float k;
132   float l0;
133   float d;
134 
135   Spring(Particle p0, Particle p1, float k0, float l00, float d0)
136   {
137     particles = new Particle[2];
138     particles[0] = p0;
139     particles[1] = p1;
140     k = k0;
141     l0 = l00;
142     d = d0;
143   }
144 
145   void init() {
146   }
147 
148   void calc() {
149     float dx = particles[1].x - particles[0].x;
150     float dy = particles[1].y - particles[0].y;
151     float l = sqrt(dx * dx + dy * dy);
152     float ex = dx / l;
153     float ey = dy / l;
154     float fx = -k * (l - l0) * ex;
155     float fy = -k * (l - l0) * ey;
156 
157     float vx = particles[1].vx - particles[0].vx;
158     float vy = particles[1].vy - particles[0].vy;
159     fx += -d * vx;
160     fy += -d * vy;
161 
162     particles[0].addForce(-fx, -fy);
163     particles[1].addForce(fx, fy);
164   }
165 
166   void draw() {
167     noFill();
168     stroke(128);
169     strokeWeightScaled(1.0);
170 
171     line(particles[0].x, particles[0].y, particles[1].x, particles[1].y);
172   }
173 };
174 
175 // simulation ////////////////////////////////////////////////////////////////
176 
177 void simulationInit() {
178   particles = new Particle[nParticles];
179   particles[0] = new Particle(0.0, 0.0, 0.0, 0.0, m, true, 0.0);
180   particles[1] = new Particle(x0, y0, 0.0, 0.0, m, false, radius);
181 
182   springs = new Spring[nSprings];
183   springs[0] = new Spring(particles[0], particles[1], k, l0, d);
184 
```

```
  for (Particle p: particles) {
    p.init();
  }

  for (Spring s: springs) {
    s.init();
  }
}

void simulationCalc(float dt) {
  for (Particle p: particles) {
    p.clearForce();
    p.addForce(p.m * gx, p.m * gy);
  }

  for (Spring s: springs) {
    s.calc();
  }

  for (Particle p: particles) {
    p.move(dt);
  }
}

void simulationDraw() {
  ellipseMode(RADIUS);
  background(255);

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

  for (Spring s: springs) {
    s.draw();
  }

  for (Particle p: particles) {
    p.draw();
  }
}

// utility function ////////////////////////////////////////////////////////

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

// setup ///////////////////////////////////////////////////////////////////

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

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

  simulationInit();
}

// draw ////////////////////////////////////////////////////////////////////

void draw() {
```

```
248   simulationCalc(dt);
249   simulationDraw();
250 }
251
252 // callbacks ////////////////////////////////////////////////////////////////
253
254 void keyPressed() {
255   if (key == ESC || key == 'q') {
256     exit();
257   }
258 }
```

■■■ 演習: クラス化の確認 前節のソースコードと比較し，クラス化によってどこがどう変わったのか，確認せよ.

■新しい事項の説明

- 配列の各要素に対して順次処理を行う場合，for 文を次のように記述することができます.

```
1 particles = new Particle[nParticles];
2 :
3 for (Particle p: particles) {
4   // p について処理
5 }
```

6 マウスによるインタラクション

簡単なインタラクションを実装してみよう．

6.1 ポインタに引き寄せられる

ウィンドウ内にポインタがあり，マウスのボタンが押されていたら，そのポインタの方向に引き寄せられるようにしてみる．

マウスのボタンが押されているかどうかは，Processing が用意している変数 mousePressed を調べることにより判断できる．https://processing.org/reference/mousePressed.html

またポインタの位置は，変数 mouseX および mouseY に格納されている．https://processing.org/reference/mouseX.html

ポインタの方向に球を引き寄せる方法の一つは，球の中心からポインタに向かうベクトルを考えて，そのベクトルに比例した力を加えることである．このとき，ポインタの座標はウィンドウ座標系で，球の座標は物理シミュレーションの座標系で与えられているので，ポインタの座標を物理シミュレーションの座標系へと変換することが必要である．

以上を踏まえて，リスト 5 の 199 行目，simulationCalc() 中に，

1. まず，マウスのボタンが押されているか調べる．
2. 押されている場合，ポインタの座標を物理シミュレーションの座標系へと変換する．
3. 以下の処理を全ての Particle に対して行う．
   （a）球の中心からポインタへと向かうベクトルの成分を計算する．
   （b）適当な比例係数を乗じて，メソッド addForce() を呼び出す．

といった処理を加えれば実現できる．この部分に加えるべきソースコードは以下のようになる．

```
1   if (mousePressed == true) {
2    float mx = (mouseX - xOffset) / viewingScale;
3    float my = (mouseY - yOffset) / viewingScale;
4    for (Particle p: particles) {
5      float dx = mx - p.x;
6      float dy = my - p.y;
7      float pullK = 50.0;
8      p.addForce(pullK * dx, pullK * dy);
9    }
10  }
```

■■■ 演習: ポインタ方向への引力 上記ソースコードを付け加えよ．

■■■ 演習: 球に働いている力 Particle クラスのメソッド draw() を改良し，球に働いている力を，球の中心から力の方向に向かって伸びる線で表せ．力の X 成分および Y 成分はフィールド fx および fy に格納されているので適当にスケーリングして表示する．線の色は適当に付けよ．

■■■ 演習: 球とバネを複数にする クラス化によるメリットとして，物体を複数に増やすことが容易であることが挙げられる．以下の変更を加えて，物体を 4 個にしてみよう．注目すべき点は，シミュレーションの本体部分には全く手を入れずに，ただインスタンスの数を増やしてそれらの関係 (ここでは Spring のコンストラクタで指定する両端の Particle) を適切に設定するだけで，簡単に拡張できる点である．

- viewingSize を 2.0 にする (本質的ではないが，物体が増えたので見える範囲を広くする)．
- nParticles と nSprings を以下のように設定する．

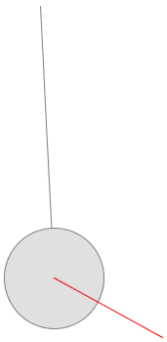

図 5　2 次元化 + 力の表示

図 6　物体を複数に

```
int nParticles = 5;
int nSprings = nParticles - 1;
```

- `simulationInit()` 冒頭の，Particle と Spring の初期化の箇所を，以下のように書き換える．

```
  particles = new Particle[nParticles];
  for (int i = 0; i < nParticles; i++) {
    if (i == 0) {
      particles[i] = new Particle(0.0, 0.0, 0.0, 0.0, m, true, 0.0);
    } else {
      particles[i] = new Particle(x0 * i, y0 * i, 0.0, 0.0, m, false, radius);
    }
  }

  springs = new Spring[nSprings];
  for (int i = 0; i < nSprings; i++) {
    springs[i] = new Spring(particles[i], particles[i + 1], k, l0, d);
  }
```

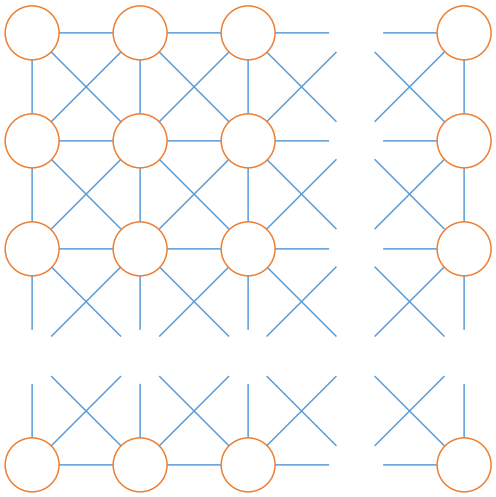

図 7　粒子とバネの配置

7　粘弾性体の表現

バネ-ダンパ-質点モデルでは，複数の質点とそれらを接続するバネ-ダンパで粘弾性体を表現する．基本的にはこれまで作成してきたプログラムの物体を差し替えればよいが，壁面との衝突を適切に表現するために，計算と表示を分離する必要がある．

7.1　多数の質点への拡張

粒子とバネを図 7 のように接続する．斜めのバネを入れることによって，剪断力に対して強くなる．

なお説明の都合上，まず重力無しの状態にしている．固定点から吊り下げていたバネが無いため，重力があると一瞬で画面外に落ちていく．

以下の変更を加えよ．

- 重力をなしにして (`gy = 0.0`)，見える範囲を広げる (`viewingSize = 4.0`).
- プログラムの `particles[]` や `springs[]` を宣言している付近に，以下の粒子配置に関連するパラメータを加える．

```
float gridSizeX = 0.2;
float gridSizeY = 0.2;

int gridNumX = 8;
int gridNumY = 8;

float gridOffsetX;
float gridOffsetY;
```

- Particle と Spring の初期化の箇所を，以下に入れ替える．

```
  gridOffsetX = -gridSizeX * (gridNumX - 1) / 2.0;
  gridOffsetY = -gridSizeY * (gridNumY - 1) / 2.0;

  nParticles = gridNumX * gridNumY;
  nSprings = gridNumX * (gridNumY - 1) + (gridNumX - 1) * gridNumY
      + 2 * (gridNumX - 1) * (gridNumY - 1);

  particles = new Particle[nParticles];
  for (int j = 0; j < gridNumY; j++) {
```

```
  for (int i = 0; i < gridNumX; i++) {
    int n = i + j * gridNumX;
    particles[n] = new Particle(gridOffsetX + i * gridSizeX,
                                gridOffsetY + j * gridSizeY,
                                0.0, 0.0, m, false, radius);
  }
}

springs = new Spring[nSprings];
int n = 0;
for (int j = 0; j < gridNumY; j++) {
  for (int i = 0; i < gridNumX - 1; i++) {
    springs[n] = new Spring(particles[i + j * gridNumX],
                            particles[(i + 1) + j * gridNumX],
                            k, gridSizeX, d);
    n++;
  }
}

for (int j = 0; j < gridNumY - 1; j++) {
  for (int i = 0; i < gridNumX; i++) {
    springs[n] = new Spring(particles[i + j * gridNumX],
                            particles[i + (j + 1) * gridNumX],
                            k, gridSizeY, d);
    n++;
  }
}

float l = sqrt(gridSizeX * gridSizeX + gridSizeY * gridSizeY);
for (int j = 0; j < gridNumY - 1; j++) {
  for (int i = 0; i < gridNumX - 1; i++) {
    springs[n] = new Spring(particles[i + j * gridNumX],
                            particles[(i + 1) + (j + 1) * gridNumX],
                            k, l, d);
    n++;
  }
}
for (int j = 0; j < gridNumY - 1; j++) {
  for (int i = 0; i < gridNumX - 1; i++) {
    springs[n] = new Spring(particles[(i + 1) + j * gridNumX],
                            particles[i + (j + 1) * gridNumX],
                            k, l, d);
    n++;
  }
}
```

■■■ 演習: 粘弾性体 ソースコードを変更し，図 8 のような結果が表示されることを確かめよ．

■■■ 演習: 重力の印加 gy に正の値を設定し，画面下方に落下させてみよ．9.8 を設定すると，たいへんな速度で画面外に落下して見ることができない．1.0 くらいで様子を見てみよ．

7.2 ペナルティ法による壁との接触

床面を作成し，接触による反力を導入する．床面から受ける力は，ペナルティ法によって計算する．めりこんだ距離に比例する反発力を，壁面から受けるものとする．

以下のソースコードを，更に加える．

- 物理パラメータ．バネに関するパラメータ (k,l0,d) の設定の後に以下を挿入．

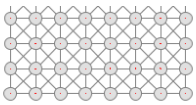

図 8　粘弾性体その 1

```
float wallY = 3.0;
float wallK = 100.0;
float wallD = 0.5;
```

wallY は壁面の位置，wallK および wallD はそれぞれ弾性および粘性の係数である．さしあたり，粒子間相互作用と同じ大きさにしている．

- 壁を表すクラス Wall のインスタンス wall を宣言．Spring[] springs の後に以下を挿入．

```
  Wall wall;
```

- 壁を表すクラス Wall の作成．Spring クラスの後に挿入．

```
// wall

class Wall {
  float y;
  float k;
  float d;

  Wall(float y0, float k0, float d0) {
    y = y0;
    k = k0;
    d = d0;
  }

  float collisionForceX(float dy, float dvx, float dvy) {
    return -d * dvx;
  }

  float collisionForceY(float dy, float dvx, float dvy) {
    return -k * dy - d * dvy;
  }

  void init() {
  }

  void calc() {
  }

  void draw() {
    fill(224);
    noStroke();
```

```
    rect(-viewingSize / 2, y, viewingSize, viewingSize);

    stroke(128);
    strokeWeightScaled(1.0);
    line(-viewingSize, y, viewingSize, y);
  }
};
```

- 壁初期化．simulationInit() の末尾に以下を挿入．

```
  wall = new Wall(wallY, wallK, wallD);
  w.init();
```

- 壁面から受ける力の計算．simulationCalc() の後ろに，以下を挿入．

```
  for (Particle p: particles) {
    float dy = (p.y + p.radius) - wall.y;
    if (dy > 0.0) {
      p.addForce(wall.collisionForceX(dy, p.vx, p.vy),
                 wall.collisionForceY(dy, p.vx, p.vy));
    }
  }
```

- 壁面の描画．simulationDraw() 内のバネ描画メソッド呼び出しの前に，以下を挿入．

```
    wall.draw();
```

よし実行! ... この物理パラメータの設定だと，たいへんおかしな挙動をする．

壁をもっと硬くしないといけない．つまり，めりこみ量が少しであっても，大きな力を返す，すなわち，wallK を大きくする必要がある．すると反力が大きくなるため，タイムステップを小さくしないと，質点のタイムステップあたりの移動量が大きくなり破綻する．

■■■ 演習: パラメータの調整 しばらくいろいろなパラメータを調整して，挙動が安定するかどうか探ってみよ．調整するパラメータは，m，radius，k，d，wallK，wallD，dt あたり．

7.3 計算と描画の分離

うまくシミュレートできない問題の根本は，タイムステップを小さくする方が安定するが，小さくすると時間経過が遅すぎてシミュレーションが進行しない点にある．

ここでタスクマネージャなどで CPU の使用状況を見てみると，実はそれほど計算に CPU を使用していない．つまり，計算能力にはまだまだ余裕がある．

しかし，描画 1 回ごとに計算 1 回というプログラムになっているため，全力で計算できていない．

スレッドを用いることで，この問題を解決する．スレッドはプログラム内で複数の処理を並列に実行するための機構である．具体的には，ある関数の処理を，現在の処理と並列に行うように，実行を分岐する．今回は計算と描画を分離したが，他にも例えば外部デバイスからの入力を受け付ける処理を並列化するなど，利用すると効率がよくなる局面は多い．

並列化は以下のように行う．

- simulationCalc() をスレッドとして実行する．この関数では一度実行されるとひたすら計算を続けるものとするので，関数内の処理全体を while (true) で囲む．また引数 \float dt を削除する．

```
void simulationCalc() {
  while (true) {
    for (Particle p: particles) {
    :
      p.move(dt);
```

```
    }
  }
}
```

- draw() から simulatoinCalc() を呼び出していたが，これを削除する．
- 新たに，setup() の末尾に，次のソースを加える．

```
thread("simulationCalc");
```

これによって，関数 simulationCalc() が新たなスレッドとして呼び出され，描画の速度と関係なく，計算するようになる．

- 安定した挙動をする物理パラメータはおおむね以下の通り．

```
float m = 0.5;

float radius = 0.05;

float k = 10000.0;
float l0 = 0.2;
float d = 100.0;

float wallY = 3.0;
float wallK = 10000.0;
float wallD = 100.0;

float gx = 0.0;
float gy = 9.8;

float dt = 3e-6;
```

タイムステップを非常に小さくできていることに注目．

■■■ 演習: 並列化 並列化処理を行い，シミュレーションの挙動を安定化させよ．またパラメータを様々に変えて，挙動や形状の変化を見てみよ．

力を表す線が鬱陶しい場合は適宜表示しないようにせよ．

7.4 インタラクション再び

挙動が安定化したので，物体のマウスによる操作を行えるようにする．ここでは質点を掴んで移動できるようにする．

以下の処置を行う．

- 6.1 節で追加したマウスクリックすると物体がその点に引き寄せられるコードを削除する．
- 物体の把持に関する以下のソースコードを，ソースコードの末尾に追加する．

```
// interaction

boolean isGrabbed = false;
Particle grabbedParticle;
float grabStartX;
float grabStartY;

void mousePressed() {
  float mx = (mouseX - xOffset) / viewingScale;
  float my = (mouseY - yOffset) / viewingScale;
  for (Particle p: particles) {
    float dx = p.x - mx;
    float dy = p.y - my;
```

```
    if (dx * dx + dy * dy < p.radius * p.radius) {
      isGrabbed = true;
      grabbedParticle = p;
      grabbedParticle.isFixed = true;
      grabStartX = mouseX;
      grabStartY = mouseY;
      break;
    }
  }
}

void mouseReleased() {
  if (isGrabbed) {
    isGrabbed = false;
    grabbedParticle.isFixed = false;
  }
}
```

- 把持している質点の色を変えるために，Particle クラスのメソッド draw() 内で，楕円を描く前にこの物体がつかまれていたら色を変えるコードを追加する．this はインスタンス自体を指す．

```
    if (isGrabbed == true && this == grabbedParticle) {
      fill(240, 128, 128);
    }
```

- 把持されている物体をマウスの位置に固定するために，以下のコードを simulationCalc() の永久ループ while (true) ブロックの冒頭に記述する．

```
    if (isGrabbed) {
      float mx = (mouseX - xOffset) / viewingScale;
      float my = (mouseY - yOffset) / viewingScale;
      grabbedParticle.x = mx;
      grabbedParticle.y = my;

      float vx = (mouseX - pmouseX) / viewingScale;
      float vy = (mouseY - pmouseY) / viewingScale;
      grabbedParticle.vx = vx;
      grabbedParticle.vy = vy;
    }
```

■■■ 演習: 質点の把持 上記の処理を行い，質点を掴んで移動できるようにせよ．

■■■ 演習: [発展] 壁面の移動 壁面を適当な振幅，適当な周期で上下運動させてみよ．

■■■ 演習: [発展] 距離センサの使用 Arduino を介して距離センサからの入力を得ることで，壁面の位置を手により制御するようにせよ．

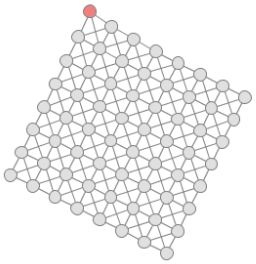

図 9　粘弾性体を掴んでいるところ