3.1 可視化実験（１）: Processingの基礎とArduinoとの連携

可視化実験（今週と来週）ではProcessingの基礎を学習する．Processing とは画像処理やアニメーションのビジュアルデザイン，インタラクションデザインなどの分野でのプログラミングに特化したオープンソースのプログラミング環境である．Java をベースにした実行環境とエディタが用意されていて，Windows，Mac OS，Linux 版ソフトが無償で提供されている．

Processingの特徴の一つはプログラミング言語の仕様や必要なライブラリといったおぜん立てに精通しなくても，とにかく簡単にグラフィックを用いたプログラムを作成できる点にある．また，Processing と Arduinoの開発環境は非常に似ており，これは Arduinoの開発環境が Processingから派生して作られていることに起因している．さらに，Processing と Arduinoは開発環境が似ているだけでなく，両者を連携させたプログラムを書くことも容易である．例えば，Arduinoの内部状態を Processingに送信し，リアルタイムで Arduinoの内部状態を Processingで可視化することができる．また，Processing に入力した情報を Arduinoに送信して，Arduinoを制御するような使用法も可能である．

実験目標 (今回と次回) :

- Processing を使った基本的な描画法を理解し，各自で基本的な描画プログラムを作成できるようになる．
- Arduino と Processing をシリアル通信で連携して情報をやり取りするための仕組みを理解し，各自が通信のためのプログラムを書けるようになる．
- Arduinoで読み取ったセンサーの値のデータなどを Processingに送信して，送信したデータをリアルタイムで可視化する方法を学ぶ．
- Processing に入力した情報を Arduinoに送信して Arduinoを操作する方法を学ぶ．

3.1.1 Processing をはじめよう

コマンドプロンプトに `processing &` （`&` はバックグラウンドで実行するため）と入力することで Processing 開発環境 (IDE) が起動する．図 3.1 に Processing IDE を示す．Processing で作成するプログラムは（Arduinoのプログラムと同様に）スケッチと呼ばれ，Processing IDE のエディタ領域にプログラムコードを記述してスケッチを作成する．スケッチの実行も Processing IDE から行う．

＜演習 3.1.1 ＞ とりあえず動かしてみよう.
以下の手続きに従ってサンプルスケッチ “MousePress” を実行せよ．

1. Processing IDE を起動：コマンドプロンプトに `processing &` と入力．
2. サンプルスケッチ “MousePress” を開く：Processing IDE のボタン [ファイル] → [サンプル] を選択．メニューウィンドウが開くので，[Basic] → [Input] → [MousePress] を選択．スケッチ MousePress が編集可能な Processing IDE が開かれる．
3. スケッチを実行：開かれた Processing IDE のボタン [実行] をクリックしてスケッチを実行．
4. 挙動の確認 : 現れたウィンドウ上でマウスポインタを動かしたり，クリックして挙動を確認．このスケッチは size(640, 360) のウィンドウ上でマウスのポインタを動かすと，ポインタを追従して十字印が描かれる．十字の色はデフォルトでは黒だが，クリックしている間は白になる．
5. 実行を停止：IDE のボタン [停止] をクリック．IDE を閉じるには左上の×印をクリック．

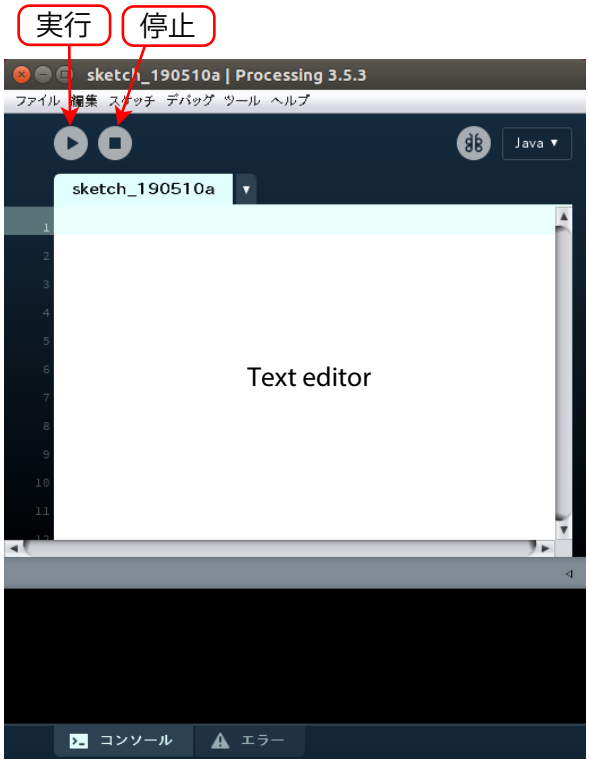

図 3.1: Processing IDE 画面

他にもさまざまなサンプルスケッチがあるのでいろいろ試してみよ．

3.1.2 Processing IDE の初期設定

Processing IDE の初期設定を行う．Processing IDE の メニュー [ファイル] → [設定] を選択し，立ち上がるポップアップウィンドウ上で図 3.2 を参考に以下の項目の設定を変更する．その前に自分のホームディレクトリに processing という名前のフォルダを作成しておく．具体的な操作内容は以下の通りである．

スケッチの保存ディレクトリ： 本演習では各自のホームディレクトリに processing というディレクトリを作成して，ここに作成したスケッチを保存することにする．そのため，Sketchbook location の項目を図 3.2 のように変更（ディレクトリ名は自分の環境に合わせる）．

日本語フォントの利用： デフォルトの設定では日本語入力ができない（かもしれない）．「複雑なテキスト入力を有効にする」の項目にチェックを入れる．また，Editor and Console font の項目で日本語フォントが使用可能なフォントを選択する（日本語でコメントを書かない場合は不要）．

フォントサイズ： 14 ポイント以上に設定．デフォルトでは 12 ポイントになっているが，アンダーバーが表示されない場合がある．14 ポイント以上にすることでこの問題が回避できる．

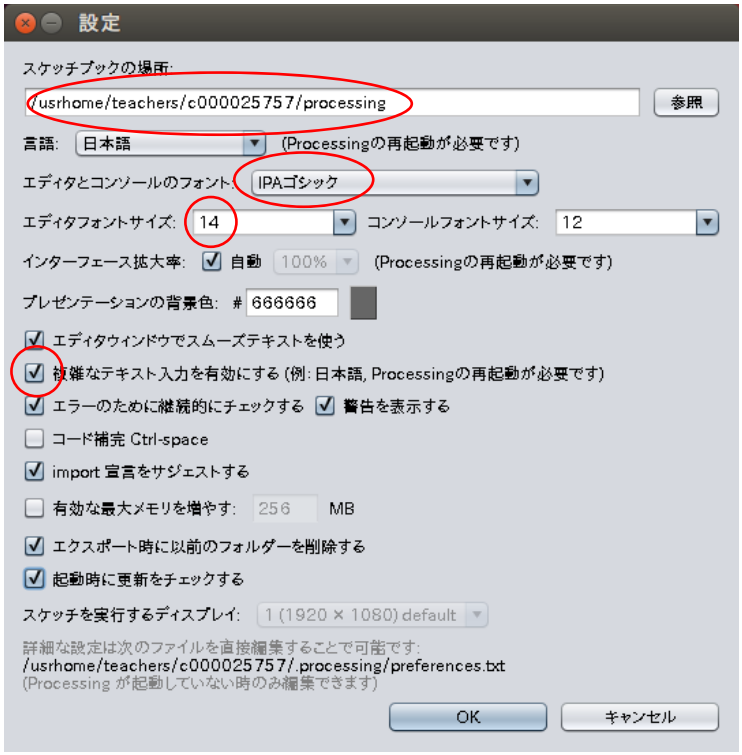

図 3.2: 初期設定の例（ユーザー名等は適宜読み替える）

3.1.3　スケッチの作成，実行，保存

スケッチを新規に作成したり，作成したスケッチを保存したりするための基本操作を解説する．図 3.1 に Processing IDE を示す．Processing IDE にはスケッチの実行と停止を行うためのボタンが配置されているが，それ以外の操作は主に File メニューから実行する．以下に基本的な操作法をまとめておく．

新規 新しい Processing IDE 画面が現れ，新規スケッチが編集可能となる．

開く 既存のスケッチを開く．ポップメニューが表示され，そこから読み込むスケッチを選択できる．

保存 スケッチを保存する．既存のスケッチを編集している場合は上書き保存される．新規スケッチを作成している場合は，適当なディレクトリを指定して名前を付けて保存．

名前を付けて保存 既存のスケッチや修正したものを別名で保存．

Processing では例えば sample という名前でスケッチを保存すると，sample という名前のフォルダが作成され，その中に sample.pde という名前のファイルが保存される．

＜演習 3.1.2 ＞ スケッチの新規作成，実行，保存

以下の手順に従いサンプルスケッチを作成，スケッチを実行して図形を描画，そして作成したスケッチを保存する一連の基本操作を実行してみよう．

1. Processing IDE を起動．

2. 新規にスケッチを作成し，IDE のテキストエディタで下記のスケッチ (sample1) を編集（コメント文は入力しなくてよい）.
3. 作成したスケッチを実行して，出力される画面を確認.
4. 作成したスケッチに sample1 という名前をつけて保存.
5. スケッチの実行を停止．次に，そのスケッチを作成した IDE を閉じる（左上の×印をクリック）.
6. 今作成したスケッチを再び開く．スケッチを適当に変更して，sample2 という名前を付けて別名保存.

Processing スケッチ: sample1

```
size(100, 120); // 幅 100 ピクセル，高さ 120 ピクセルのウィンドウを作成
ellipse(50, 50, 80, 100 ); // 中心が (50, 50)，幅 80 高さ 100 の楕円を描く
```

3.1.4 基本的な描画

Processing では描画を行うために図形の位置を**ピクセル座標**で表す．ピクセル座標の 1 はコンピュータ画面の 1 ピクセルに対応する．ピクセル座標は (x,y) のように表し，画面サイズが 300 × 200 であれば，左上隅が (0,0) で右下隅が (299, 199) となる．y 座標については画面上から下に向かってピクセル座標の値が増加することに注意が必要である．図 3.3 にピクセル座標系の例を示す．特に断り書きがない場合，ピクセル座標のことを単に座標と呼ぶことにする．

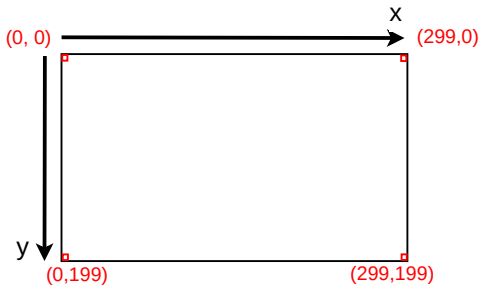

図 3.3: サイズ 300 × 200 のピクセル座標系

通常の Processing の利用法ではスケッチは**連続モード**（3.1.7 節参照）で記述される．ただし，静止画像を描くだけであれば**基本モード**と呼ばれる命令を並べただけのシンプルなスタイルでスケッチを記述することができる．静止画像を描く演習では基本モードでスケッチを記述することにする．

基本モードで記述さたスケッチの例を下記のスケッチ (example1) に，これを実行して出力されるウィンドウを図 3.4 示す.

Processing スケッチ: example1

```
size(500, 400);  // 幅 500，高さ 400 のウィンドウ生成
background(255); // 背景を白
strokeWeight(5); // 線の太さを 5 ピクセル
line(50, 50, 150, 100);   // (50,50),(150,100) を結ぶ線
rect(250, 200, 150, 100); // (250,250) を左上端，幅 150，高さ 100 の長方形
stroke(255, 0, 0);        // 線の色を赤
ellipse(100, 250, 100, 150); // (100,300) を中心幅 100，高さ 150 の楕円
```

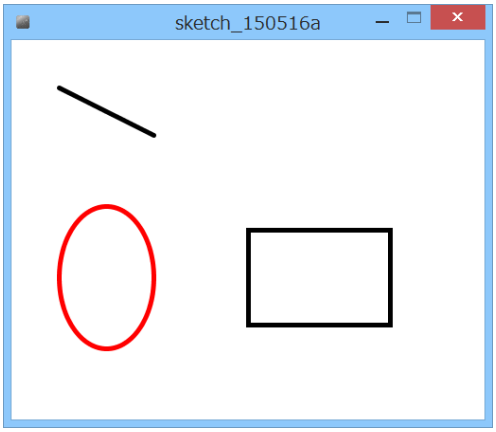

図 3.4: スケッチ (example1) で出力されるウィンドウ

以下に図形を描画するための基本的な関数を紹介する（図 3.5 も参照せよ）．これらの関数を使うだけでもたいていの図形を描くことが可能である．それ以外の関数を調べたいときは本資料の最後に挙げたリファレンスサイト等を参考にせよ．デフォルトでは，ウィンドウの背景は明灰色，線の太さは 1 ピクセルで色は黒，図形内部の色は白で塗りつぶされる．下記の関数の引数は整数で与えるのが基本であるが，浮動小数点でも問題ない．ただし，size() 関数の引数だけは整数でなくてはならない．

描画のための基本的な関数

size(width, height) ：幅が size，高さが height のサイズのウィンドウを生成．

point(x, y) ：座標 (x,y) 上に点を描く．

line(x1, y1, x2, y2) ：2 つの座標 (x1, y1) と (x2, y2) の間に線を描く．

triangle(x1, y1, x2, y2, x3, y3) ：3 つの座標 (x1, y1), (x2, y2), (x3, y3) を頂点とする三角形を描く．

quad(x1, y1, x2, y2, x3, y3, x4, y4) ：4 つの座標 (x1, y1), (x2, y2), (x3, y3), (x4, y4) を頂点とする四角形を描く．

rect(x, y, width, height) ：幅が width，高さが height の長方形を描く．ただし，長方形の左上隅の座標が (x, y)．

ellipse(x, y, width, height) ：幅が width，高さが height の楕円を描く．ただし，楕円の中心座標が (x, y)．

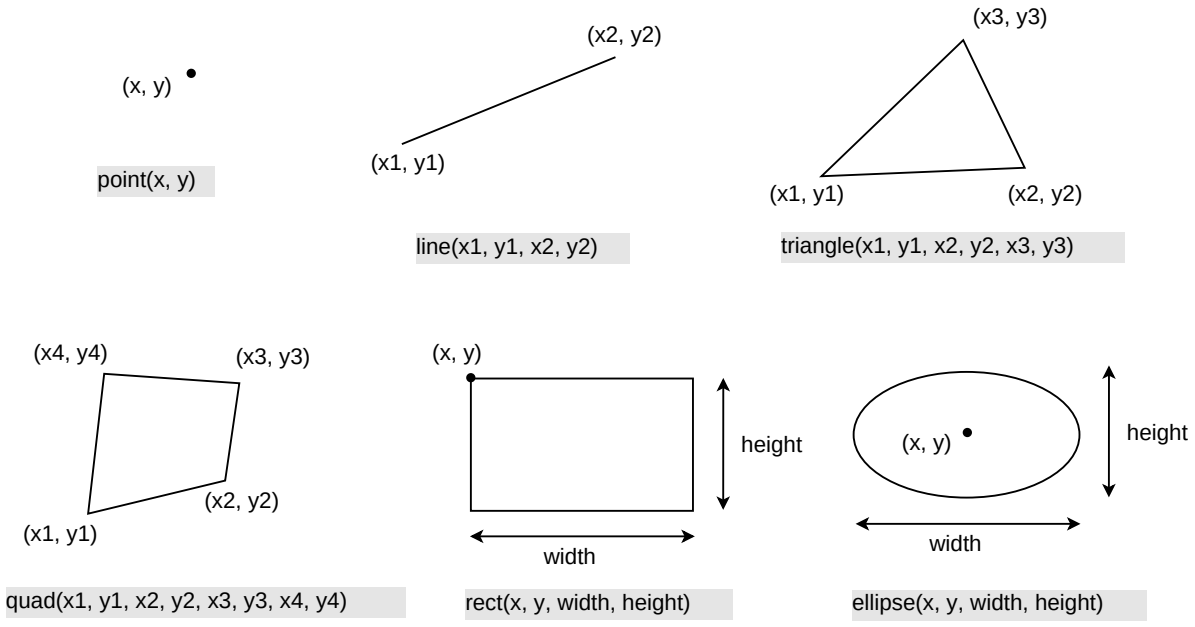

図 3.5: 基本図形を描画する関数

Processing では Arduino と同じく print() 関数と println() 関数が用意されており，引数として与えた値（数値や文字列など）はコンソールに出力される．これらの関数はプログラムのバグを発見するための有用な情報を与えてくれるので，以下のような使い方 (example2) を知っておくと良い．print() 関数と println() 関数の違いは改行なし/ありの差のみである．

Processing スケッチ: example2

```
int a = 10;
int b = 20;
println(a);
println(a,b);
println("a =", a );
println("a =", a, "b =", b );
```

表示結果

```
10
10 20
a = 10
a = 10 b = 20
```

＜演習 3.1.3＞ 基本図形の描画

サイズ 600 × 400 のウィンドウ上に，図 3.6 に描かれた 4 つの図形を描画する（格子は描かなくてよい）スケッチを作成して実行せよ．作成したスケッチを draw_basic1 という名前で保存せよ．

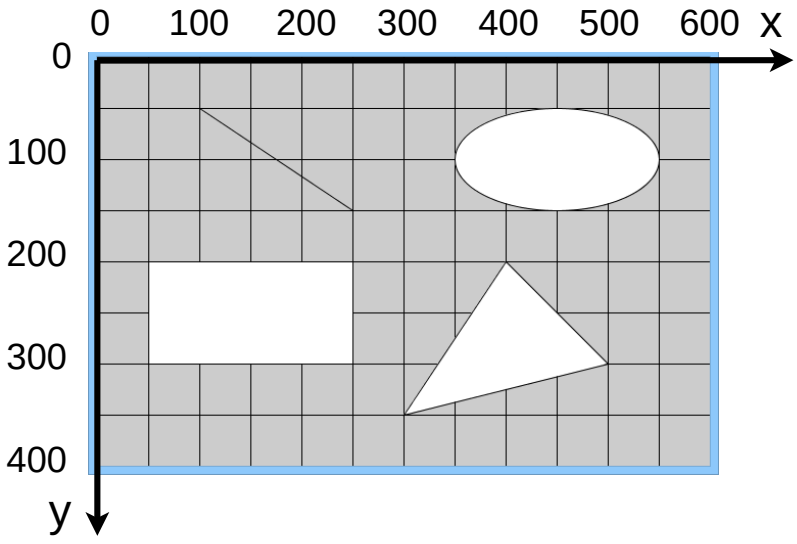

図 3.6: サイズ 600 × 400 のウィンドウに描いた 4 つの図形（格子は参考）．

3.1.5　線や図形の属性の変更

Processing では図形の属性のデフォルト値は次のようになっている：ウィンドウの背景=明灰色，線の太さ=1 ピクセル，線色=黒，図形内部の色=白．これらの属性値を変更するための基本的な関数を紹介する．属性の変更はその後に描けれる全ての図形に適用される．

background() 関数：
書式：background(gray), background(r, g, b)
引数：gray, r, g, b は整数または浮動小数点
機能：ウィンドウ背景を指定の色で塗りつぶす．色は引数が 3 つの場合は RGB 値を順に指定．引数が 1 つの場合はグレースケールで指定．値は 0（黒）から 255（白）の範囲で指定．

strokeWeight() 関数：
書式：strokeWeight(w)
引数：w は整数または浮動小数点
機能：線の太さを w ピクセルに設定

stroke() 関数：
書式：stroke(gray), stroke(r, g, b)
引数： gray, r, g, b は整数または浮動小数点
機能：線の色を指定．色の指定法は background() 関数と同じ．

fill() 関数：
書式： fill(gray), fill(r, g, b)
引数： gray, r, g, b は整数または浮動小数点
機能：図形内部の色を指定．色の指定法は background() 関数と同じ．

noFill() 関数：
書式：noFill()
引数： なし
機能：図形内部の塗りを無効化

fill() 関数や stroke() 関数に 4 つ目の引数（グレースケールで色を指定している場合は 2 つ目）を指定することで，透明度を設定することが出来る．このパラメータをアルファ値と呼ぶ．アルファ値を 0 にすると完全な透明，255 の場合は完全な不透明となる．

＜演習 3.1.4 ＞ 図形の属性変更
演習 3.1.3 で作成したスケッチ (draw_basic1) で描かれる図形（図 3.6）の属性を変更して，図 3.7 に描かれている図形を描画するスケッチを作成して実行せよ．作成したスケッチを draw_basic2 という名前で保存せよ[1]．

[1]この作業は draw_basic1 を直接編集して draw_basic2 に別名保存すればよいが，誤って draw_basic1 に上書き保存してしまう危険がある．従って，初めに draw_basic1 を draw_basic2 に別名保存してから作業した方が良い（今後，同じような状況が何度もある）．

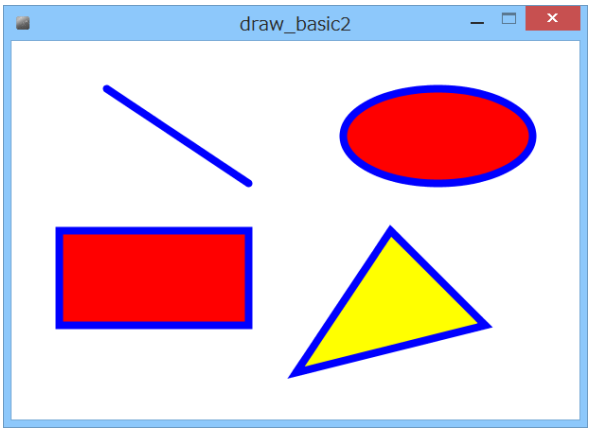

図 3.7: 図 3.6 の図形の属性を変更した例：背景=白 (gray=255)，線の太さ=8，線の色=青 (r=0,g=0,b=255)，図形内部の色=赤 (r=255,g=0,b=0) or 黄 (r=255,g=255,b=0).

3.1.6 基本的なプログラム

Processing は Java や C 言語とほぼ同じ文法でプログラムを書くことができる．例えば，変数の使い方，if 文による条件分岐，for ループによる繰り返し処理等は Java や C 言語の文法と同じである．本演習ではプログラムの文法の一般的な説明はしないが，いくつかのスケッチの例を通してこれらの理解を深めることにする．

Processing に特有の**システム変数**（あらかじめ定義されている変数）にはさまざまなものがあり，特に重要なものは適宜説明していく．下記のシステム変数 width と height もその一例である．

width：システム変数．ウィンドウの幅を保持する．size() 関数を実行したときにセットされる．
height：システム変数．ウィンドウの高さを保持する．size() 関数を実行したときにセットされる．

システム変数 width と height を使うことで，ウィンドウの大きさに対して相対的に図形の位置を指定することができる．例えば，図 3.8 のような画像を作成したとしたとして，その後ウィンドウサイズを変更しても，ウィンドウの伸縮に応じて図形も伸縮するようにプログラミングすることができる．

＜演習 3.1.5 ＞ 繰り返し処理による描画

図 3.8 のような画像を出力したい．すなわち，サイズが 800 × 120 のウィンドウを作成し，このウィンドウを横方向に均等に xn 分割（パラメータ）するように格子状の枠線を引く．さらに，全てのマス目ごとに内接する楕円を描く．色は背景は白で，枠線が黒，楕円が赤（内部は白）である．以下の手順で画像を描画せよ．

1. 以下に記述されているスケッチ (draw_latice_1D) を作成する．途中，格子状の枠線だけを描いた段階で描画を行い，結果を確認せよ．
2. スケッチを最後まで作成して結果を確認せよ．
3. ウィンドウサイズを (800,120) から適当な値に変更して描画してみよ．ウィンドウサイズが変わっても，ウィンドウが均等に分割されていることを確認せよ．
4. 分割数 xn を適当に変化させても，分割数が変化するだけでウィンドウが均等に分割されていることを確認せよ．
5. 作成したスケッチを draw_latice_1D という名前のファイルに保存せよ．

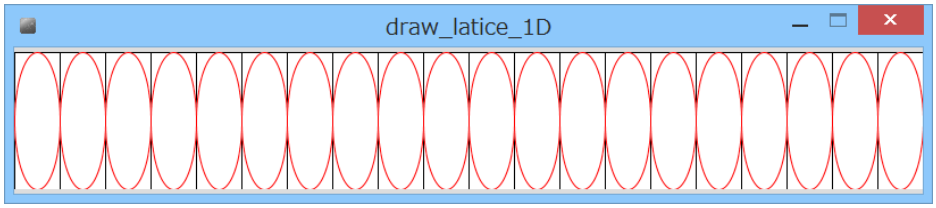

図 3.8: サイズ (800,120) のウィンドウを横方向に均等に xn(=20) 分割するよう格子上の枠線を引き，各マス目に接する楕円を描いた例.

スケッチ: draw_latice_1D

```
int xn = 20; // 分割数
int x;

size(800, 120);
background(255); // 背景を白

stroke(0); // 線の色を黒
line(0, 0, width, 0);            // 格子の横線を引く
line(0, height, width, height); // 格子の横線を引く

for ( int i = 0; i <= xn; i++ ) { // 格子の縦線を引く
  x = i* width/xn;  // 線の x 座標
  line(x, 0, x, height);
}

stroke(255, 0, 0); // 線の色を赤
for ( int i_c = 0; i_c < xn; i_c++ ) { // i_c はマス目のインデックス
  x = width/(2*xn) + i_c*width/xn; // 楕円の中心の x 座標（y 座標は height/2）
  ellipse(x, height/2, width/xn, height); // マス目内に楕円を描く
}
```

＜課題 3.1.1 ＞ 繰り返し処理による描画（2D バージョン）

図 3.9 のような画像を出力したい．すなわち，サイズが (600,600) のウィンドウを作成し，このウィンドウを横方向に均等に xn 分割（パラメータ），縦方向に yn 分割（パラメータ）するように格子状の枠線を引く．さらに，全てのマス目ごとに内接する楕円を描く．色は背景は白で，枠線が黒，楕円が赤（内部は白）である．作成したスケッチを draw_latice_2D という名前で保存せよ．

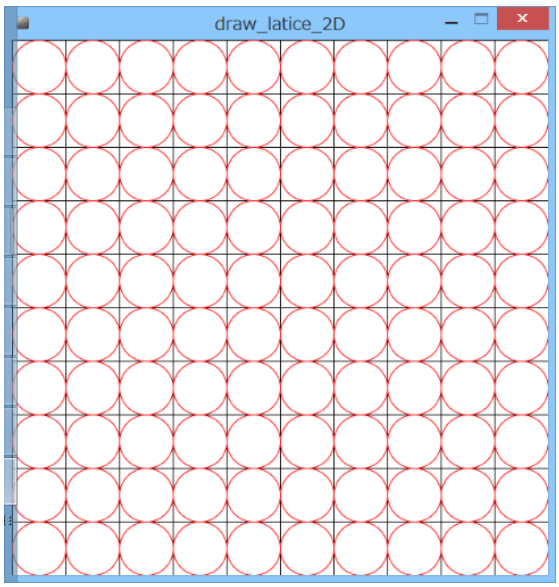

図 3.9: サイズ (600,600) のウィンドウを横方向に均等に xn(=10) 分割，縦方向に yn(=10) 分割するように格子状の枠線を引き，各マス目に接する楕円を描いた例．

3.1.7　連続モードのスケッチ

これまでの演習では基本モードでスケッチを作成したが（静止画像を描くだけならこれで十分），通常，Processing は連続モードと呼ばれるスタイルでスケッチを作成する．連続モードのスケッチの基本構造を以下に示す．連続モードのスケッチでは setup() 関数と draw() 関数を記述する．setup() 関数は初期設定を行う関数で，スケッチが実行されると最初に一回だけ関数内のプログラム内容が実行される．その後は一定時間間隔で draw() 関数内のプログラム内容が繰り返される．したがって，draw() 関数には繰り返し実行したい処理内容を記述する．draw() 関数の無限ループは Processing IDE でスケッチの実行を Stop するか Processing IDE を閉じるまで継続される．

連続モードのスケッチの基本構造

```
// 最初に 1 度だけ実行される
void setup()
{
   // 初期設定など
}

// 関数内部の処理が繰り返される
void draw()
{
  // 1 フレームの処理内容
}
```

連続モードでは draw() 内の処理を繰り返すが，これを 1 回実行することを 1 フレームと呼ぶ．1 フレームごとに 1 ショットの描画内容をプログラムすることで動画が作成される．1 秒間に draw() 内の処理が繰り返される回数をフレームレートと呼ぶ．フレームレートは何も指定しなければ 60 に設定されているが，frameRate() 関数で変更可能である[2]．

＜演習 3.1.6 ＞ 動画の作成: 移動する楕円

演習 3.1.5 で作成したスケッチ (draw_latice_1D) で描かれる図形（図 3.8 参照）において，楕円を全てのマス目に描くのではなく，1 つの楕円が左から右に移動する動画を作成する．ただし，右端に到達した後は左端に戻るものとする．以下の手順で動画を描画せよ．

1. 演習 3.1.5 で作成したスケッチ (draw_latice_1D) を開き，別名のスケッチ (draw_laticeMove_1D) として保存する．
2. 別名保存したスケッチを編集して，下記スケッチ (draw_laticeMove_1D）を作成．二つのスケッチ draw_latice_1D と draw_laticeMove_1D の差異を良く考えながらスケッチを編集すること（単にコピペしないこと）．
3. 作成したスケッチを実行して挙動を確認せよ．
4. フレームレートの値をデフォルト値 (60) から適当に変更して（100, 40, 5 など）挙動を比較せよ．フレームレートが大きい場合は楕円の線を太くすると良いかもしれない．
5. 分割数 xn を適当に変化させて，挙動を確認せよ．

[2]draw() 関数内の処理が重い場合，指定したフレームレートを実現できない場合もある．

Processing : draw_laticeMove_1D

```
int xn = 20; // 分割数
int i_c;     // 楕円を描くマス目番号

void setup(){
  size(800, 120);
  // frameRate(20); // フレームレートのデフォルト値は 60
  i_c = 0;
}

void draw()
{
  int x;

  background(255); // 背景を白
  stroke(0); // 線の色を黒
  line(0, 0, width, 0);            // 格子の横線を描く
  line(0, height, width, height); // 格子の横線を描く

  for ( int i = 0; i <= xn; i++ ) { // 格子の縦線を描く
    x = i* width/xn;  // 線の x 座標
    line(x, 0, x, height);
  }

  // 楕円を描くマス目番号の更新
  i_c++;
  if ( i_c == xn )
    i_c = 0;

  stroke(255, 0, 0); // 線の色を赤
  x = width/(2*xn) + i_c*width/xn; // 楕円の中心の x 座標（y 座標は height/2）
  ellipse(x, height/2, width/xn, height); // マス目内に楕円を描く
}
```

＜課題 3.1.2 ＞ 動画の作成: 移動する楕円（2D バージョン）

課題 3.1.1 で作成したスケッチ (draw_latice_2D) で描かれる図形（図 3.9 参照）において，楕円を全てのマス目に描くのではなく，1 つの楕円が左から右に移動する動画を作成する．ただし，楕円が右端に到達した後は一段下の段の左端に移動し，右下端に達した後は左上端に移動するものとする．作成したスケッチを draw_laticeMove_2D という名前で保存せよ．

3.1.8 Arduino から Processing へのデータ送信（シリアル通信による連携）

Processing の利点の一つは Arduino とシリアル通信で連携して Ardino が出力するデータを簡単に可視化できることにある．基礎実験（第 5 回）の演習では Arduino から PC へシリアル通信でデータを送信し，送信されたデータをシリアルモニタで確認した．Processing を使うと Arduino からシリアル通信で送られてくるデータをリアルタイムに可視化することができる．

Arduino でシリアル通信を行う方法については基礎実験（第 5 回目）で解説した．Processing でシリアル通信を行う場合もほぼ同様の文法に従ってデータの送受信が可能である．以下に Processing でシリアル通信を行う際に最も頻繁に使用される Serial クラスの関数をいくつか紹介する（その他に必要な関数は適宜解説する）．Processing の Serial クラスで利用可能な関数の詳細は Processing のリファレンスサイト（英語）https://processing.org/reference/libraries/serial/ などを参照されたい．

read() 関数：
書式： port.read() ※ port（任意の名前）は Serial クラスのインスタンス
引数： なし
機能：シリアルポートの受信バッファから（もっとも過去に受信した）データを 1 バイト読み込む．
戻り値：読み込み可能なデータの最初の 1 バイトのデータ．

write() 関数：
書式：port.write(val) ※ port（任意の名前）は Serial クラスのインスタンス
引数：val は 1 バイトのデータ（文字列を送ることもできるが説明省略）
機能：シリアルポートに引数で指定されるデータ（1 バイト）を出力．
戻り値：送信したバイト数

clear() 関数：
書式： port.clear() ※ port（任意の名前）は Serial クラスのインスタンス
引数： なし
機能：シリアルポートの受信バッファのデータをクリアする．
戻り値：なし

available() 関数：
書式：port.available() ※ port（任意の名前）は Serial クラスのインスタンス
引数：なし
機能：シリアルポートの受信バッファに新しいデータが到着しているかどうかを確認する．
戻り値：シリアルポートの受信バッファにあるデータのバイト数

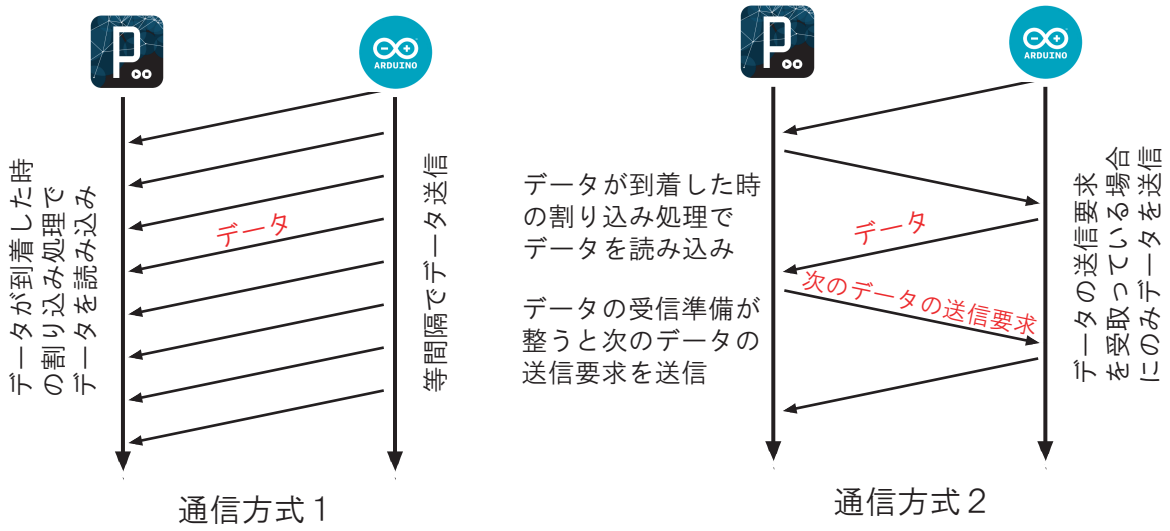

図 3.10: Arduino から Processing にデータを送信するための 2 つの通信方式

Arduino から Processing にシリアル通信でデータを送信することを考えよう．Arduino から Processing にシリアル通信でデータを送信するための方式として，以下の 2 つ通信方式を紹介する．またそれらの通信方式の概念図を図 3.10 に示す．

通信方式 1 Arduino からは（Processing の都合は考えずに）定期的にデータを送信する．Processing ではシリアルポートにデータが到着したことをトリガーとする割り込み処理によってデータを読み込む．

通信方式 2（ハンドシェイク） Processing は次のデータを受信する準備が整うと，データの送信要求を Arduino に送信する．Arduino は Processing から送られてくる送信要求を受け取ると次のデータを送信する．

通信方式 1 は等間隔にデータの送受信を行うための最も単純な方法である．しかし，一般に，機器 A から機器 B にデータを送信する場合，受信側 (B) が受信状態でない（あるいはデータの受信はできてもそれを処理する準備が出来ていない）状況で送信側 (A) からデータを送信するとデータが失われる可能性がある．例えば，通信方式 1 では Processing の 1 フレームの間隔が Arduino のデータ送信間隔よりも長い場合，Processing は 1 フレームの間に複数回データを読み込むことになり，Arduino から送信されたデータの一部は利用されない[3]．このような問題が起きないようにデータの送受信を行うためには，お互いの状態を確認することが重要となる．通信方式 2 では受信側 (B) はデータの受信準備が整ったことを送信側 (A) に伝え，送信側 (A) はデータの送信要求を受けてからデータを受信側 (B) に送ることによりデータがロスすることを防いでいる．このような通信方式をまず握手してからデータを送信するということをイメージしてハンドシェイクと呼ぶ．

[3]単に Processing で描画を行うという目的では，Processing は 1 フレームの間に複数回データを受け取っても，その中の任意の 1 つのデータに基づいて描画を行えば良いので特に問題はない．

3.1.9 Arduino から Processing にデータを送り可視化する：通信方式 1

Arduino から Processing にデータを送り，Arduino の状態をリアルタイムで Processing で可視化する例として，以下の単純なタスク（タスク 1）を考えてみよう．

タスク 1 ：電圧値情報の可視化

- 図 3.11 に示す回路を Arduino とブレッドボードを用いて構成し，半固定抵抗を回して変化する点 S の電圧値を Arduino の A0 ポートで観測．
- Arduino で観測した点 S の電圧値（を AD 変換した値を）を Processing に送信し，この値を直径とする円として電圧値をリアルタイムに可視化．
- ウィンドウサイズは (256,256) とし，ウィンドウの中心を円の中心とする．円の直径は電圧値が 0V 時 0，電圧値が 5V の時 256 となるように表示．

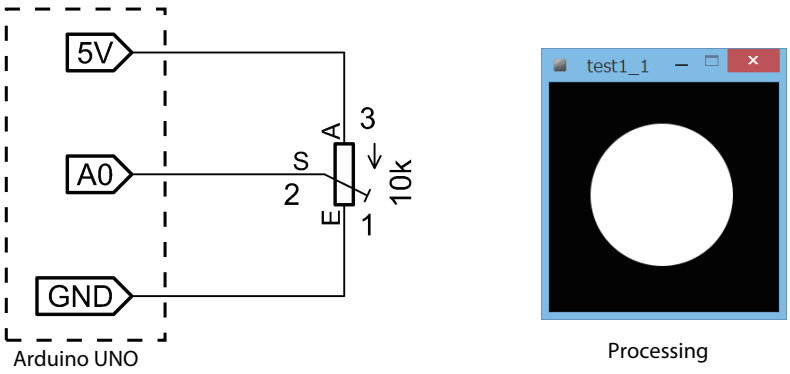

図 3.11: 半固定抵抗により変化する S 点の電圧値を Arduino で観測．Processing では観測した電圧値をこの値を直径とする円としてリアルタイムに可視化．

通信方式 1 でタスク 1 を実現するための Arduino と Processing のスケッチを Arduino スケッチ (serial_method1_1byte) と Processing スケッチ (serial_method1_1byte_circle) に示す．このスケッチでは等間隔 (50ms) でシリアル通信でデータ送信している．本来であれば，等間隔でデータ送信したい場合，タイマー割り込みを用いて割り込みサービスルーチン内でデータ送信するのが自然である．しかし，タイマー割り込み中にシリアル通信を行うとバグが発生することが報告されているので，下記のスケッチではポーリング処理で 50ms 間隔の送信を行っている．そのために必要な関数として millis() 関数の仕様を以下に紹介しておく．

millis() 関数：
書式：millis()
引数：なし
機能：実行中のスケッチがスタートしてらかの経過時間を計測
戻り値：unsigned long 型の値でスケッチがスタートしてからの経過時間をミリ秒単位で出力
※ unsigned long 型は 4 バイトの符号なし整数（0〜4,294,967,295）

Arduino スケッチ: serial_method1_1byte

```
int sensorValue, sendValue;
unsigned long int timePrev, timeNow;

void setup()
{
  Serial.begin(9600); // シリアル通信を 9600bps で初期化
  timePrev = millis(); // 経過時間の初期値
}

void loop()
{
  timeNow = millis(); // 現在の経過時間
  sensorValue = analogRead(0); // A0 ピンの AD 変換結果を取得 (0-1023)

  if ( timeNow - timePrev >= 50 ) { // 50ms ごとに実行
    sendValue = sensorValue/4; // 0-255 の値に変換
    Serial.write(sendValue); // 1 バイトのバイナリデータとして値を送信
    timePrev = timeNow; // 1 ステップ前の経過時間を更新
  }
}
```

Arduino のスケッチの概要はコメントを見れば理解できると思うが，簡単にまとめると以下のようになる．

- loop() 関数では常時 A0 ポートから電圧値 (0-5 V) を AD 変換した値 (0-1023) を読み込む．
- loop() 関数では常時現在の経過時間（timeNow）を計測し，前回通信を行った経過時間（timePrev）との差が 50 (ms) を越えたら再度データ送信．loop() 関数の 1 イテレーションは 50ms より十分短いので（数 ms 程度），ほぼ 50ms ごとにデータ送信することになる．
- データ送信は，電圧値を AD 変換した値 (0-1023) を 1 バイトで送信できる値 (0〜255) に変換して送信．

Processing スケッチ: serial_method1_1byte_circle

```
import processing.serial.*; // Serial ライブラリを取り込む
Serial port; // Serial クラスのオブジェクトを宣言
int val;

void setup()
{
  size(256, 256); // サイズ 256 × 256 のウィンドウ生成
  port = new Serial(this, "/dev/ttyACM0", 9600);//Serial クラスのインスタンス生成
}

void draw()
{
  background(0); // 背景を黒
  ellipse(125, 125, val, val); // 中心 (125,125)，直径 val の円（白塗り）を描画
  println("R");  // 描画タイミング（確認用）
}

// シリアルポートにデータが到着するたびに呼び出される割り込み関数
void serialEvent(Serial p) { // p にはデータが到着したシリアルポートに対応するインスタンス（ここでは port）が代入される
  val = p.read();  // 受信バッファから 1 バイト読み込み
  println("<-");   // データ受信タイミング（確認用）
}
```

Processing のスケッチの概要は以下の通りである．

- デフォルトのフレームレート 60（1 秒に 60 回）で draw() 関数内の処理が繰り返される．1 フレームごとにウィンドウを黒でクリアしてから，中心が (125,125) で直径 val の円を表示．
- シリアルポートにデータが到着するたびに serialEvent() 関数が呼び出され，受信バッファからデータを 1 つ（1 バイト）読み込んでデータ値を int 型の変数 val に代入．
- 描画とデータ受信のタイミングを観測する目的で，描画を行ったタイミングで “R” を，データを受信したタイミングで “←” をコンソールに出力．

serialEvent() 関数はシリアルポートにデータが到着すると関数内のコードを実行する割り込み関数である．serialEvent() 関数の仕様は以下の通りである．

serialEvent() 関数：
書式：serialEvent(Serial p)
引数：p は Serial クラスのオブジェクト（任意の名前）
機能：受信バッファにデータが到着するたびに割り込み処理が行われ，関数内部のプログラムが実行される．引数の p にはデータが到着したポートに対応するインスタンスが代入される．
戻り値：なし

上記の Processing のスケッチではシリアル通信を行うための設定がいくつか行われているが，それらの内容について解説する．Processing でシリアル通信を利用する際には同様の設定を常に記述することになる（機械的にこの通り設定すれば良い）．

- Processing でシリアル通信を行う場合，Serial ライブラリを取り込む (import) 必要があるが，import processing.serial.*; という 1 文でこれを行っている．
- Serial port; という一文で port という名前の Serial クラスのオブジェクトを宣言している．
- setup() 関数内の port = new Serial(this, "/dev/ttyACM0", 9600); という一文で，Processing で利用するシリアルポート ("/dev/ttyACM0") と転送速度 (9600) を指定して，シリアルポートを利用するための Serial クラスのインスタンスを生成している．ここで，シリアルポートの名前は Arduino が接続されているシリアルポート名と同じでなければならにいことに注意する（Arduino IDE の [Tools] → [Serial Port] を選択して確認）．転送速度についても，Arduino の設定と同じにする必要がある（通常 9600）．
- 上記の設定を理解するためにはオブジェクト指向言語の知識が必要となるが，要は上記の設定をすることで，port という名前で指定したシリアルポートを利用することができるようになる．例えば，port.read() という一文で port が指定するシリアルポートの受信バッファからデータを 1 バイト読み込むことができる．

＜演習 3.1.7＞ Arduino から Processing へのデータ送信：通信方式 1

タスク 1 を通信方式 1 で実現する Arduino スケッチと Processing スケッチを作成し，これを実行して挙動を分析する．基本手順を以下に示す．

1. 図 3.11 に示す回路を Arduino とブレッドボードを用いて構成する．
2. Arduino スケッチ (serial_method1_1byte) と Processing スケッチ (serial_method1_1byte_circle) を作成し，この名前で保存する．
3. Arduino を PC に接続し Arduino スケッチを書き込む．次に Processing スケッチを実行して電圧値のデータを可視化する．Processing を実行した状態では Arduino にスケッチを書き込めないので，Arduino のスケッチを書き込む時には，Processing は実行していないこと（IDE は開いていても良い）．
4. 半固定抵抗を回してリアルタイムに正しく可視化が実現できることを確認．
5. Arduino スケッチ内のデータ送信間隔を 50 から 200, 16 ($\approx \frac{1}{60}$), 5 と変化してみて，Processing が描画するタイミングとデータを受信するタイミングを観測せよ（コンソールに出力される "R" と "←" を確認）．ここで，Processing のフレームレートはデフォルト値の 60（1 秒間に 60 回）であることにも注意する．

注意：Processing でシリアル通信を使用したスケッチを実行中はシリアルケーブルを絶対に抜かないこと．同様に Arduino でシリアル通信を使用している時にも（シリアルモニタを見ているときなど）シリアルケーブルを絶対に抜かないこと．これらの事をするとシリアルポートが使用不能になることがあります．

3.1.10 Arduino から Processing にデータを送り可視化する：通信方式 2

通信方式 2 でタスク 1 を実現するための Arduino と Processing のスケッチを以下の Arduino スケッチ (serial_method2_1byte) と Processing スケッチ (serial_method2_1byte_circle) に示す.

Arduino スケッチ: serial_method2_1byte

```
int sensorValue, sendValue;
int inByte; // Arduino からの送信要求を受け取る変数

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  sensorValue = analogRead(0); // A0 ピンの電圧値の AD 変換結果を取得 (0-1023)

  if (Serial.available() > 0){ // 送信要求を受け取った（受信バッファにデータあり）
    inByte = Serial.read();  // 受信済みの信号を読み込む（受信バッファが空になる）
    sendValue = sensorValue/4; // 0-255 の値に変換
    Serial.write(sendValue); // 1 バイトのバイナリデータとしてデータ値を送信
  }
}
```

Arduino のスケッチの概要.

- A0 ピンで電圧値を観測.
- ポーリング処理で Arduino のシリアルポートの受信バッファをチェックして，データを受信していれば（Processing からのデータ送信要求を受け取ったので），電圧の観測値を 0-255 に AD 変換した値をシリアルポートに送信．この際，受信バッファからデータを読み込みバッファを空にして，次に Processing から送られてくるデータの受信に備える.
- Arduino では loop 関数内の処理が単純であれば，loop 関数内の処理は数ミリ秒間隔で繰り返されているので，データ送信要求を受け取ると即座に（数ミリ秒以内に）データを送信する.

Processing スケッチ: serial_method2_1byte_circle

```
import processing.serial.*;
Serial port;
int val;

void setup()
{
  size(256, 256);
  port = new Serial(this, "/dev/ttyACM0", 9600);
}

void draw()
{
  background(0);
  ellipse(125, 125, val, val);
  port.write('A');  // 次のデータ送信要求（任意の 1 バイト）を送信
  println("R");     // 描画タイミング（確認用）
}

// シリアルポートにデータが到着するたびに呼び出される割り込み関数
void serialEvent(Serial p) {
  val = p.read();
  println("<-");    // データ受信タイミング（確認用）
}
```

Processing のスケッチの概要.

- デフォルトのフレームレート 60（1 秒に 60 回）で draw() 関数内の処理が繰り返される．1 フレームごとにウィンドウを黒でクリアしてから，中心が (125,125) で直径 val の円を表示．1 フレームごとに次のデータ送信を要求（任意を 1 バイト）を Arduino に送信.
- シリアルポートにデータが到着するたびに serialEvent() 関数が呼び出され，受信バッファからデータを 1 つ（1 バイト）読み込んでデータ値 (val) を更新する（val は最新のデータ値を表す変数）.
- 描画とデータ受信のタイミングを観測する目的で，描画を行ったタイミングで “R” を，データを受信したタイミングで “←” をコンソールに出力.

＜演習 3.1.8＞ Arduino から Processing へのデータ送信：通信方式 2

タスク 1 を通信方式 2 で実現する Arduino スケッチと Processing スケッチを作成し，これを実行して挙動を分析する．基本手順を以下に示す.

1. 図 3.11 に示す回路を Arduino とブレッドボードを用いて構成する（構成済み）.
2. Arduino スケッチ (serial_method2_1byte) と Processing スケッチ (serial_method2_1byte_circle) を作成し，この名前で保存せよ.
3. Arduino を PC に接続し Arduino スケッチを書き込む．次に Processing スケッチを実行して電圧値のデータを可視化する．Processing を実行した状態では，Arduino にスケッチを書き込めないので注意すること.
4. 半固定抵抗を回してリアルタイムに正しく可視化が実現できることを確認.
5. Processing が描画するタイミングとデータを受信するタイミングを観測せよ（コンソールに出力される “R” と “←” を確認）.

3.1.11 時間軸グラフによる可視化

＜課題 3.1.3＞ 半固定抵抗値による電圧変化の時間軸グラフによる可視化

3.1.9 節では，図 3.11 に示す回路を Arduino とブレッドボードを用いて構成し，半固定抵抗を回して変化する点 S の電圧値を Arduino の A0 ポートで観測し，その値を円の直径で表示する可視化を行った（タスク 1）．本課題では点 S の電圧値を図 3.12 に示すような時間軸グラフ（横軸を時間として描くグラフ）としてデータを可視化する．ただし，本課題では簡略化のため，データの表示回数（1 フレームの表示回数）を時間とみなすことにする（本来は時間のデータも送信する必要がある）．具合的には次の仕様を満たすよう可視化を行う.

- 図 3.11 に示す回路を Arduino とブレッドボードを用いて構成し（構成済み），半固定抵抗を回して変化する点 S の電圧値を Arduino の A0 ポートで観測．観測データを通信方式 1 に従ってシリアル通信で Processing に送信．作成済みの Arduino スケッチ (serial_method1_1byte) をそのまま使用すれば良い.
- Processing では送られてくるデータを図 3.12 に示すような時間-電圧グラフとして可視化する．ただし，ウィンドウサイズは 800 × 300．x 座標はデータの表示回数（1 フレームごとに 1 増加），y 座標は送られてくるデータの値が 0（電圧 0V）の時 0，255（電圧 5V）の時 300 (=height) となるように変換せよ.
- x 座標が振り切れた場合は，画面をクリアして再度 x=0 の位置からグラフを描く.
- その他の設定（背景色，点の形や大きさ）は各自の自由とする.

次の課題を行う.

1. 上記の仕様を満たす Processing スケッチを作成し，serial_method1_1byte_graph1 という名前で保存せよ.
2. Arduino スケッチ内のデータ送信間隔を 10, 50, 200 と変化させて可視化の変化を考察せよ．Processing のフレームレートはデフォルトの 60 とする.

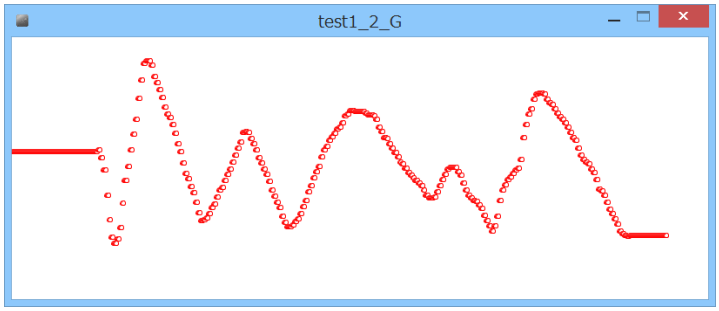

図 3.12: 時間-電圧グラフの例

＜課題 3.1.4＞ 光センサによる電圧変化の時間軸グラフによる可視化

図 3.13 に示す回路を Arduino とブレッドボードを用いて構成し，光センサ（照射光が強いほど電気抵抗が減少する素子，極性なし）にあたる光の量によって変化する点 S の電圧変化を課題 3.1.3 と同様に時間-電圧グラフとして可視化することを考える．可視化の仕様は課題 3.1.3 と同じとする（後で少し変更する）．次の課題を行う．

1. 課題 3.1.3 と同じ仕様で，光センサによる電圧変化を時間軸グラフで可視化せよ．要は，Arduino スケッチと Processing スケッチは課題 3.1.3 と同じものを使用して，回路のみ変更すれば良い．ただし，Processing のスケッチは修正するので，serial_method1_1byte_graph2 という名前で保存せよ．
2. Arduino スケッチ内のデータ送信間隔を 10, 50, 200 と変化させて可視化の変化を考察せよ．Processing のフレームレートはデフォルトの 60 とする．
3. フレームレートの値をデフォルト値 (60) から適当に変更して挙動を比較せよ．
4. 現状では光センサにあたる光量が増えるほど（観測される電圧値は増加するので），ウィンドウ下方に点がプロットされる．この表示を y 軸方向で反転して光センサに当たる光量が増えるほど，ウィンドウ上方に点をプロットしたい．そのように Processing のスケッチを修正せよ．
5. 現状では点のプロットとしてグラフを描いているが，折れ線グラフを描くように Processing のスケッチを修正せよ（点のプロットは残して線を引く）．

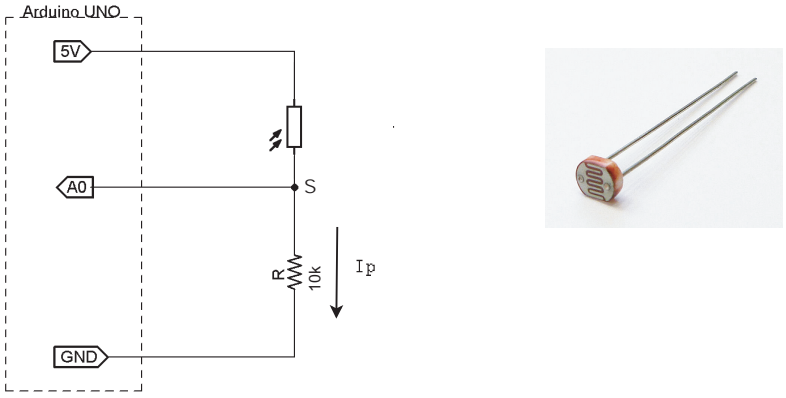

図 3.13: CdS 光センサ（右図）にあたる光の量で変化する電圧変化を Arduino で観測する回路図

＜課題 3.1.5 ＞ データのメーター表示

課題 3.1.4 では光センサによる電圧変化の時間軸グラフによる可視化を行った．本課題では光センサによる電圧変化を図 3.14 に示されるようなメーター表示で表すことを考える．仕様は以下の通りとする．作成した Processing スケッチを serial_method1_1byte_meter という名前で保存せよ．

- ウィンドウサイズ 500 × 500
- 回転するメーター（棒）の中心が (200,200)，長さが 100.
- 光センサーに当たる光量が 0 の時（送信されてくる AD 変換後の値が 0 の時），メーターが上を向く．
- 送信されてくる AD 変換後の値の増加（光量が増える）に比例してメータが時計まわりに回る．AD 変換後の値が 256（実際は 255 が最大）でちょうど 1 週するものとする．
- 適当に装飾する．例えば図 3.14 の例ではメータの回りを円で囲っている．

（補足）Processing では

```
float omega;
float x = cos(omega);
float y = sin(omega);
```

のように三角関数が使用できる．引数 omega の単位はラジアン（$2\pi = 360°$）である．また，円周率は定数 PI で定義されている．

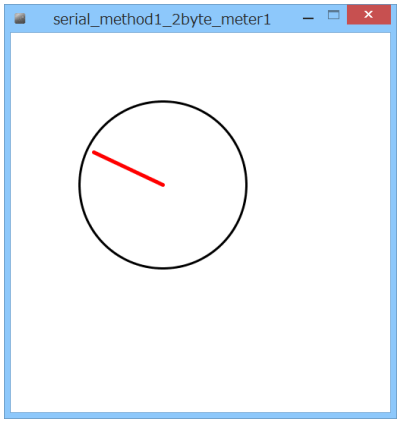

図 3.14: メーター表示の例

3.1.12　発展課題

＜発展課題 3.1.1＞ 複数データの送信（この課題は来週きちんとやります）
次のタスク 2 を通信方式 1 で実現する Arduino スケッチと Processing スケッチを作成して実行せよ．

タスク 2 ：2 つの電圧値情報の可視化（複数）

- 図 3.15 に示す回路を Arduino とブレッドボードを用いて構成し，2 つの半固定抵抗を回して変化する 2 点 S0 と S1 の電圧値を Arduino の A0 ポートと A1 ポートで観測（要するに，タスク 1 で抵抗を 2 つにした場合に相当）．
- Arduino で観測した点 S0 と S1 の電圧値（を AD 変換した値を）を Processing に送信し，S0 の電圧値（に比例する値）を横幅，S1 の電圧値（に比例する値）を縦幅，とする楕円として電圧値をリアルタイムに可視化．
- ウィンドウサイズは (256,256) とし，ウィンドウの中心を楕円の中心とする．楕円の横と縦の幅はそれぞれ電圧値が 0V 時 0，電圧値が 5V の時 256 となるように表示．

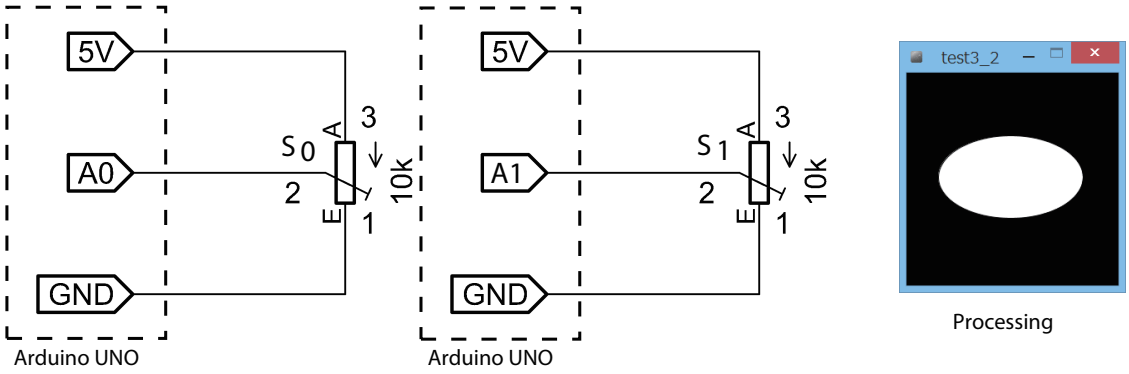

図 3.15: 2 つの半固定抵抗により変化する 2 点 S0 と S1 の電圧値を楕円の横幅と縦幅でリアルタイムに可視化．

＜発展課題 3.1.2＞ パーコレーションシミュレーション（1 時間以上要するかも知れない）
Processing を用いて，以下のシミュレーションを可視化せよ．

- 森林での火事の広がり方を $N \times N$ の 2 次元の格子点でモデル化してシミュレーションする．格子点の状態は，焼野原（状態 0），木が生えている（状態 1），火事（状態 2) の 3 種類とする．
- あるタイムステップ t で，ある格子点の木が火事（状態 2）であった場合，その格子点に隣接する 4 つの格子点それぞれにおいて，木が生えている（状態 1）場合は確率 p で次のタイムステップ $t+1$ に火事（状態 2）になるものとする．ただし，あるタイムステップで火事（状態 2）となっている格子点は次のタイムステップでは焼野原（状態 0）となり，それ以降は永久に焼野原（状態 0）となる．
- $N \times N$ の格子点の外には燃え広がらないものとする．
- タイムステップ $t = 0$ において，全ての格子点には木が生えていて（状態 1），中心の 1 つの格子点のみ火事（状態 2）であるとする．その後のタイムステップにおける格子点の状態をシミュレーションして可視化せよ．ここで，焼野原（状態 0）は黒，木が生えている（状態 1）は緑，火事（状態 2）は赤で表すものとする．図 3.16 に参考図を示す．

- パラメータとして，ウィンドウサイズ (width, height), N, p を与えられるようプログラムを作成.

上記のようモデルはハーコレーション (percolation) と呼ばれ，感染症の広がりや水の浸透のような現象を分析するための簡単なモデルである．浸透確率 p が小さい場合は火事は有限な範囲で鎮火するが（N は無限大とする），ある値を超えると急に浸透が無限に広がるようになる（相転移）．この時の確率を臨界確率という．上記の 2 次元格子モデルの場合の臨界確立は 0.593…であることが知られている．

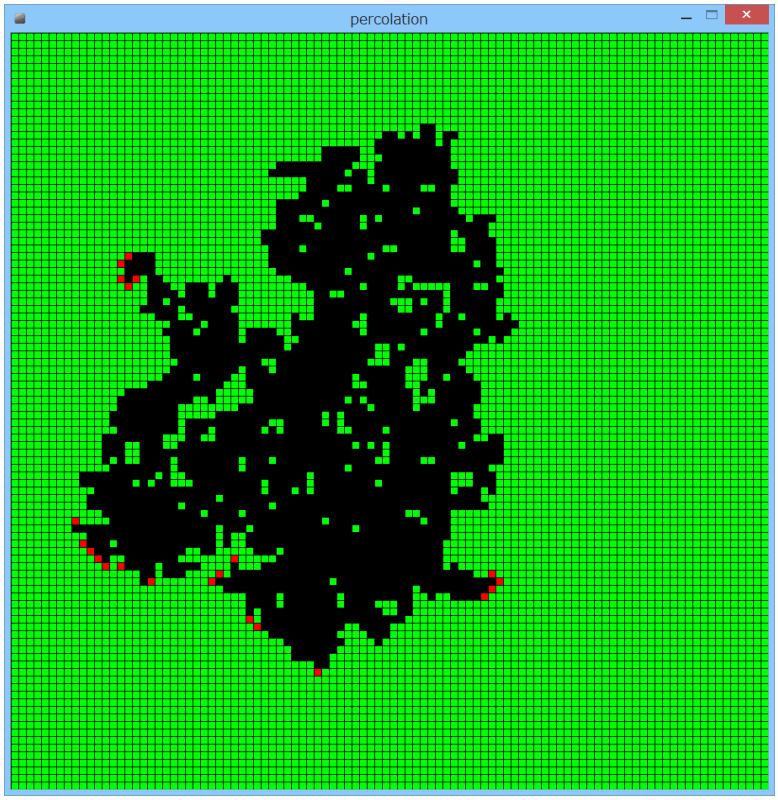

図 3.16: ハーコレーションシミュレーション: $N = 100, p = 0.5$，タイムステップ 100 の状況

図 3.16 を作成した Processing スケッチの外部変数の定義部分のみヒントとして載せておく（この方法に沿うのがベストとは限らない）．ここでは 2 次元配列を定義しているが，1 次元配列も同様に定義できる．N が 100 を超えるくらい大きくなると，$N \times N$ の格子を 1 フレームごとに全部描画すると，時間がかかりすぎて計算が 1 フレームに追いつかなくなる．状態が変化した格子のみ色を書き換える工夫が必要である．

スケッチ: percolation

```
int N = 150; // 分割数
float prob = 0.5; // 浸透確率
int sizeX = 1000; // ウィンドウサイズ
int sizeY = 1000; // ウィンドウサイズ
int [][] status = new int[N][N];     // 0:焼失, 1:森, 2:火事, 3:現ステップで焼失
int step; // ステップ数

int [][] listFireN = new int[N*N][2];// 次ステップで新しく火事になる位置のリスト
int numOfFireN;

int [][] listFire = new int[N*N][2]; // 火事の位置リスト
int numOfFire;

int [][] listChange = new int[N*N][2]; // 現ステップでステータスに変化のあった位置
リスト
int numOfChange;

int x_pos, y_pos;
int x_posN, y_posN;

void setup()
{
  ....
```

例えば，ある時点で `numOfFire = 3` であったとすると，現在火事となっている格子数が 3 である．その格子の位置が (3, 10), (78, 93), (22, 15) であれば，2 重配列 `listFire[N*N][2]` の値は以下のようになる．

```
listFire[0][0] = 3; listFire[0][1] = 10;
listFire[1][0] = 78; listFire[1][1] = 93;
listFire[2][0] = 22; listFire[2][1] = 15;
```

【レポート 3.1　（2019 年 5 月 24 日出題，2019 年 5 月 31 日 (12:50) 締切）】

※ スケッチには詳細なコメントを記述すること．コメントの無いものは採点しない．

レポート 3.1.1 Arduino から Processing へのデータ送信：通信方式 1

演習 3.1.7 の 5. で考察した内容を報告せよ．

レポート 3.1.2 Arduino から Processing へのデータ送信：通信方式 2

演習 3.1.8 の 5. で考察した内容を報告せよ．

レポート 3.1.3 繰り返し処理による描画 (2D バージョン）

課題 3.1.1 で作成したスケッチ (draw_latice_2D) を報告せよ．

レポート 3.1.4 動画の作成: 移動する楕円（2D バージョン）

課題 3.1.2 で作成したスケッチ (draw_laticeMove_2D) を報告せよ．

レポート 3.1.5 電圧変化の時間軸グラフによる可視化（点のプロット）

課題 3.1.3 ので作成した Processing のスケッチ (serial_method1_1byte_graph1) を報告せよ．また，出力したウィンドウのスナップショットを報告せよ．

レポート 3.1.6 電圧変化の時間軸グラフによる可視化（折れ線）

課題 3.1.4 で作成した Processing のスケッチ (serial_method1_1byte_graph2) を報告せよ．また，出力したウィンドウのスナップショットを報告せよ．

レポート 3.1.7 電圧変化のメーター表示

課題 3.1.5 で作成した Processing のスケッチ (serial_method1_1byte_meter) を報告せよ．また，出力したウィンドウのスナップショットを報告せよ．

発展課題レポート 3.1.1 複数データの送信

発展課題 3.1.1 で作成した Arduino と Processing のスケッチを報告せよ．

発展課題レポート 3.1.2 パーコレーションシミュレーション

発展課題 3.1.2 で作成した Processing のスケッチを報告せよ．また，シミュレーション画面のスナップショットを報告せよ．浸透確率を変化させたときの挙動について考察せよ．

Processing 言語のリファレンスサイトとして以下を挙げておく．

（日本語）http://www.musashinodenpa.com/p5/index.php
（英語） https://processing.org/reference/
（日本語）http://tetraleaf.com/p5_reference_alpha/index.html

参考図書

[1] Casey Reas, Ben Fry（著）　船田巧（訳）: Processing を始めよう，株式会社オライリー・ジャパン (2013).