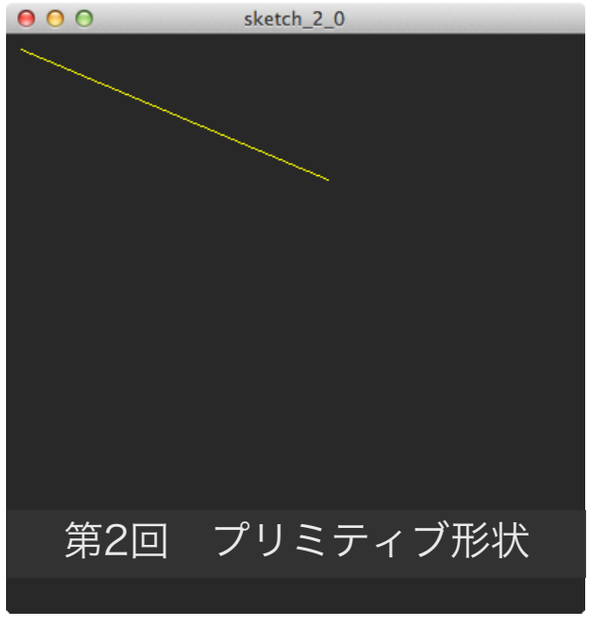
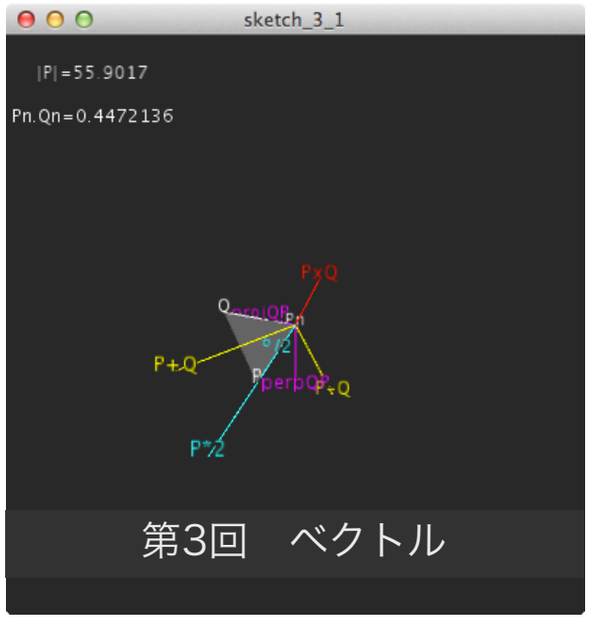
# CGとCADの数理

# GEOMETRIC MODELING AND COMPUTER GRAPHICS

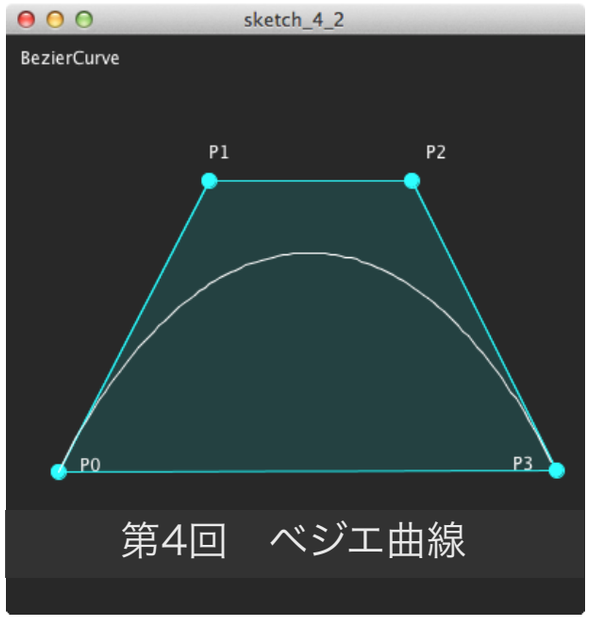
## 第02回 プリミティブ図形



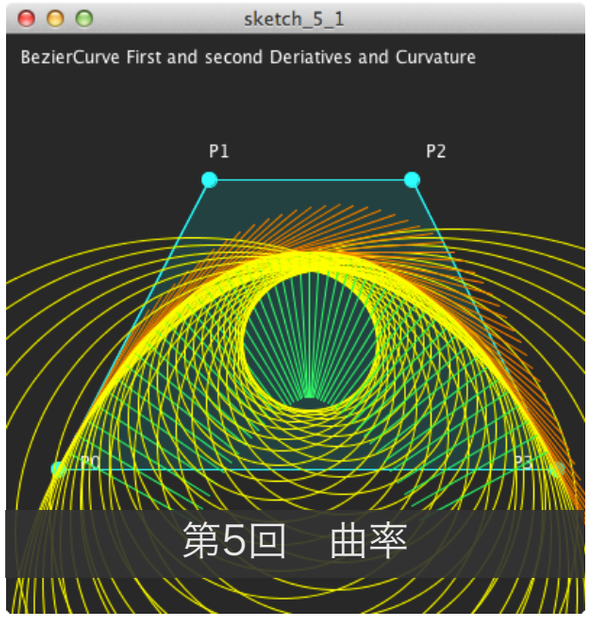

第2回　プリミティブ形状



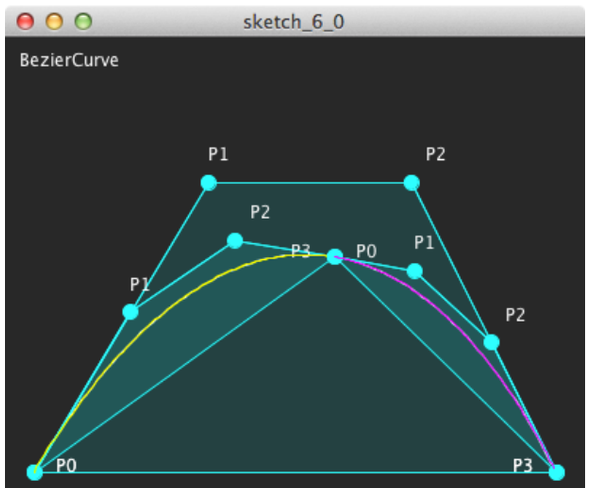

第3回　ベクトル



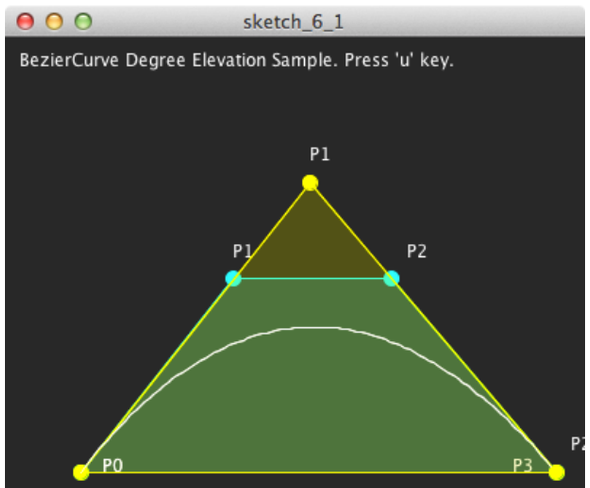

第4回　ベジエ曲線

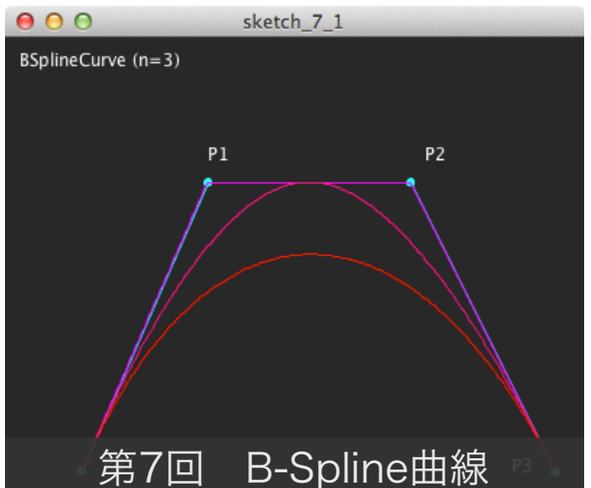



第5回　曲率

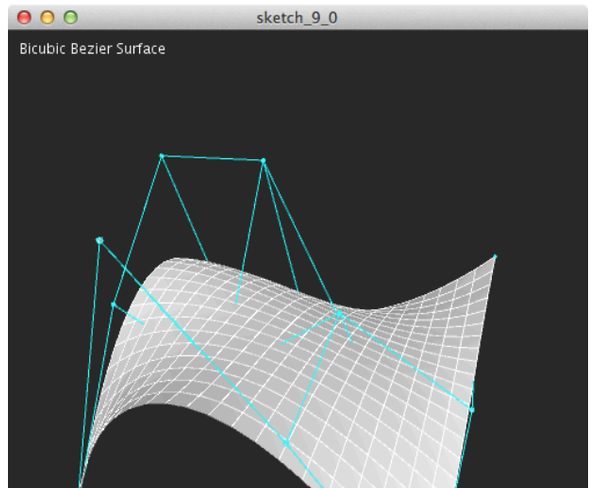



第6回　次数上げ、分割と接続




第7回　B-Spline曲線
第8回　NURBS曲線




第9回　パラメトリックな曲面

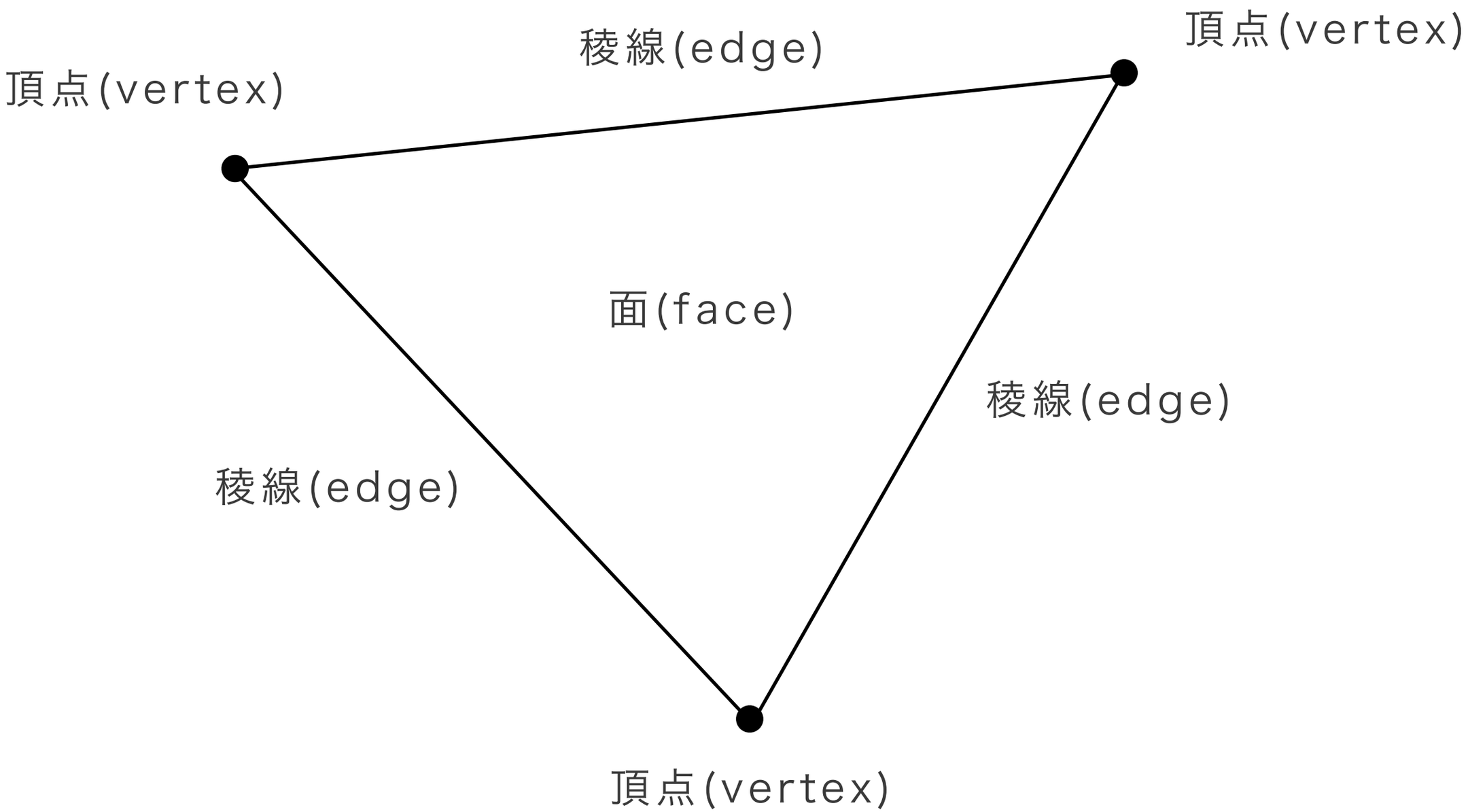

第10回　行列


第11回　三次元物体の表示


第12回　レイトレーシング

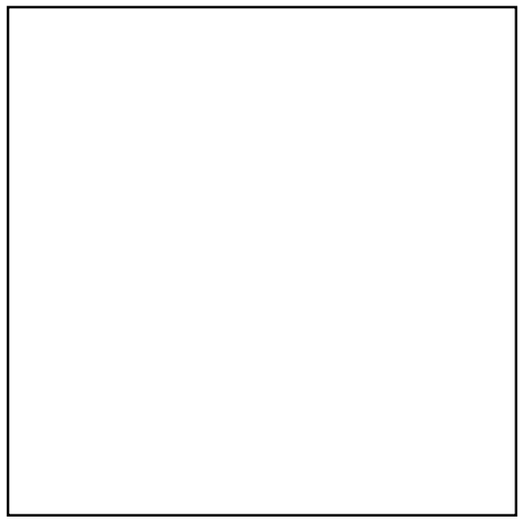

第13回　シェーディング

# ポリゴン
Polygon

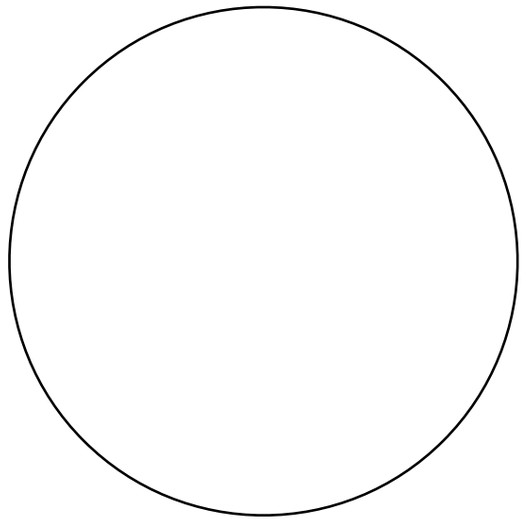



CG/CADの世界の最小要素は、頂点（vertex）、稜線（edge）、面（face）で構成されるポリゴン（polygon）と呼ばれています。業界によっては、三角形をポリゴンと言ったり、四角形をポリゴンと言ったりしますが、最小要素であるというニュアンスは共通しています。

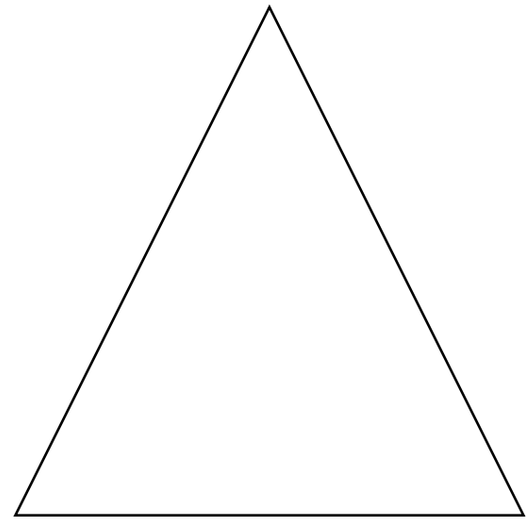
# 基本図形
Primitive



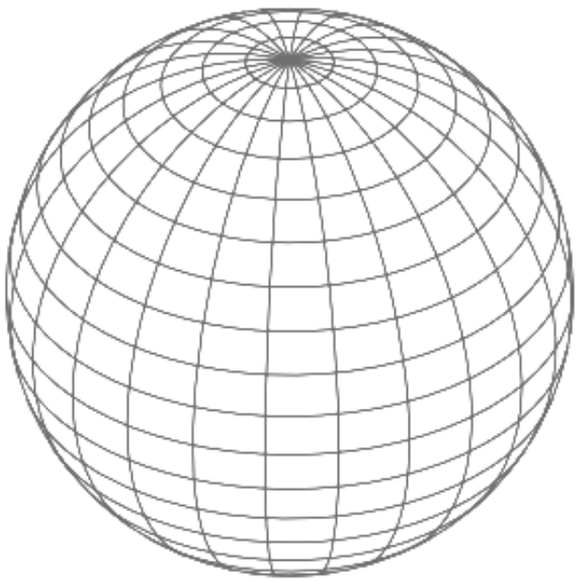
平面(plane)



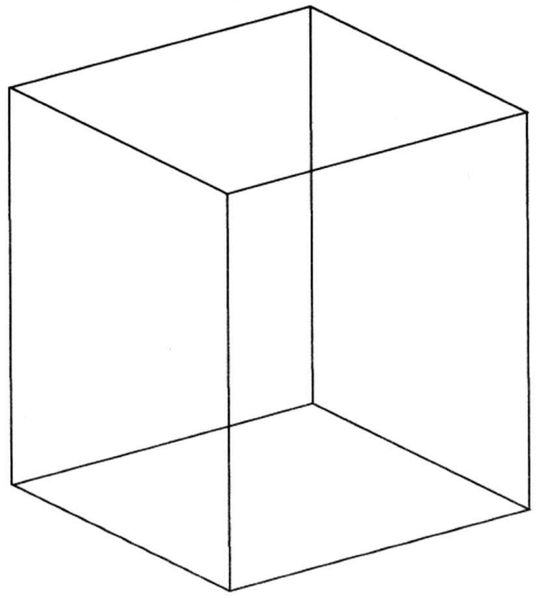
円(circle)



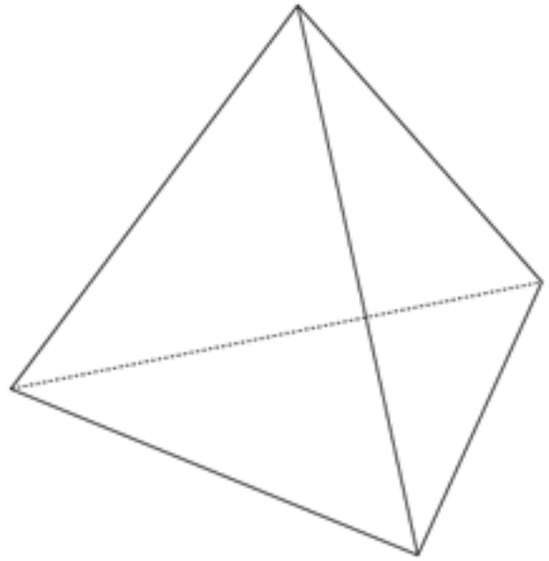
三角形(triangle)



球(sphere)

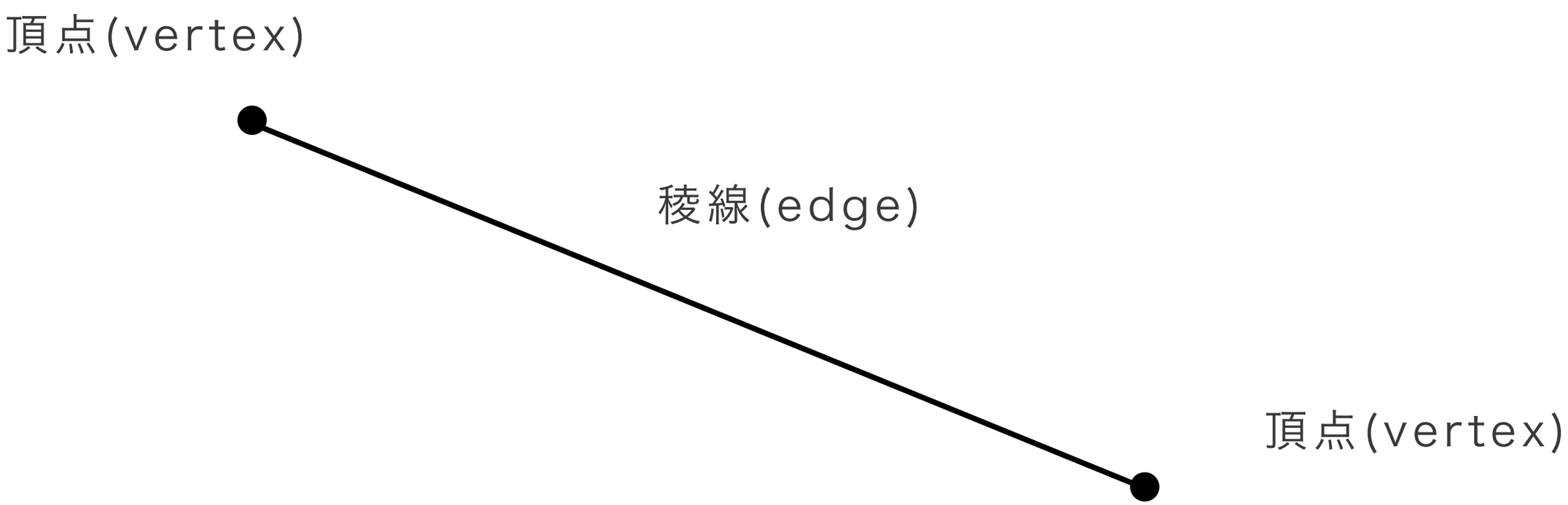


六面体(cube)



四面体(pyramid)

# 稜線
Edge




CG/CADにおいて頂点と頂点とを結ぶ線分を稜線と呼びます。昨今のクリエイティブコーディング環境（ProcessingやopenFrameworks、Cinderなど）はもちろん、グラフィックスライブラリでは、この線分描画は関数（例えば、line関数）などで標準実装されていますね。

では、line関数のアルゴリズムはご存知でしょうか。

# 頂点の描画

vertex example

SOLから今日の雛形プログラム sketch_2_0.zip を
ダウンロードして下さい

```
// Vertex Class
class CVertex {
  int x;
  int y;
  CVertex() { x=0; y=0; }
  void draw() {
    set(x, y, color(255));
  }
}
```

＜頂点クラス＞

オブジェクト指向プログラミングについては、講義では詳しくは扱いませんので、もしこのクラスの意味が全く分からない場合は、遠慮なく講義後にご質問に来て下さい。

```
CVertex v0;
CVertex v1;
```

頂点 $V_0$ と 頂点 $V_1$ を準備します。

```
void setup() {
  size(400, 400);
  v0=new CVertex(); v0.x=10;  v0.y=10;
  v1=new CVertex(); v1.x=222; v1.y=100;
}
```

頂点 $V_0$

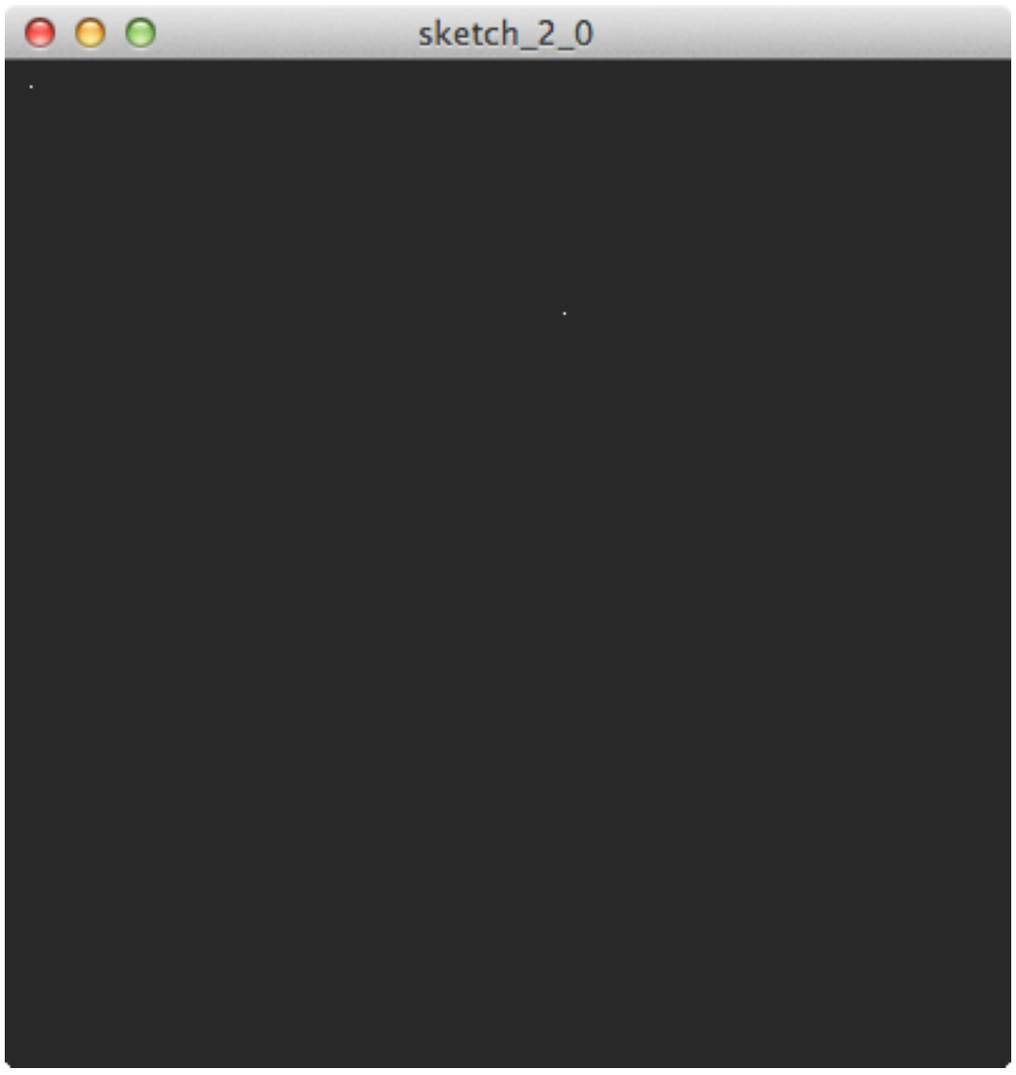
頂点 $V_1$

```
void draw() {
 background(40);
 v0.draw();
 v1.draw();
}
```

頂点 $V_0$ と 頂点 $V_1$ を描画します。

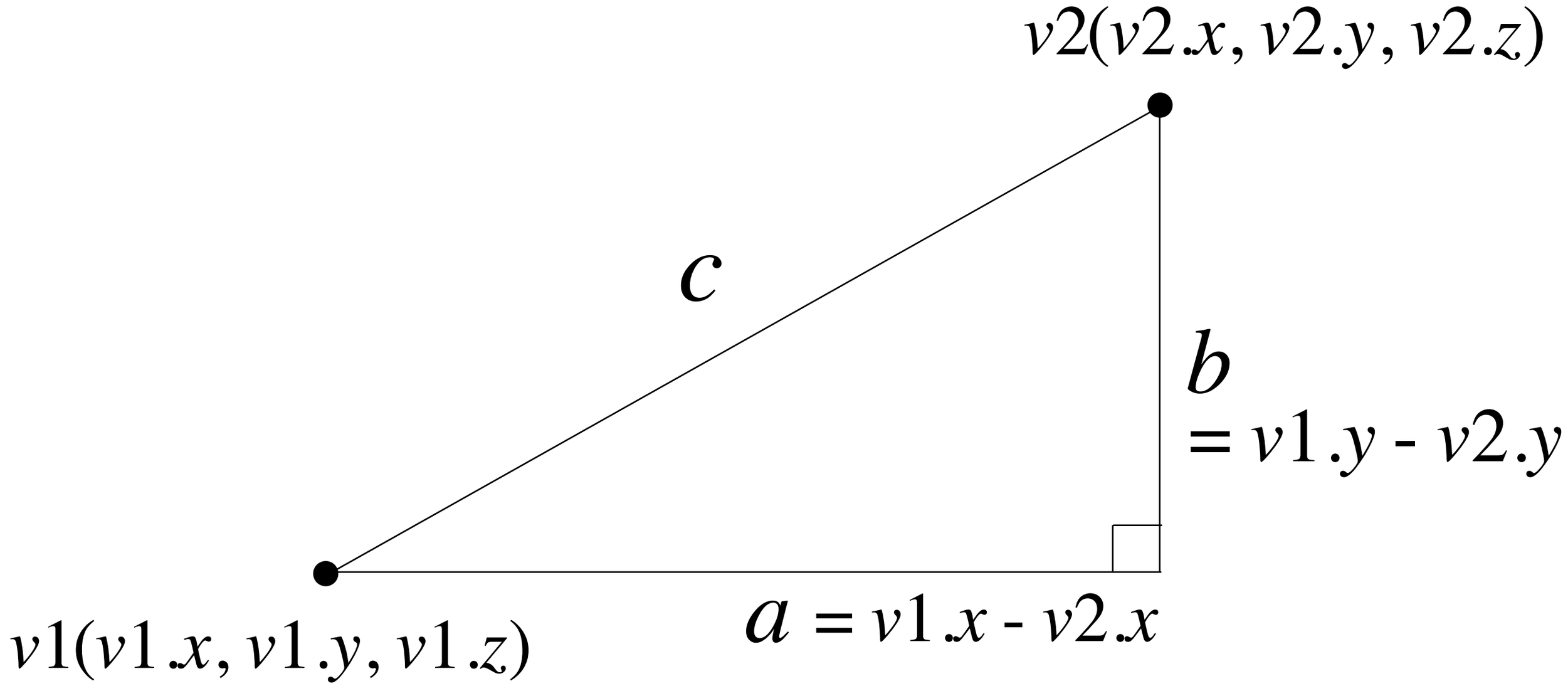
# 頂点の描画

vertex example



# ピタゴラスの定理（三平方の定理）

2頂点間の距離を求めよう




$$a^2 + b^2 = c^2$$

$$(v1.x - v2.x)^2 + (v1.y - v2.y)^2 = c^2$$

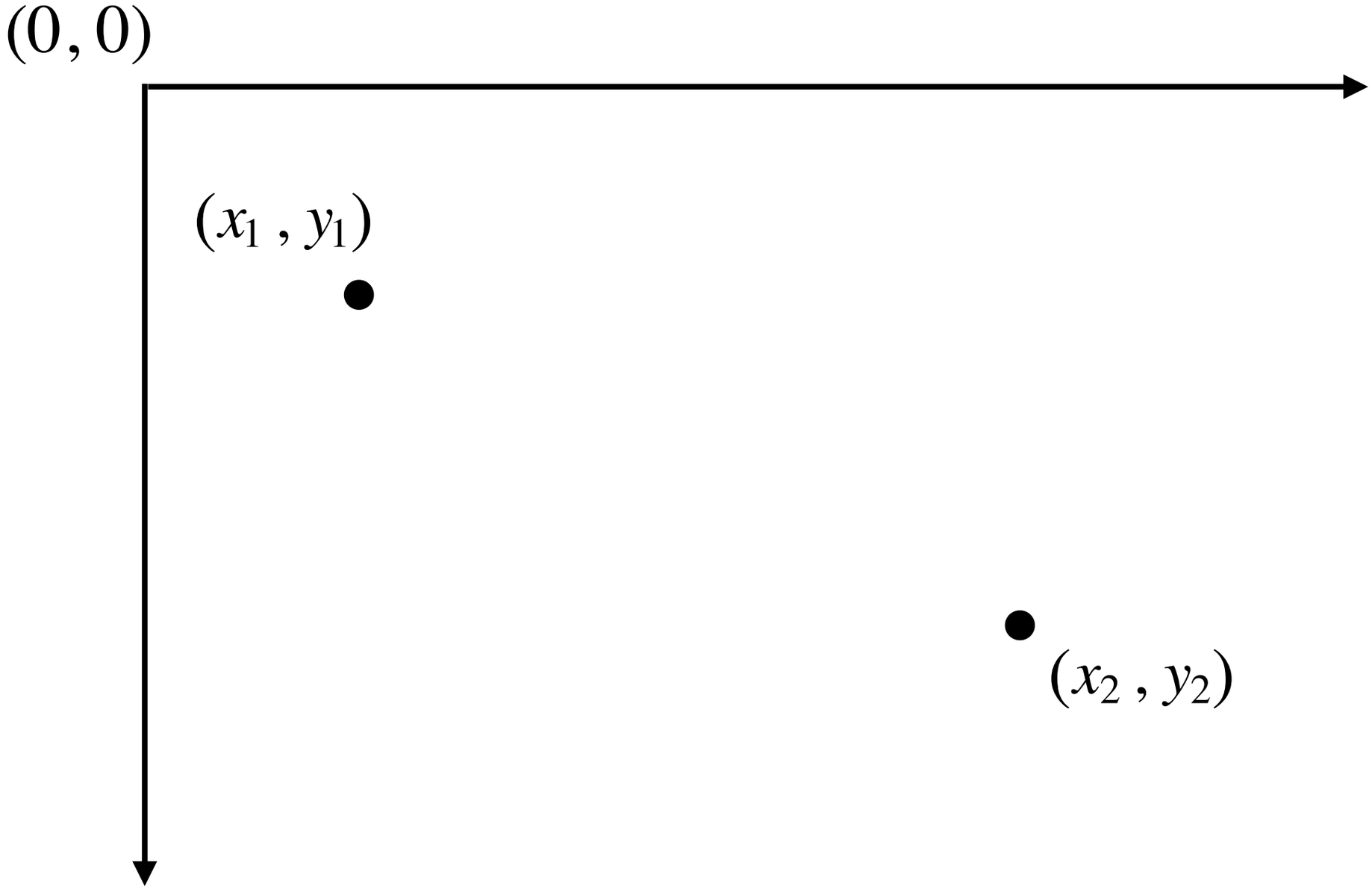
$$c = \sqrt{(v1.x - v2.x)^2 + (v1.y - v2.y)^2}$$

$$c = \sqrt{(v1.x - v2.x)^2 + (v1.y - v2.y)^2}$$

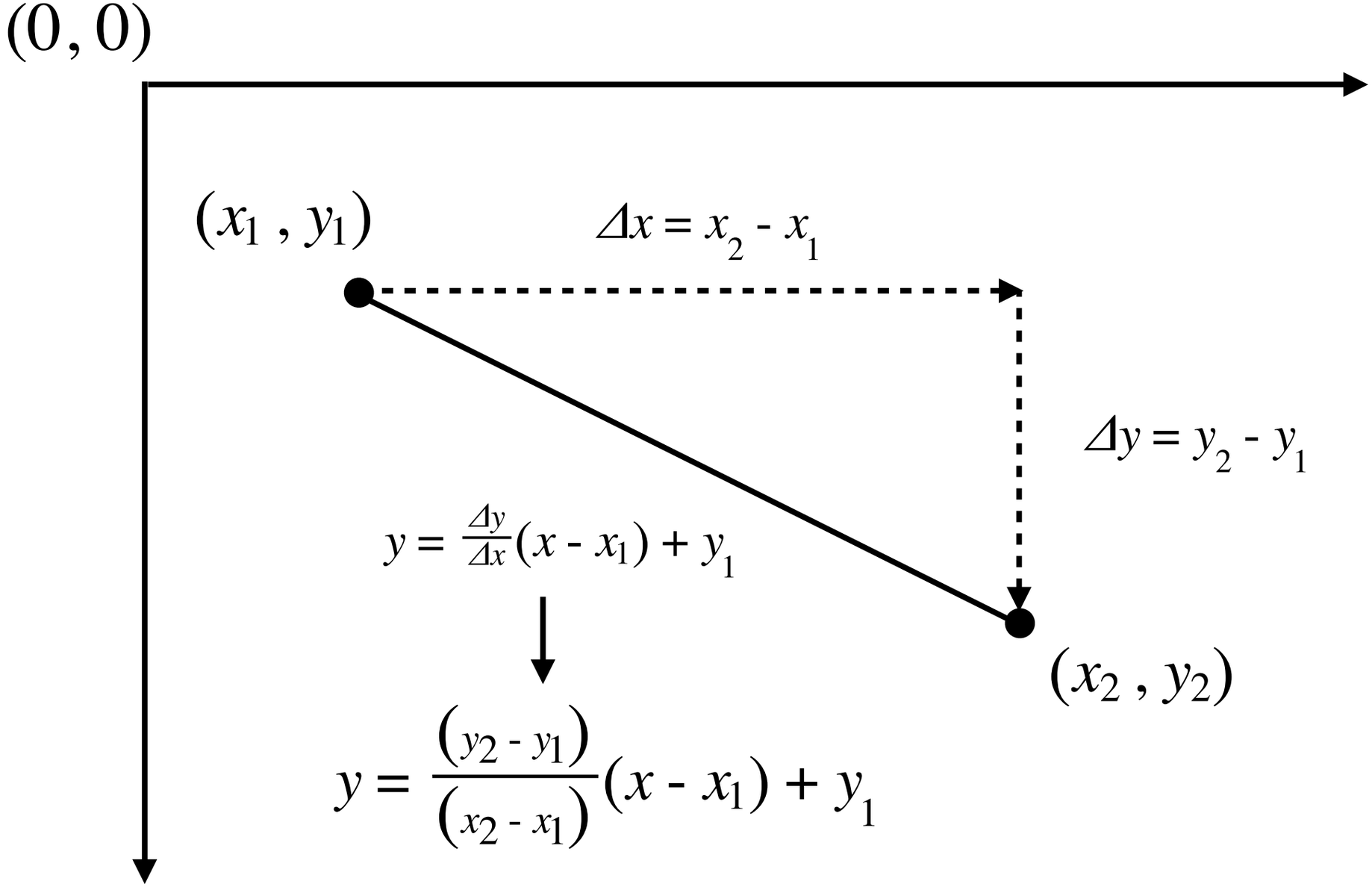
$$= \sqrt{(v1.x - v2.x) \cdot (v1.x - v2.x) + (v1.y - v2.y) \cdot (v1.y - v2.y)}$$

$$= sqrt((v1.x - v2.x) \cdot (v1.x - v2.x) + (v1.y - v2.y) \cdot (v1.y - v2.y))$$

```
void draw() {
  background(40);

  v1.x = mouseX;
  v1.y = mouseY;

  // 2. Distance
  float D = sqrt((v1.x - v0.x)*(v1.x - v0.x)
               + (v1.y - v0.y)*(v1.y - v0.y) );

  text("Distance="+D, 10, 20);

  v0.draw();
  v1.draw();

}
```




いま、画面の左上に原点（0,0）があります。今後はProcessingの座標系でお話しします。
この座標系の中に、$(x_1 , y_1)$ と $(x_2 , y_2)$ があったとしましょう。僕らの目標は、
この二つの頂点を結び、稜線を作る事です。この二つの頂点のX座標とY座標が整数な点に注意して下さい。




頂点 $(x_1 , y_1)$ を通る直線の式は、以下のようになります。

$$y = \frac{\Delta y}{\Delta x}(x - x_1) + y_1$$

この式に、先ほど計算した $\Delta x = x_2 - x_1$ と $\Delta y = y_2 - y_1$ を代入すると、以下のようになります。

$$y = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) + y_1 \qquad \left| \frac{y_2 - y_1}{x_2 - x_1} \right| \leq 1 \quad x_1 \neq x_2$$

$$y = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) + y_1$$

```
float dx=v1.x-v0.x;
float dy=v1.y-v0.y;


int x=v0.x;


for(; x<=v1.x; x++) {
  y=(dy/dx)*(x-v0.x)+v0.y;
  println(x,y);
}
```

printlnの出力結果

| | |
|---|---|
| 10.0 | 10.0 |
| 11.0 | 10.424528 |
| 12.0 | 10.849056 |
| 13.0 | 11.273584 |

| | |
|---|---|
| 218.0 | 98.30189 |
| 219.0 | 98.72642 |
| 220.0 | 99.15094 |
| 221.0 | 99.57547 |

$x$ 座標は整数型を単純に＋１していくだけなので整数ですが、$y$ 座標は実数になってしまうことがあります。

（intに型変換すれば？と思われた方は鋭いですが、ここで議論しているのはそもそもfloatという型を使っているということ自体です。そのため、今回は型変換はしないことにします）

# Breseham’s Line Algorithm

## Algorithm for computer control of digital plotter (1965)

Jack Elton Bresenham

昔の計算機では、浮動小数点（今で言えば、floatとかdouble）の負荷が高くできるだけ整数を使い、除算処理をしないアルゴリズムが求められていました。これを**DDA（Digital Differential Analyzer）**と言います。Bresenham’s Line Algorithm はDDAを踏襲し、数直線を離散化したアルゴリズムです。

# Breseham's Line Algorithm

## Algorithm for computer control of digital plotter (1965)

$y$を整数部$y_n$、少数部$e_n$に分けると以下のようになります。

$$y = y_n + e_n$$

そうすると、

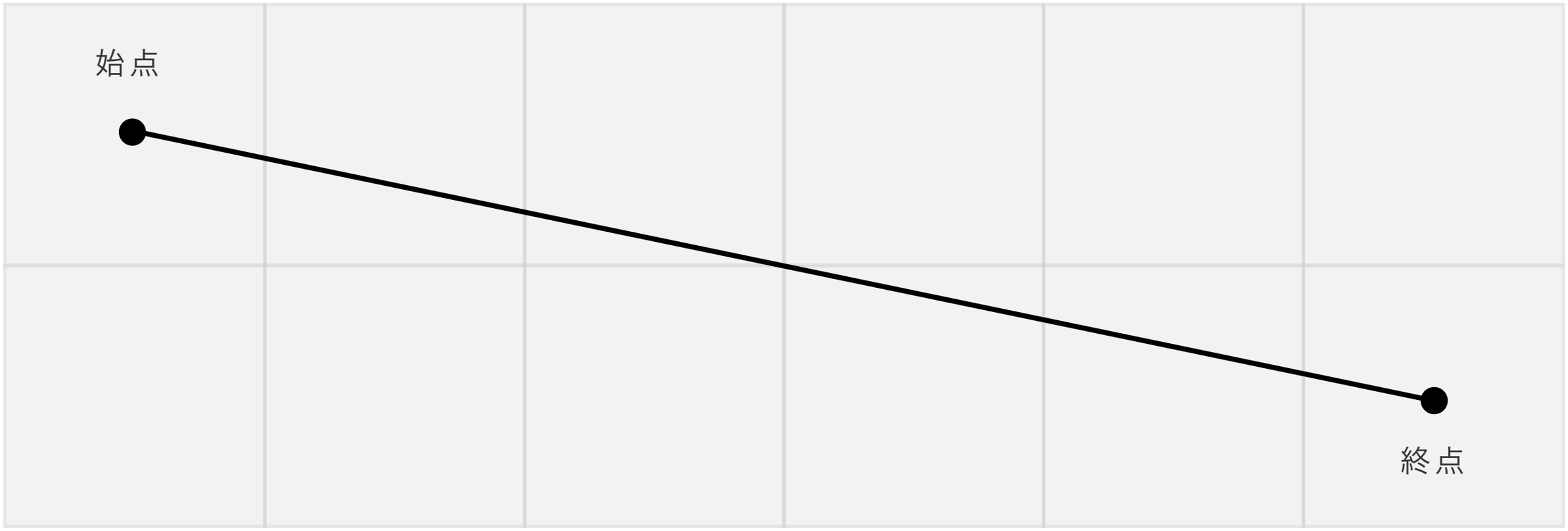
$e_n = e_{n-1} + \frac{\Delta y}{\Delta x}$ として（つまり、$\frac{\Delta y}{\Delta x}$ を蓄積していって）

$e_n \geq 0.5$ のとき、$y_n = y_{n-1} + 1$ して、$e_n$ から $1$ を引いて調整します。

$e_n < 0.5$ のとき、$y_n = y_{n-1}$

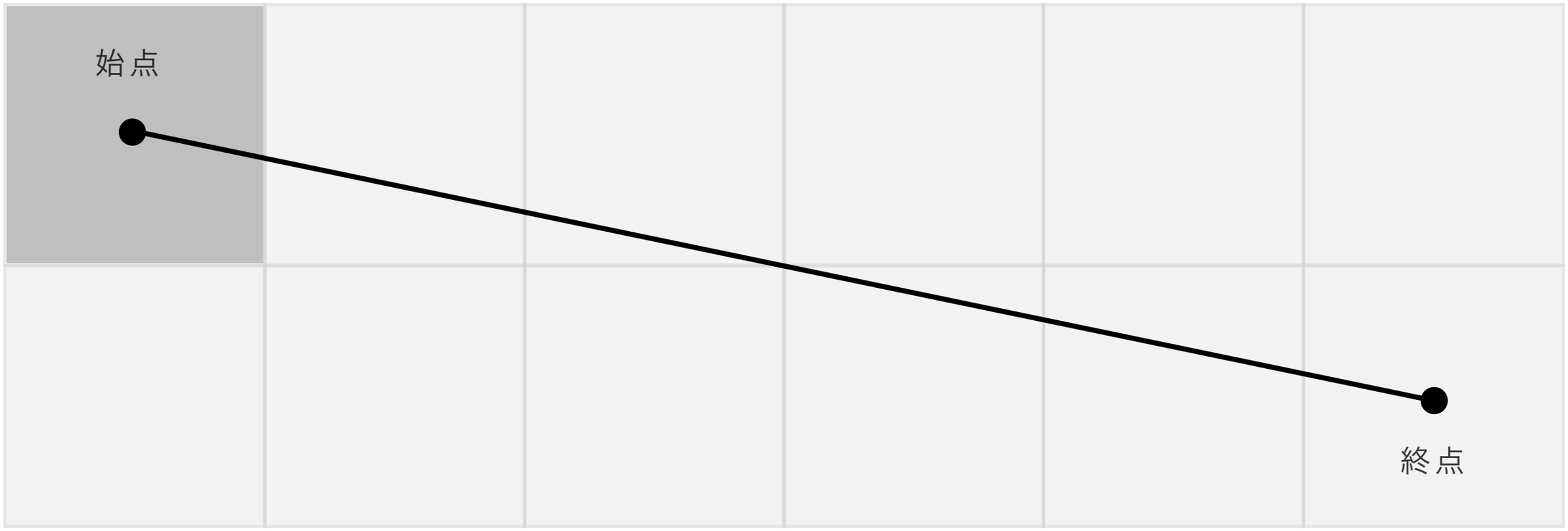
# Breseham’s Line Algorithm

## Algorithm for computer control of digital plotter (1965)



# Breseham's Line Algorithm

## Algorithm for computer control of digital plotter (1965)

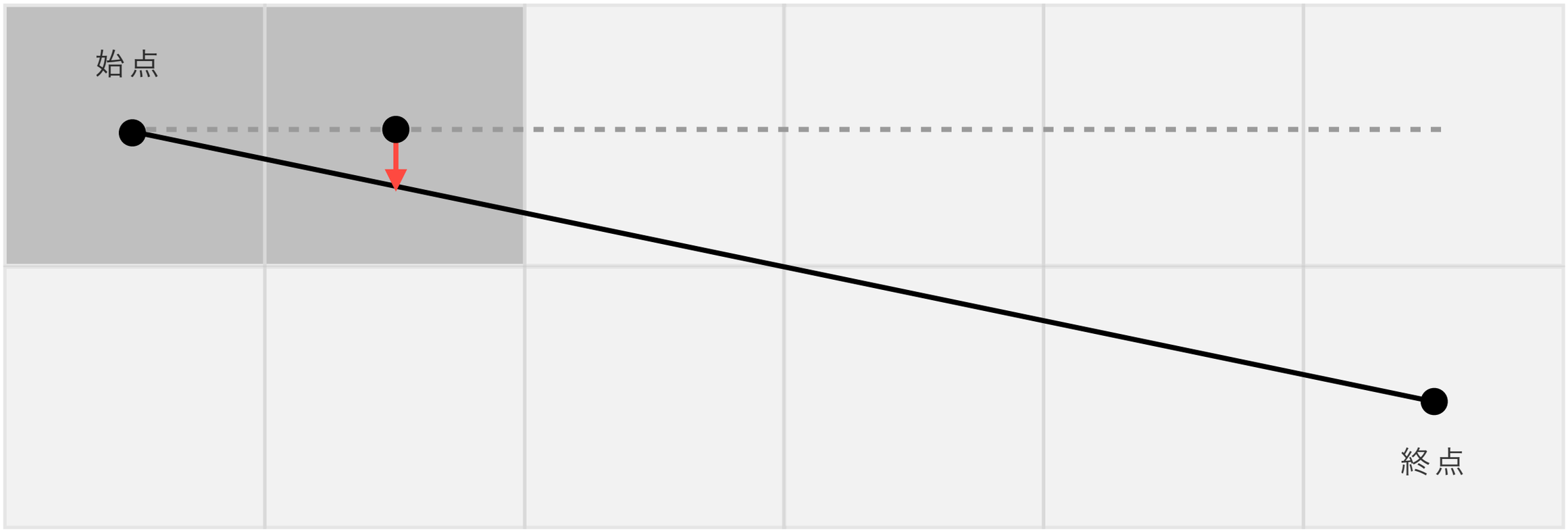



まず、始点を塗りつぶします。

灰色の矩形がピクセルで、黒い直線は本来の直線です。ぼくらの目標は、なるべくこの黒い直線に近い位置のピクセルを塗りつぶすことです。

# Breseham's Line Algorithm

## Algorithm for computer control of digital plotter (1965)



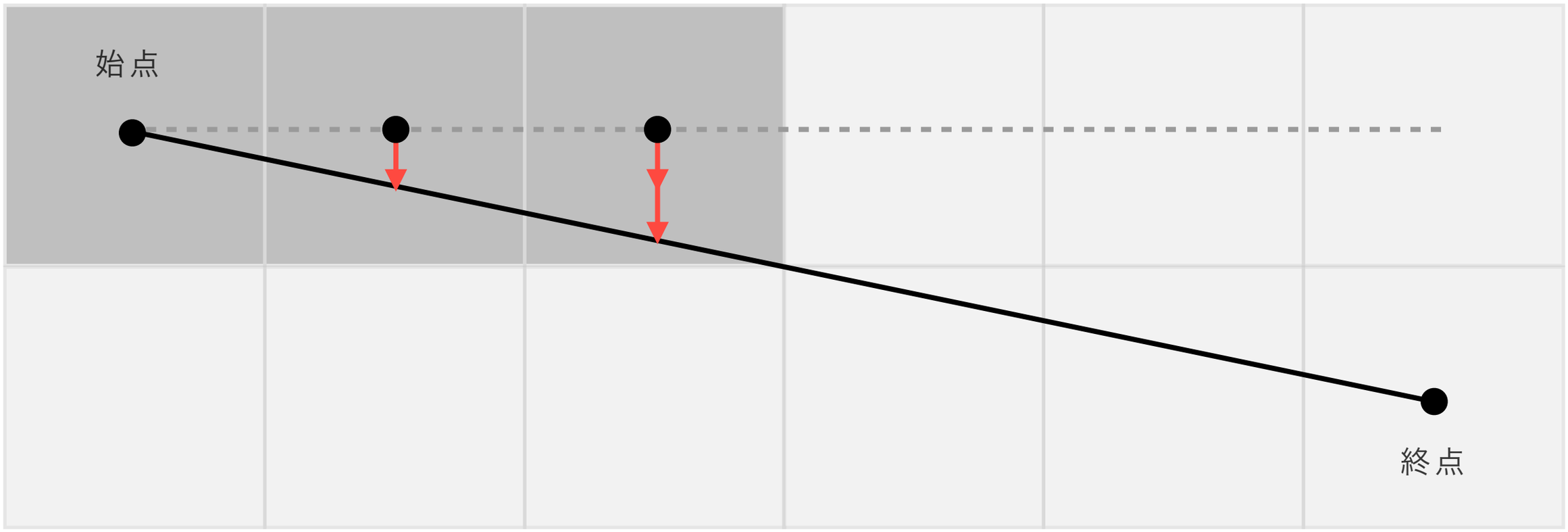

次の $x$ 座標は、＋１すれば求められます。

$\frac{dy}{dx}$ を計算して、変数 $e$ に蓄積します。

このとき、 $e<0.5$ の場合は、$y$ 座標は、そのままです。

# Breseham's Line Algorithm

## Algorithm for computer control of digital plotter (1965)



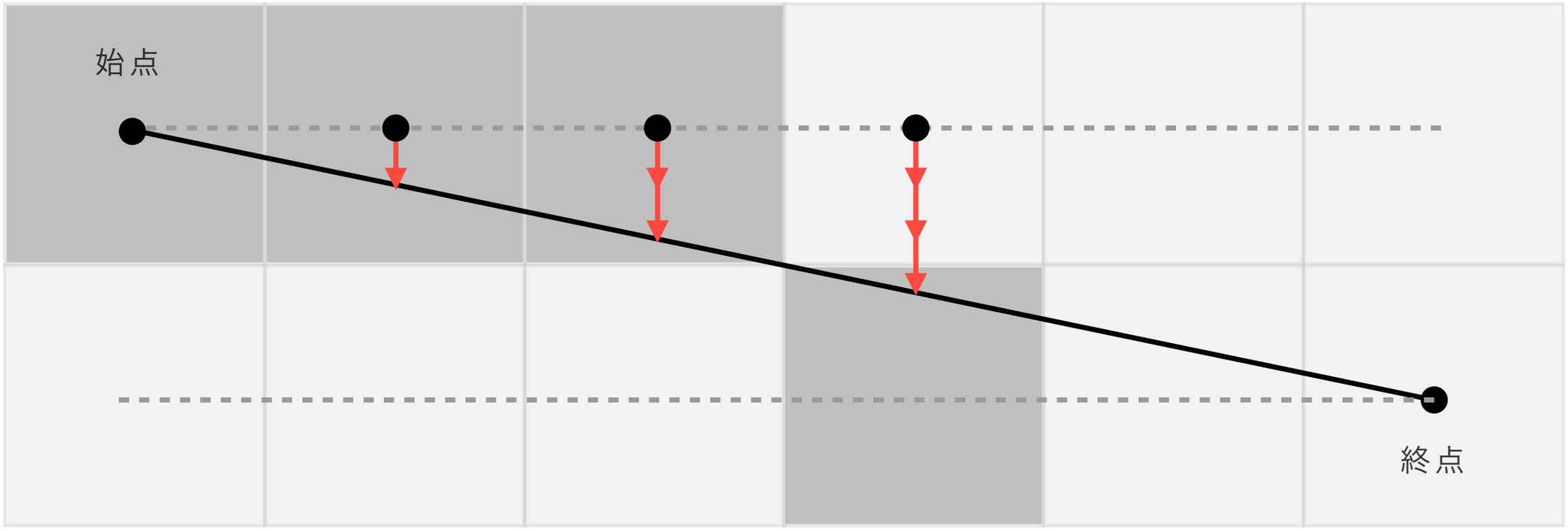

同様に、次の $x$ 座標は、＋１すれば求められます。

$\frac{dy}{dx}$ を計算して、また変数 $e$ に蓄積します。

このとき、 $e < 0.5$ の場合は、$y$ 座標は、そのままです。

# Breseham's Line Algorithm

## Algorithm for computer control of digital plotter (1965)



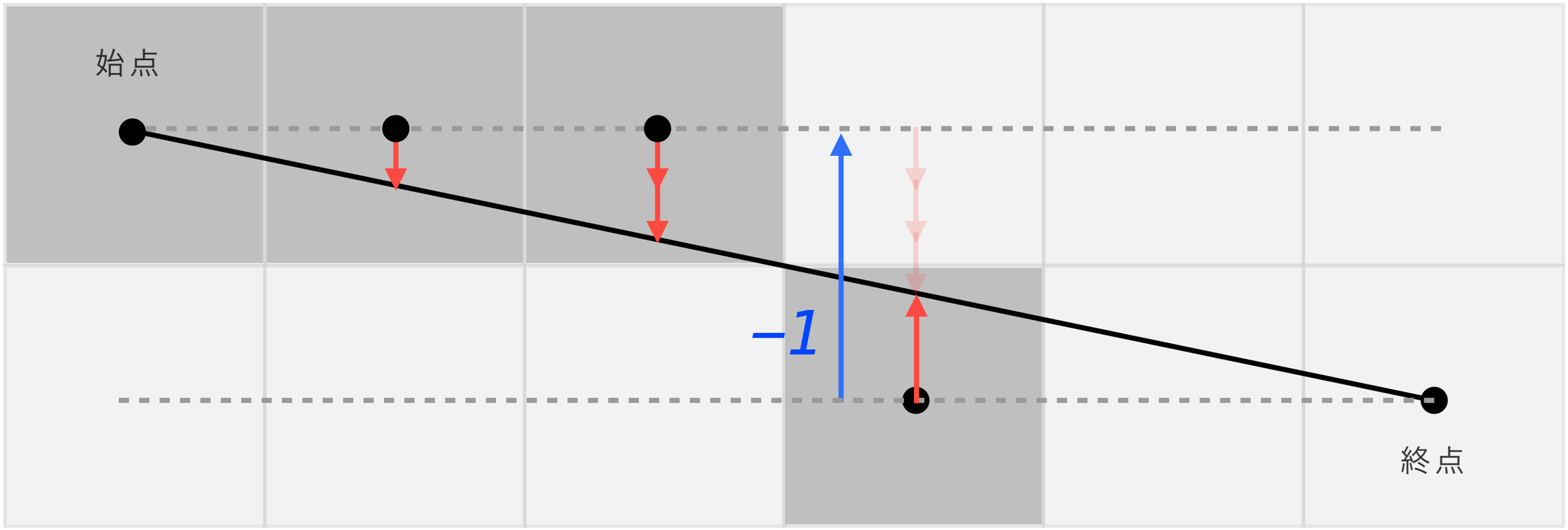

同様に、次の $x$ 座標は、＋１すれば求められます。

$\frac{dy}{dx}$ を計算して、また変数 $e$ に蓄積します。

$e \geq 0.5$ の場合は、$y$ 座標をプラス１します。 $y = y + 1$

# Breseham's Line Algorithm

## Algorithm for computer control of digital plotter (1965)




同様に、次の $x$ 座標は、＋１すれば求められます。

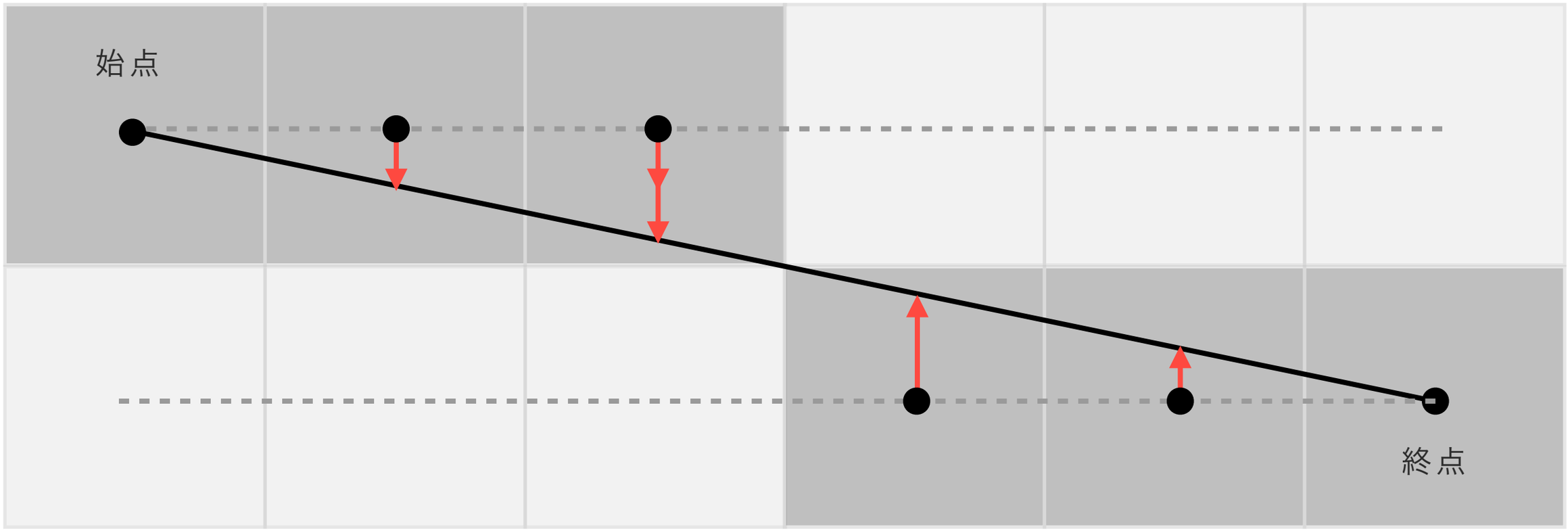
$\frac{dy}{dx}$ を計算して、また変数 $e$ に蓄積します。

$e \geq 0.5$ の場合は、$y$ 座標をプラス１します。 $y = y + 1$

$e = e - 1$ として、黒い直線との差を更新（調整）します。

# Breseham's Line Algorithm

## Algorithm for computer control of digital plotter (1965)




次の $x$ 座標は、＋１すれば求められます。

$\frac{dy}{dx}$ を計算して、変数 $e$ に蓄積します。

このとき、 $e<0.5$ の場合は、$y$ 座標は、そのままです。

以下、同様にしてピクセルを埋めていきます。

# Breseham's Line Algorithm

## Algorithm for computer control of digital plotter (1965)

SOLから今日の雛形プログラム sketch_2_1.zip を
ダウンロードして下さい

```
void draw() {
  background(40);

  int dx = v1.x - v0.x; // Δx の計算

  int dy = v1.y - v0.y; // Δy の計算

  float e = 0.0; // e_1

  int x, y; // 塗りつぶすピクセル(x,y)。両方とも整数。

  x = v0.x;   // 始点のx座標は v_0x

  y = v0.y; // 始点のy座標は v_0y

  for ( ; x <= v1.x; x=x+1 ) { // 始点から終点までxを単純増加させます。

    set(x, y, color(255));      // (x,y) の地点を塗りつぶします。

    e =  e + (float)dy/dx;      // e_n = e_n-1 + Δy/Δx の部分。

    if ( e >= 0.5 ) { // e_n ≥ 0.5 のとき
      y = y + 1; // y_n = y_n-1 + 1 して、
      e = e - 1; // e_n から 1 を引きます
    }
  }
}
```

① ② ③ ④

まだ $e$ に実数や除算が残っていますので、これを取り除くための工夫を考えてみます。

# ①

$$e_n \geq 0.5$$

右辺を移項すると、以下になります。

$$e_n - 0.5 \geq 0$$

さらに、$2\Delta x$ を両辺に乗じます。

$$2\Delta x(e_n - 0.5) \geq 0$$

この左辺を $\mathrm{E}_n$ とします。つまり、

$$\mathrm{E}_n = 2\Delta x(e_n - 0.5)$$

です。先ほどの判定式は、

$$\mathrm{E}_n \geq 0$$

と同じですね。

# プログラムの変更

```
if (e >= 0.5)
```

↓

```
if (E >= 0)
```

# ②

$$e_1 = 0$$

ですので、

$$\mathrm{E}_n = 2\Delta x(e_n - 0.5)$$

に代入すると（$n=1$とすると）、

$$\mathrm{E}_1 = 2\Delta x(e_1 - 0.5)$$

$$= 2\Delta x(0-0.5)$$

$$= -\Delta x$$

になります。

# プログラムの変更

```
float e = 0.0;
```

↓

```
int E = -dx;
```

int型になった！

# ③

$$e_n = e_{n\text{-}1} + \frac{\Delta y}{\Delta x}$$

も、変形して $E$ の形にします。

まず、両辺に $2\Delta x$ を乗じます。

$$2\Delta x\, e_n = 2\Delta x\left(e_{n\text{-}1} + \frac{\Delta y}{\Delta x}\right)$$

$$2\Delta x\, e_n = 2\Delta x\, e_{n\text{-}1} + 2\Delta y$$

次に、両辺から $0.5\cdot 2\Delta x$ を引きます。

$$2\Delta x\, e_n - 0.5\cdot 2\Delta x = 2\Delta x\, e_{n\text{-}1} - 0.5\cdot 2\Delta x + 2\Delta y$$

これを整理すると、以下になります。

$$2\Delta x(e_n - 0.5) = 2\Delta x(e_{n\text{-}1} - 0.5) + 2\Delta y$$

$$E_n = E_{n\text{-}1} + 2\Delta y$$

# プログラムの変更

```
e = e+(float)dy/dx;
```

↓

```
E = E + 2*dy;
```

(float)が消えて整数だけになった！

$$E_n = 2\Delta x(e_n - 0.5)$$

# ④

$$e_n = e_n - 1$$

も、変形して $E$ の形にします。

まず、両辺に $2\Delta x$ を乗じます。

$$2\Delta x\, e_n = 2\Delta x\,(e_n - 1)$$

$$2\Delta x\, e_n = 2\Delta x\, e_n - 2\Delta x$$

次に、両辺から $0.5\cdot 2\Delta x$ を引きます。

$$2\Delta x\, e_n - 0.5\cdot 2\Delta x = 2\Delta x\, e_n - 0.5\cdot 2\Delta x - 2\Delta x$$

これを整理すると、以下になります。

$$2\Delta x\,(e_n - 0.5) = 2\Delta x(e_n - 0.5) - 2\Delta x$$

$$E_n = E_n - 2\Delta x$$

# プログラムの変更

```
e = e - 1;
```

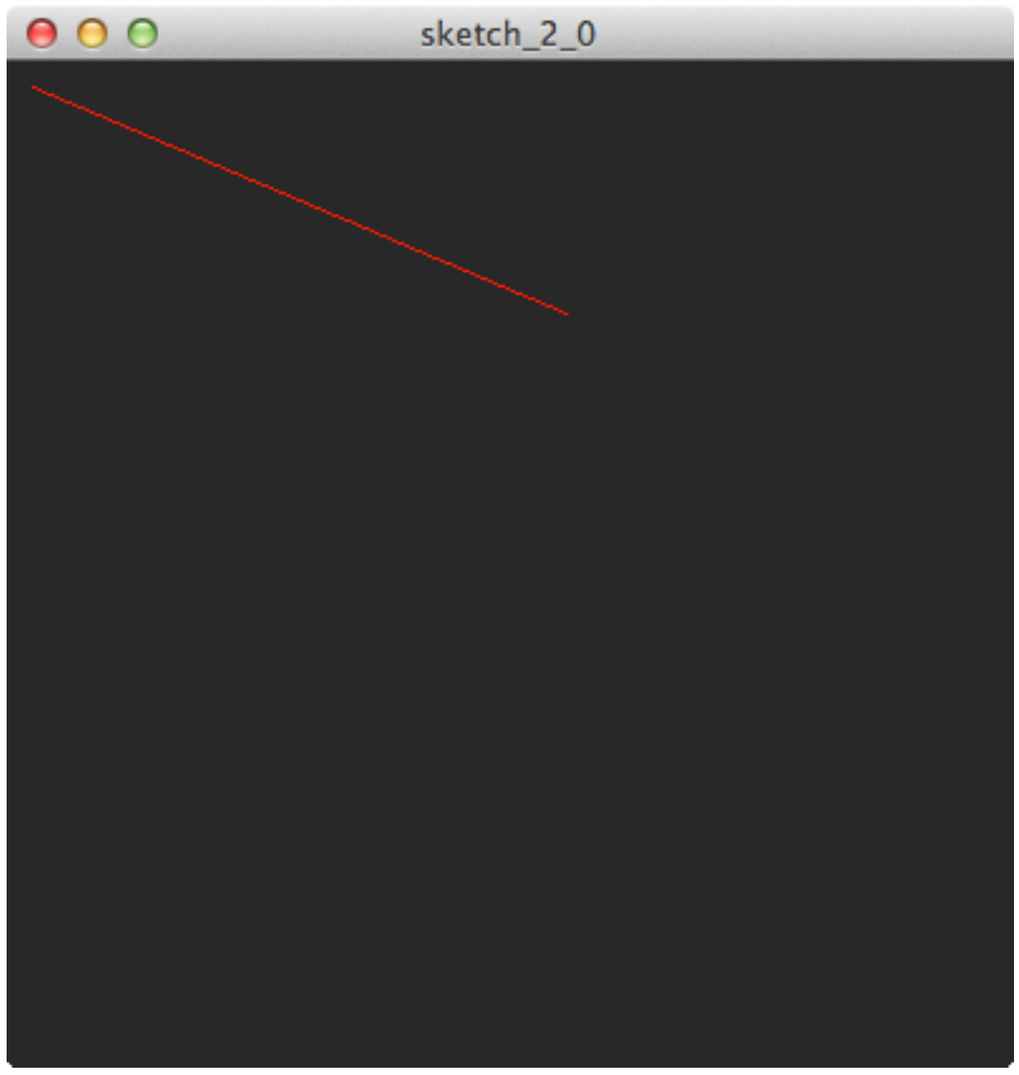
```
E = E - 2*dx;
```

```
void draw() {
  background(40);

  int dx = v1.x - v0.x;
  int dy = v1.y - v0.y;
  int E  = -dx;

  v0.draw();
  v1.draw();

  int x = v0.x;
  int y = v0.y;

  for ( ; x <= v1.x; x=x+1 ) {
    set(x, y, color(255,0,0));
    E = E + 2 * dy;
    if ( E >= 0 ) {
      y = y + 1;
      E = E - 2 * dx;
    }
  }
}
```

Bresenham's Line Algorithm のサンプルプログラムになります。背景色や直線の色はお好きに指定して下さって構いません。座標値はこのままでお願いします。実行して、下図のような画面が表示されたら成功です。

成功した方は座標値を変えてみて下さい。

実行結果



```
void draw() {
  background(40);

  stroke(255,0,0);
  line(v0.x, v0.y, v1.x, v1.y);
}
```

Processing で同様の実行結果を示すプログラムは３行でおしまいです。先ほど座標値を変えて下さった方は気づいたかもしれませんが、講義でご紹介したアルゴリズムはまだ完成ではありません。ただ我々は直線の一部の数理を学びましたので、これからはこのような数理に尊敬と感謝の意を表しながら line 関数を使わせて頂きましょう。

実行結果