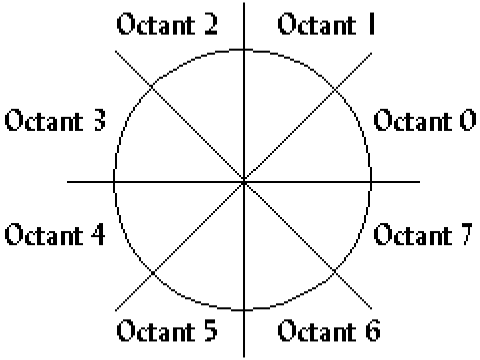
Scan Conversion

Scan Conversion

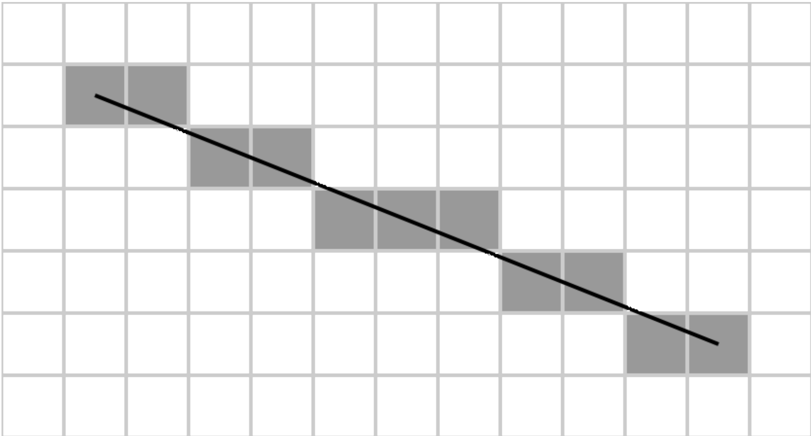
- The process of representing continuous graphics objects as a collection of discrete pixels is called scan conversion.

Basic Idea

- Bresenham's line algorithm is named after Jack Elton Bresenham who developed it in 1962 at IBM.
- Bresenham's algorithm was later extended to produce circles, the resulting algorithms being **'Bresenham's circle algorithm** and **midpoint circle algorithm**.
- The algorithm will be initially presented only for the **octant** in which the segment goes down and to the right ($x_0 \le x_1 \text{ and } y_0 \le y_1$), and its **horizontal projection** $x_1 - x_0$ **is longer than the vertical projection** $y_1 - y_0$ (the line has a positive slope whose absolute value is less than 1).



- In this octant, for each column x between x_0 and x_1, there is **exactly one row y (computed by the algorithm) containing a pixel of the line**, while each row between y_0 and y_1 may contain multiple rasterized pixels.

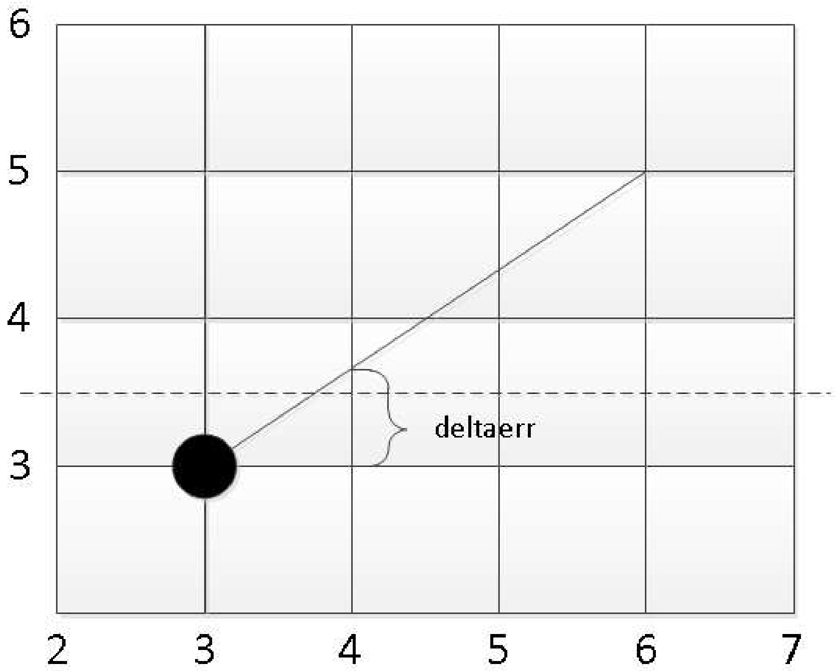


[Fig] Illustration of the result of Bresenham's line algorithm. (0,0) is at the top left corner of the grid, (1,1) is at the top left end of the line and (11, 5) is at the bottom right end of the line.

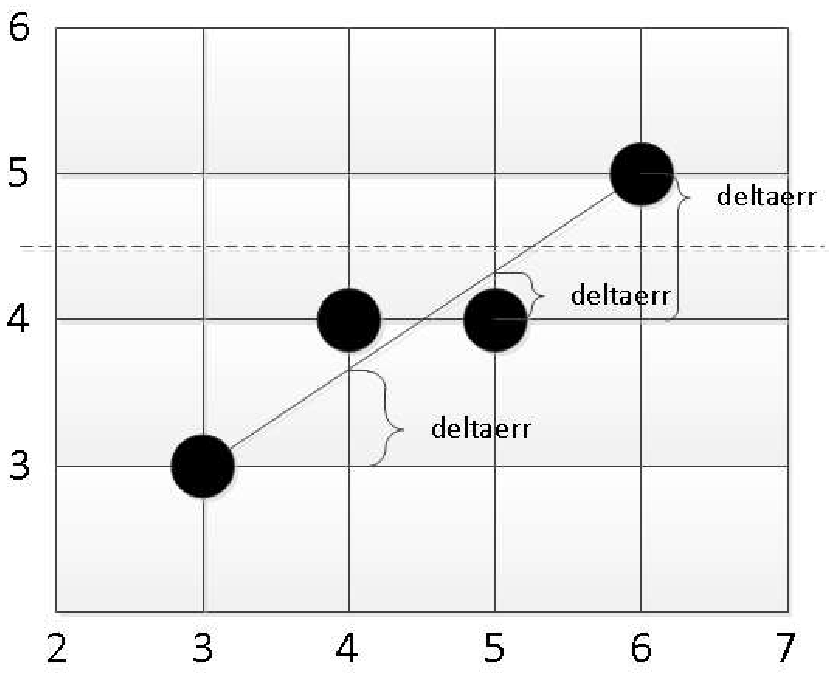
$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

- The **slope** $(y_1 - y_0)/(x_1 - x_0)$ depends on the endpoint coordinates only and can be **precomputed,** and the ideal y for successive integer values of x can be computed starting from y_0 and repeatedly adding the slope.

Ex) Draw line from (3,3) to (6,5)



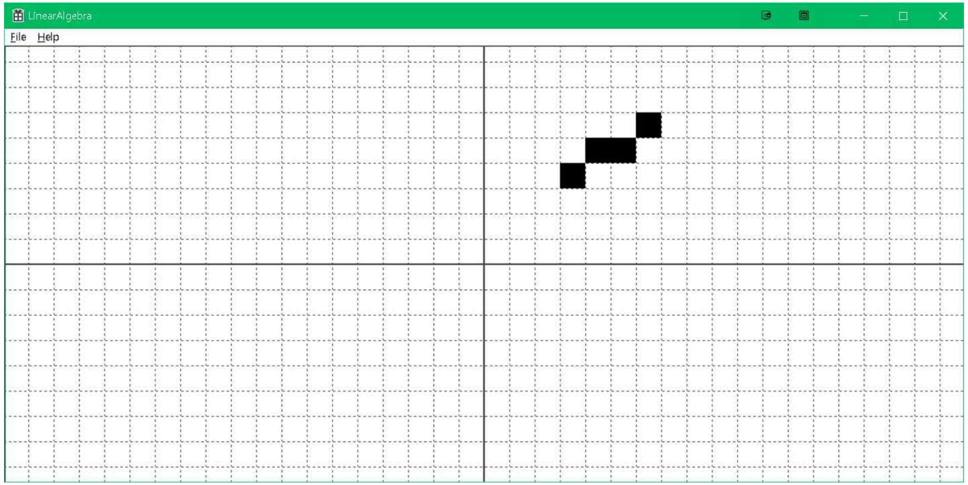
[Fig] Determine next point



[Fig] Determine next with deltaerr

```
void KVectorUtil::_ScanLineLow(HDC hdc, int x0, int y0, int x1, int y1, Gdiplus::Color color)
{
    auto sign = [](float delta){ return delta > 0.f ? 1.0f : -1.0f; };

    float deltax = x1 - x0;
    float deltay = y1 - y0;
    float deltaerr = abs(deltay / deltax); // Assume deltax != 0 (line is not vertical),
    // note that this division needs to be done in a way that preserves the fractional part
    float error = 0.0f; // No error at start
    int y = y0;
    for (int x = x0; x <= x1; ++x) {
        PutPixel(hdc, x, y, color);
        error = error + deltaerr;
        if (error >= 0.5f) {
            y = y + sign(deltay) * 1.0f;
            error = error - 1.0f;
        }
    }
}
```

Result

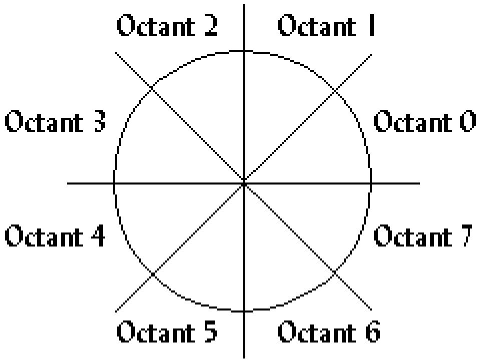


[Fig] Result

All cases

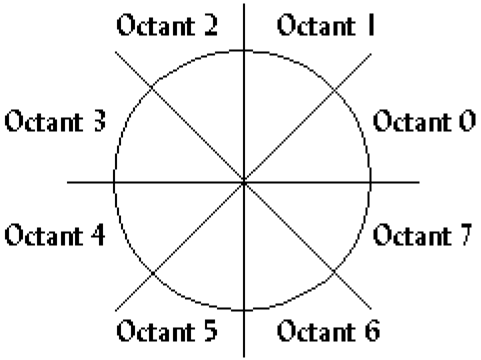
- This is only for **octant zero**, that is lines starting at the origin with a gradient between 0 and 1 where x increases by exactly 1 per iteration and y increases by 0 or 1.

- The algorithm can be extended to cover **gradients between 0 and -1** by checking whether y needs to increase or decrease (i.e. dy < 0)



[Fig] _ScanLineLow covers Octant0 and Octant7

- By switching the x and y axis an implementation for positive or negative steep gradients can be written as



[Fig] _ScanLineHigh covers Octant1 and Octant6

```
void KVectorUtil::_ScanLineHigh(HDC hdc, int x0, int y0, int x1, int y1, Gdiplus::Color color)
{
    auto sign = [](float delta){ return delta > 0.f ? 1.0f : -1.0f; };

    float deltax = x1 - x0;
    float deltay = y1 - y0;
    float deltaerr = abs(deltax / deltay); // Assume deltax != 0 (line is not vertical),
    // note that this division needs to be done in a way that preserves the fractional part
    float error = 0.0f; // No error at start
    int x = x0;
    for (int y = y0; y <= y1; ++y) {
        PutPixel(hdc, x, y, color);
        error = error + deltaerr;
        if (error >= 0.5f) {
            x = x + sign(deltax) * 1.0f;
            error = error - 1.0f;
        }
    }
}
```

- A complete solution would need to detect whether x1 > x0 or y1 > y0 and reverse the input coordinates before drawing, thus

```
void KVectorUtil::ScanLine(HDC hdc, int x0, int y0, int x1, int y1, Gdiplus::Color color)
{
    if (abs(y1 - y0) < abs(x1 - x0)) {
        if (x0 > x1) {
            _ScanLineLow(hdc, x1, y1, x0, y0);
        } else {
            _ScanLineLow(hdc, x0, y0, x1, y1);
        }
    } else {
        if (y0 > y1) {
            _ScanLineHigh(hdc, x1, y1, x0, y0);
        } else {
            _ScanLineHigh(hdc, x0, y0, x1, y1);
        }
    }
}
```

Bresenham's line algorithm

Only use integer arithematic

Line Equation

- The slope-intercept form of a line is written as

$$y = f(x) = mx + b$$

- where m is the slope and b is the y-intercept.
- This is a function of only x and it would be useful to make this equation written as a function of both x and y.
- Using algebraic manipulation and recognition that the slope is the "rise over run" or $\Delta y/\Delta x$ then

$$
\begin{aligned}
y &= mx + b \\
y &= \frac{\Delta y}{\Delta x} x + b \\
(\Delta x) y &= (\Delta y) x + (\Delta x) b \\
0 &= (\Delta y) x - (\Delta x) y + (\Delta x) b
\end{aligned}
$$

- Letting this last equation be a function of x and y then it can be written as

$$f(x,y) = 0 = Ax + By + C$$

- where the constants are

$$
\begin{aligned}
A &= \Delta y \\
B &= -\Delta x \\
C &= (\Delta x) b
\end{aligned}
$$

- The line is then defined for some constants A, B, and C anywhere f(x,y)=0. For any (x,y) not on the line then f(x,y) != 0.

Example)

$$y = \frac{1}{2}x + 1$$

$$f(x,y) = x - 2y + 2$$

- The point (2,2) is on the line

$$f(2,2) = x - 2y + 2 = (2) - 2(2) + 2 = 2 - 4 + 2 = 0$$

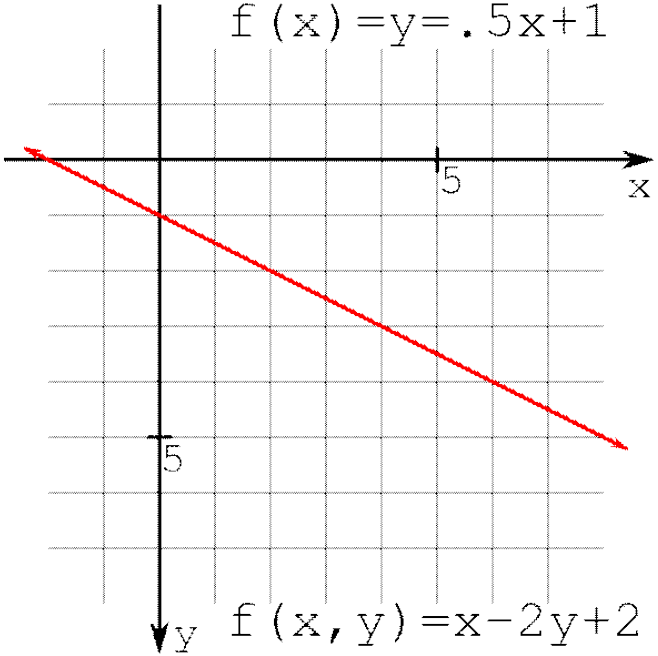
- and the point (2,3) is not on the line

$$f(2,3) = (2) - 2(3) + 2 = 2 - 6 + 2 = -2$$

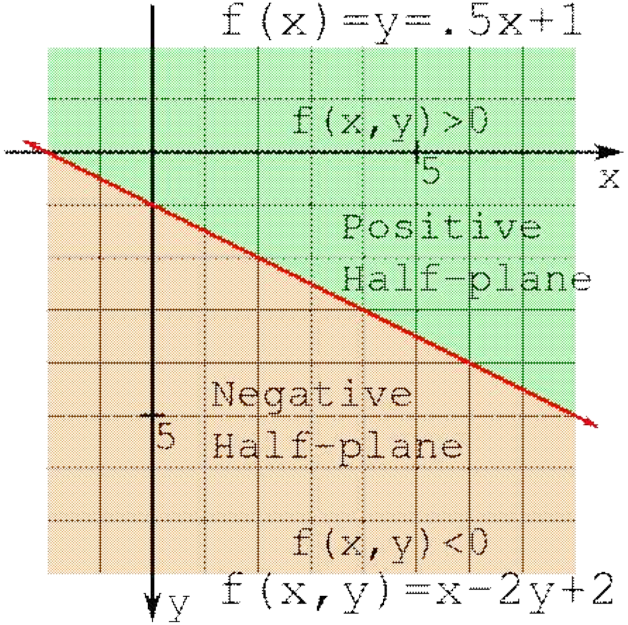
- and neither is the point (2,1)

$$f(2,1) = (2) - 2(1) + 2 = 2 - 2 + 2 = 2$$

- Notice that the points (2,1) and (2,3) are on opposite sides of the line and f(x,y) evaluates to positive or negative.



[Fig] y=f(x)=.5x+1 or f(x,y)=x-2y+2



Positive and negative half-planes

Algorithm

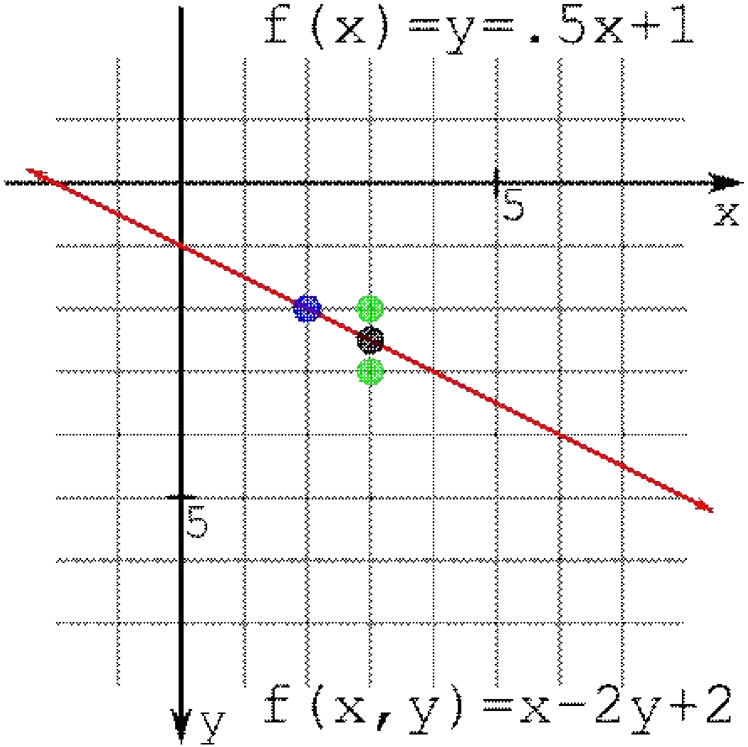
- Clearly, the starting point is on the line

$$f(x_0, y_0) = 0$$

- Keeping in mind that the **slope is less-than-or-equal-to one**, the problem now presents itself as to whether the next point should be at (x_0+1, y_0) or (x_0+1, y_0+1).

$$f(x_0+1, y_0+1/2)$$

- If the value of this is positive then the ideal line is below the **midpoint** and closer to the candidate point (x_0+1, y_0+1); in effect **the y coordinate has advanced**.



the blue point (2,2) chosen to be on the line with two candidate points in green (3,2) and (3,3). The black point (3, 2.5) is the midpoint between the two candidate points.

Algorithm for integer arithmetic

- To derive the alternative method, define the **difference** to be as follows

$$D = f(x_0 + 1, y_0 + 1/2) - f(x_0, y_0)$$

- For the first decision, this formulation is equivalent to the midpoint method since $f(x_0, y_0) = 0$ at the starting point. Simplifying this expression yields:

$$\begin{aligned} D &= [A(x_0 + 1) + B(y_0 + 1/2) + C] - [Ax_0 + By_0 + C] \\ &= [Ax_0 + By_0 + C + A + \frac{1}{2}B] - [Ax_0 + By_0 + C] \\ &= A + \frac{1}{2}B \end{aligned}$$

- Just as with the midpoint method, **if D is positive, then choose** $(x_0 + 1, y_0 + 1)$, otherwise choose $(x_0 + 1, y_0)$.

- The decision for the second point can be written as

$$f(x_0+2, y_0+1/2) - f(x_0+1, y_0+1/2) = A = \Delta y$$
$$f(x_0+2, y_0+1+1/2) - f(x_0+1, y_0+1/2) = A + B = \Delta y - \Delta x$$

- If the difference is positive then (x_0+2, y_0+1) is chosen, otherwise (x_0+2, y_0).
- This decision can be generalized by accumulating the error.

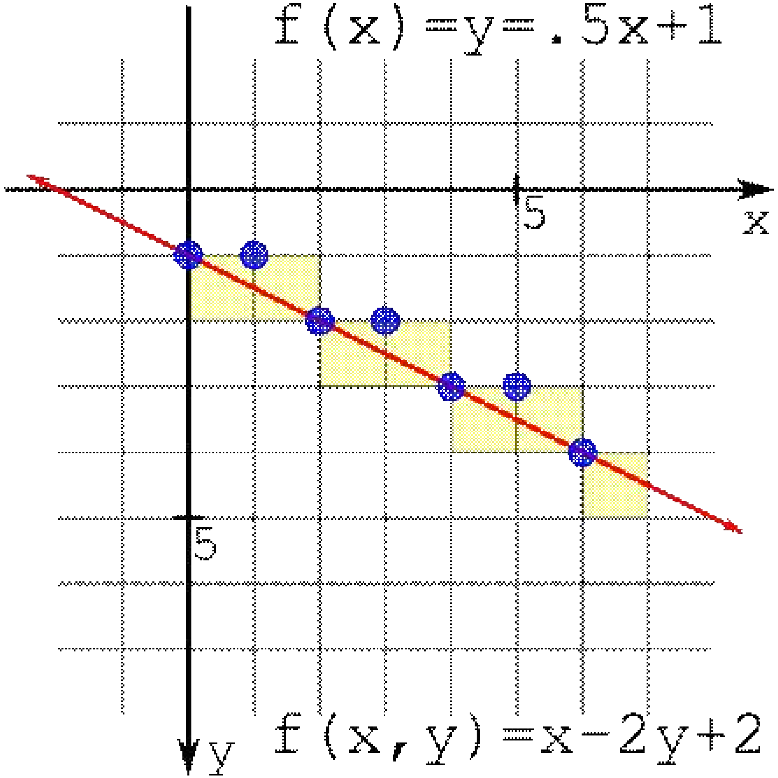
- This decision can be generalized by accumulating the error.

```
plotLine(x0,y0, x1,y1)
  dx = x1 - x0
  dy = y1 - y0
  D = dy - (1/2)*dx // A + (1/2)B
  y = y0

  for x from x0 to x1
    plot(x,y)
    if D > 0
       y = y + 1
       D = D - dx // D = D - Δx
    end if
    D = D + dy // D = D + Δy
```

- One performance issue is the 1/2 factor in the initial value of D.
- Since all of this is about the sign of the accumulated difference, then everything can be multiplied by 2 with no consequence.

```
plotLine(x0,y0, x1,y1)
  dx = x1 - x0
  dy = y1 - y0
  D = 2*dy - dx
  y = y0

  for x from x0 to x1
    plot(x,y)
    if D > 0
       y = y + 1
       D = D - 2*dx
    end if
    D = D + 2*dy
```

Ex) $f(x,y) = x - 2y + 2$ **from (0,1) to (6,4)**



[Fig] Plotting the line from (0,1) to (6,4) showing a plot of grid lines and pixels

- Running this algorithm for f(x,y)=x-2y+2 from (0,1) to (6,4) yields the following differences with dx=6 and dy=3:

D=2*3-6=0

Loop from 0 to 6

x=0: plot(0,1), D≤0: D=0+6=6

x=1: plot(1,1), D>0: D=6-12=-6, y=1+1=2, D=-6+6=0

x=2: plot(2,2), D≤0: D=0+6=6

x=3: plot(3,2), D>0: D=6-12=-6, y=2+1=3, D=-6+6=0

x=4: plot(4,3), D≤0: D=0+6=6

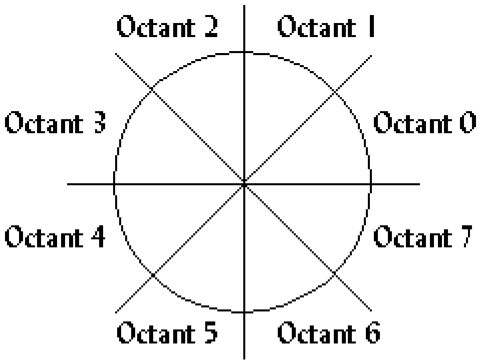
x=5: plot(5,3), D>0: D=6-12=-6, y=3+1=4, D=-6+6=0

x=6: plot(6,4), D≤0: D=0+6=6

All cases

- However, as mentioned above this is only for octant zero, that is lines starting at the origin with a gradient between 0 and 1 where x increases by exactly 1 per iteration and y increases by 0 or 1.

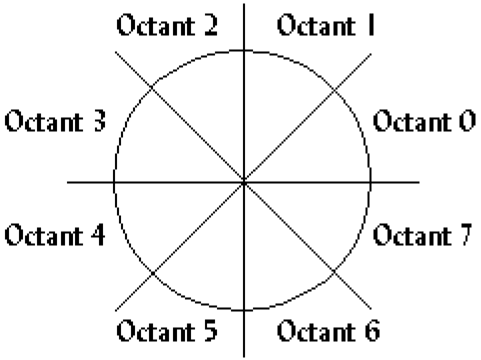
- The algorithm can be extended to cover gradients between 0 and -1 by checking whether y needs to increase or decrease (i.e. dy < 0)



[Fig] plotLineLow covers Octant0 and Octant7

```
plotLineLow(x0,y0, x1,y1)
  dx = x1 - x0
  dy = y1 - y0
  yi = 1
  if dy < 0
    yi = -1
    dy = -dy
  end if
  D = 2*dy - dx
  y = y0

  for x from x0 to x1
    plot(x,y)
    if D > 0
       y = y + yi
       D = D - 2*dx
    end if
    D = D + 2*dy
```

- By switching the x and y axis an implementation for positive or negative steep gradients can be written as



[Fig] plotLineHigh covers Octant1 and Octant6

```
plotLineHigh(x0,y0, x1,y1)
  dx = x1 - x0
  dy = y1 - y0
  xi = 1
  if dx < 0
    xi = -1
    dx = -dx
  end if
  D = 2*dx - dy
  x = x0

  for y from y0 to y1
    plot(x,y)
    if D > 0
       x = x + xi
       D = D - 2*dy
    end if
    D = D + 2*dx
```

- A complete solution would need to detect whether x1 > x0 or y1 > y0 and reverse the input coordinates before drawing, thus

```
plotLine(x0,y0, x1,y1)
  if abs(y1 - y0) < abs(x1 - x0)
    if x0 > x1
      plotLineLow(x1, y1, x0, y0)
    else
      plotLineLow(x0, y0, x1, y1)
    end if
  else
    if y0 > y1
      plotLineHigh(x1, y1, x0, y0)
    else
      plotLineHigh(x0, y0, x1, y1)
    end if
  end if
```

[문서의 끝]