The Digital Differential Analyzer (DDA)

These include a class of algorithms, which draw lines from their corresponding differential equations. Before we see the actual algorithms, let us see the concept by the example of a simple straight line.

The differential equation of a straight line is given by $\frac{dy}{dx} = \Delta y / \Delta x$

Looked another way, given a point on the straight line, we get the next point by adding x to the x coordinate and y to the y coordinate i.e. given a point p(x,y), we can get the next point as a Q(x+ Δx, y + Δ y) , the next point R as R(x+2* Δx, y+2* Δy)etc. So this is a truly incremental method, where given a starting point we can go on generating points, one after the other each spaced from it's previous points by an additional x and y, until we reach the final point.

Different values of Δx and Δy give us different straight lines.

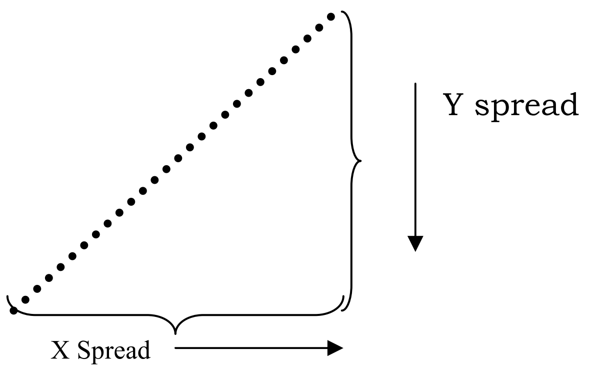
But because of inaccuracies due to rounding off, we seldom get a smooth line, but end up getting lines that are not really perfect.

We now present a simple DDA algorithm in a C like Language.

```
Procedure DDA (x1, y1, x2, y2)
/* The line extends from (x1, y1) to (x2, y2)*/
{
length = abs (x2 - x1);
if length < abs (y2-y1), then length = abs(y2-y1)
x increment = (x2-x1)/length;
y increment = y2-y1)/length;
x=x1+0.5; y=y1+0.5;
for (I=1;I<=length; i++)
{ plot (trun (x), trun (y))
x = x + x increment;
y = y + y increment;
}
}
```

We start from the point (x1,y1) and go up to the point (x2,y2)

The difference (x2-x1) gives the x spread of the line (along the x-axis) and (y2-y1) gives the y spread (along y axis) (x_2,y_2)

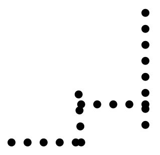


The larger of these is taken as the variable length (not exactly the length of the line)

The variables xincrement and yincrement give the amount of shifts that we should make at a time in each direction. (Note that by dividing by length, we have made one of them equal to unity. If y2-y1 is larger, then yincrement will be unity; otherwise xincrement will be unity. What this means is that in one of the directions, we move by one pixel at a time, while in the other direction, we have to calculate as to whether we have to go to the next valve or should stay in the previous value)

Plot is a function that takes the new values of x and y, truncates then and plots those points. Then we move on to the next valve of x, next value of y, plot it, and so on.

A typical DDA drawn line appears as follows:



Obviously a mean line through these points is the actual line needed.

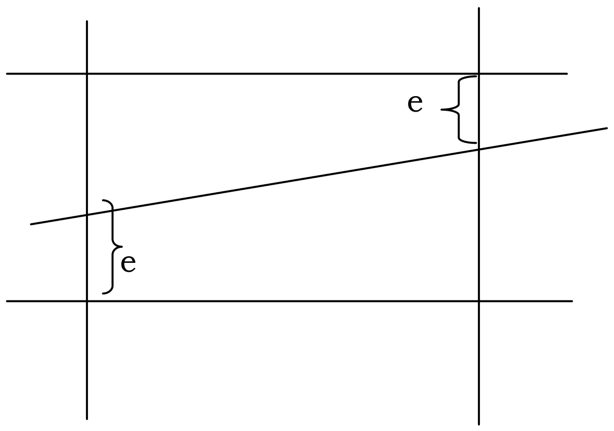
Note that the line looks like a series of steps. This effect is sometimes called a "Stair case" effect.

Now we can explain the program to generate DDA line using C Programming.

```
/*Program to Generate a Line using Digital differential Algorithm(DDA) */
   # include<stdio.h>
# include<conio.h>
# include <graphics.h>
# include<math.h>
/*function for plotting the point and drawing the line */
void ddaline(float x1,float y1,float x2,float y2)
{
int i, color=5;
float x,y, xinc, yinc, steps;
steps=ads(x2-x1);
if (steps<abs(y2-y1);
steps=abs(y2-y1);
xinc=(x2-x1)/steps;
yinc=(y2-y1)/steps;
x=x1;
y=y1;
putpixel((int)x,(int)y,color);
for(i=1;i<=steps; i++)
{
putpixel((int)x,(int)y,color);  /* plots the points with specified color */
x=x+xinc;
y=y=yinc;
}
{
/* The main program that inputs line end point and passes them to ddaline()
function */
void main()
{
 int gd=DETECT,gm,color:
float x1,y1,x2,y2;
printf("\n enter the end points:\n");
scanf("%f %f %f %f",&x1,&y1,&x2,&y2);
clrscr();
initgraph(&gd,&gm, "c:\\tc\\bgi");
ddaline(x1,y1,x2,y2);
getch();
closegraph();
}
```

The main draw back of DDA method is that it generates the line with "stair case" effect. It also needs all parameters as float but C language syntax does not take any floating-point values as co-ordinates in computer graphics.

Bresenham's algorithm: This algorithm is designed on a very interesting feature of the DDA. At each stage, one of the coordinates changes by a value 1 (That is because we have made either (y2-y1) or x2-x1) equal to length, so that either (x2-x1)/length or (y2-y1)/length will be equal to 1). The other coordinate will either change by 1 or will remain constant. This is because, even though the value should change by a small value, because of the rounding off, the value may or may not be sufficient to take the point to the next level.



Look at the above figure. The left most point should have been at the point indicated by x, but because of rounding off, falls back to the previous pixel. Whereas in the second case, the point still falls below the next level, but because of the rounding off, goes to the next level. In each case, e is the error involved in the process.

So what the Bresenham algorithm does it as follows. In each case adds Δy or Δx as the case may be, to the previous value and finds the difference between the value and the (next) desirable point. This difference is the error e. If the error is >=1, then the point is incremented to the next level and 1 is subtracted from the value. If e is <1, we have not yet reached the point, where we should go to the next point, hence keep the display points unchanged.

We present the algorithm below

```
e=(deeltay/deltax)-0.5;
for(i=1;i=deltax; i++)
{
plot (x ,y);
if e>o then
{
y=y+1;
e=e-1;
}
x=x+1;
e=e+(deltay/ deltax);
}
```

The steps of the algorithm are self-explanatory. After plotting each point, find the error involved, if it is greater than Zero, then in the next step, the next incremental point is to be plotted and error by error-1; else error remains the same and the point will not be incremented. In either case, the other coordinate will be incremented (In this case, it is presented that x - coordinate is uniformly incremented at each stage, while y coordinate is either incremented or retained as such depending on the value of error)

Now we look at the program below that draws the line using Bresneham,s line drawing algorithm.

```
/* Program to generate a line using Bresenham's algorithm */

   # include<stdio.h>
# include<conio.h>
#include<stddlib.h>
# include <graphics.h>

void brline(int,int,int,int);

void main()
{
int gd=DETECT,gm,color:
int x1,y1,x2,y2;
printf("\n enter the starting point x1,y1 :");
scanf("%d%d",&x1,&y1);
```

```
printf("\n enter the starting point x2,y2 :");
scanf("%d%d",&x2,&y2);
clrscr();
initgraph(&gdriver, &gmode,"");
brline(x1,y1,x2,y2);
getch();
closegraph();
}

/* Function for Bresenham's line */

void brline(int x1,int y1,int x2,int y2)
{
int e,l,xend;
int color;
float dx,dy,x,y;
dx=abs(x2-x1);
dy=abs(y2-y1);
if(x1>x2)
{
                x=x2;
y=y2;
                xend=x1;
}
else
{
                x=x1;
                y=y1;
xend=x2;
}
e=2*dy-dx;

while(x<xend)
{
color=random(getmaxcolor());
putpixel((int)x,(int)y,color);
if(e>0)
{
        y++;
e=e+2*(dy-dx);

}
else
            e=e+2*dy;
            x++;
}
}
```

Generation of Circles

The above algorithms can always be extended to other curves - the only required condition is that we should know the equations of the curve concerned in a differential form. We see a few cases.

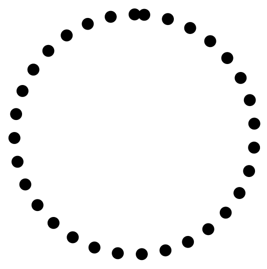
i) A circle generating DDA:

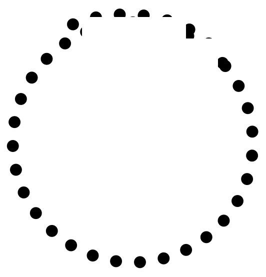
The differential equation of a circle is $\frac{dy}{dx} = -x/y$

Hence by using the above principle, we can implement the circle plotting DDA by using the following set of equations $x_{n+1} = x_n + \varepsilon y_n$ and $y_{n+1} = y_n - \varepsilon x_n$

Where the subscript n refers to the present value and n+1 to the next value to be computed. □y and □x are the increments along the x and y values.

Unfortunately, this method ends up drawing a spiral instead of a circle, because the two ends of a circle do not meet. This is because, at each stage, we move slightly in a direction perpendicular to the radius, instead of strictly along the radius i.e. we keep moving slightly away from the center. So, in the end, we get the closing point a little higher up than where it is required and hence the circle does not close up


Ideal Circle


Drawn by a DDA

However the error can be reduced to a large extent by using the term - x_{n+1} instead of x_n in the second equation.

i.e. $x_{n+1} = x_n + \varepsilon y_n$

$y_{n+1} = y_n - \varepsilon x_{n+1}$

Another way of drawing circles is by using polar coordinators

$x_{n+1} = x_n\cos\theta + y_n\sin\theta$

$y_{n+1} = y_n\cos\theta - x_n\sin\theta$

Of course, each of them has a few minor disadvantages, which are rectified by special algorithms, discussion of which is beyond the scope of the present course.

Here are the programs for generating circles using DDA and Bresneham's algorithms. Also we have given the program to generate spiral.

```
/* Program to demonstrate circle using DDA algorithm */
  # include <graphics.h>
 # include<conio.h>
  # include<dos.h>
  #include<alloc.h>
  #include<math.h>

  void main()
  {
  int gm,gd=DETECT,I,;
  int x,y,x1,y1,j;
  initgraph(&gd,&gm,"");
  x=40;  /*The c0-ordinate values for calculating radius */
  y=40;
  for(i=0;i<=360;i+=10)
  {
          setcolor(i+1);
          x1=x*cos(i*3.142/180)+y*sin(i*3.142/180);
    y1=x*sin(i*3.142/180)-y*cos(I*3.142/180);
    circle(x1+getmaxx()/2,y1+getmaxy()/2,5); /* center of the circle is center
    of the screen*/
      delay(10);
  }
  getch();
  }
```

The following program draws the circle using Bresenham's algorithm.

```
/* program to implement Bresenham's Circle Drawing Algorithm */

  # include<stdio.h>
# include<conio.h>
# include <graphics.h>
# include<math.h>
#include<dos.h>

/* Function for plotting the co-ordinates at four different angles that are placed
at egual distences */

void plotpoints(int xcentre, int ycentre,int x,int y)
{
int color=5;
putpixel(xcentre+x,ycevtre+y,color);
putpixel(xcentre+x,ycevtre-y,color);
putpixel(xcentre-x,ycevtre+y,color);
putpixel(xcentre-x,ycevtre-y,color);


putpixel(xcentre+y,ycevtre+x,color);
putpixel(xcentre+y,ycevtre-x,color);
putpixel(xcentre-x,ycevtre+x,color);
putpixel(xcentre-y,ycevtre-x,color);
}

/* Function for calculating the new points for(x,y)co-ordinates. */

void cir(int xcentre, ycentre, int radius)
{
int x,y,p;
x=0; y=radius;
plotpoints(xcentre,ycentre,x,y);
p=1-radius;
while(x<y)
{
if(p<0)
p=p+2*x+1:
else
{
  y--;
p=p+2*(x-y)+1;
}
x++;
plotpoints xcentre, ycentre,x,y);
delay(100);
}
}
```

/* The main function that takes (x,y) and 'r' the radius from keyboard and activates other functions for drawing the circle */

```
void main()

{
intgd=DETECT,gm,xcentre=200,ycentre=150,redius=5;
printf("\n enter the center points and radius  :\n");
scanf("%d%d%d", &xcentre, &ycentre, &radius);
clrscr();
initgraph(&gd,&gm,"");
putpixel(xcentre,ycentre,5);
cir(xcentre,ycentre,redius);
getch();
closegraph();
}
```

Bresenham specified the algorithm for drawing the ellipse using mid point method. This is illustrated in the following program.

```
/* BBRESENHAM's MIDPOINT ELLIPSE ALGOTITHM. */

  # include<stdio.h>
# include<conio.h>
# include<math.h>

# include <graphics.h>
int xcentre, ycentre, rx, ry;
int p,px,py,x,y,rx2,ry2,tworx2,twory2;
void drawelipse();
void main()
{
```

```
int gd=3,gm=1;
clscr();
initgraph(&gd,&gm,"");
printf("n Enter X center value: ");
scanf("%d",&xcentre);
printf("n Enter Y center value: ");
scanf("%d",&ycentre);
printf("n Enter X redius  value: ");
scanf("%d",&rx);
printf("n Enter Y redius  value: ");
scanf("%d",&ry);
cleardevice();
ry2=ry*ry;
rx2=rx*rx;
twory2=2*ry2;
tworx2=2*rx2;

/* REGION first */
x=0;
y=ry;
drawelipse();

p=(ry2-rx2*ry+(0.25*rx2));
px=0;

py=tworx2*y;
while(px<py)
{
x++;
px=px+twory2;
if(p>=0)
{

       y=y-1;
       py=py-tworx2;
  }
   if(p<0)
        p=p+ry2+px;
   else
           {

     p=p+ry2+px-py;
```