Bresenham type fast algorithm for $3-D$ linear and helical movement in CNC machines

Muhammad Saad Saleem
Research Assistant
*K.I.C.S., U.E.T., Lahore
saad@khwarzimic.org

Prof. Dr. Mian Muhammad Saleem
Department of Electrical Engineering
U.E.T., Lahore
drmsaleem@hotmail.com

March 10, 2004

Abstract

Many of commercially available circuits for CNC machines have digitized input so there is a need to map continuous motions onto discrete planes. This paper appreciates the efficiency of Bresenham algorithm and extends this concept to $3-D$ linear and helical movements.

1 Introduction

When writing a control program for CNC machines, there are two basic motions that needs to be catered. One of them is linear motion represented by G00 and G01 and other one is circular motion represented by G02 (clockwise direction) and G03 (counter clockwise direction) in [1]G-Code notation. Stepper motors move in discrete steps and servo motors depend on the output of encoders which is also discrete. So in both cases we need to deal with planes which are pixel based. One of the approaches to draw circle in $X-Y$ plane is to calculate different values of y for given x from standard circle equation with center at the origin.

$$y = \pm\sqrt{r^2 - x^2}$$

There are two basic flaws in this approach:

1. Due to curvature in circle, there will be spaces between different values of y for consecutive values of x
2. We need to evaluate square root which takes a lot of time to process

One elegant approach to solve this problem is through Bresenham algorithm that involves only integers. We will first discuss that how Bresenham algorithm is implemented in $2-D$ and then how we can extend this concept to $3-D$ to make $3-D$ lines and helical structures.

2 Bresenham line algorithm

The basic idea behind Bresenham algorithm is to divide the whole plane into octants. In each octant, the total displacement along one axis will be greater than the total displacement along the other axis. Figure (1) shows octants in a $2-D$ plane.

For example if we are moving linearly in the first octant, we will move along $x-axis$ more than along $y-axis$. So we can say that in every iteration, we will always move one step along x-axis. The only thing we have to decide whether to take step along y-axis

*I want to thank Prof. Dr. Noor M. Sheikh (Director, K.I.C.S., Dean, Electrical Engg. Deptt., U.E.T., Lahore), his support made it possible for me to complete this paper

[1]RS-274D is a recommended standard for NC machines developed by Electronic Industry Association in the early 1960's

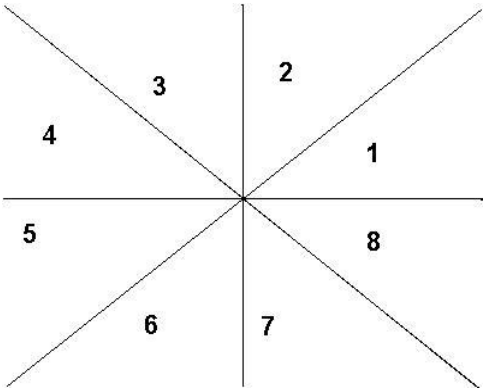

Figure 1: Octants in X-Y plane

or not. So we have reduced eight possible moves to only two.

Now we will derive a mathematical expression that can tell us whether to increase y or not when we are moving in along a line in first octant. We know that the basic line equation in $X - Y$ plane with slope m is given by:

$$y = mx + c$$

This can be rewritten as:

$$\Delta y = m\Delta x \tag{1}$$

Equation 1 tells us that if Δx is 1 unit, then Δy will be the slop of the line. Δy can be seen as an error. This error accumulates after every iteration. Hence

$$\epsilon_y = \epsilon_y + m \tag{2}$$

If

$$\epsilon_y > 1/2 \tag{3}$$

We will increase y by 1 unit and decrease ϵ_y by 1. Equation (2) contains m, which is a float. In order to change the whole equation into integers, we rewrite the equation as:

$$\epsilon_y = \epsilon_y + \Delta y/\Delta x$$

Multiplying whole equation with $2\Delta x$, we get

$$2\Delta x\epsilon_y = 2\Delta x\epsilon_y + 2\Delta y \tag{4}$$

Similarly, equation (3) contains a fraction, so we multiply whole equation by $2\Delta x$, we get

$$2\Delta x\epsilon_y > \Delta x \tag{5}$$

Left handside of equation (4) and equation (5) is same, so we only need to compute $2\Delta x\epsilon_y$. And we also know that initially ϵ_y is 0. So initial value of $2\Delta x\epsilon_y$ is also 0. We only need to compare the values of $2\Delta x\epsilon_y$ without any multiplication but only addition with $2\Delta y$. We compare its value with Δx and if it is greater than Δx then we will take one step in y direction and decreases $2\Delta x\epsilon_y$ by Δx.

During this iteration process, we will only check current value of x with final value of x because total displacement along $x - axis$ is greater than total displacement along $y - axis$. Pseudo-code for this algorithm can be written as:

```
linear([2]int x1, int y1, int x2, int y2);
begin
  DeltaY = y2 - y1;
  DeltaX = x2 - x1;
  2DeltaY = DeltaY + DeltaY;
  2DeltaXErrorY = 0;
  [3]int currentX = x1;
  int currentY = y1;
  while(currentX ≠ x2);
  begin
    currentX += 1;
    2DeltaXErrorY += 2DeltaY;
    if(2DeltaXErrorY > DeltaX);
    begin
      currentY += 1;
      2DeltaXErrorY -= DeltaX;
      moveY(1); [4]
    end
    moveX(1); [5]
```

[2](x1, y1) is initial point and (x2, y2) is final point
[3]Initialzing with initial values
[4]moves along $y - axis$ by 1 unit
[5]$x - axis$ will always move by 1 unit

```
  end
endProcedure
```

This concept can be utilized for movement in other octants. For example, if we are in 3^{rd} octant, the $currentY$ will increase in every iteration and $currentX$ will decrease when $2\Delta y \epsilon_x$ will be greater than Δy. So before any linear movement, we will have to find in which octant this line lies and call the correct procedure to move to our desired point.

Movement in 8^{th} octant can be seen in the figure (2).

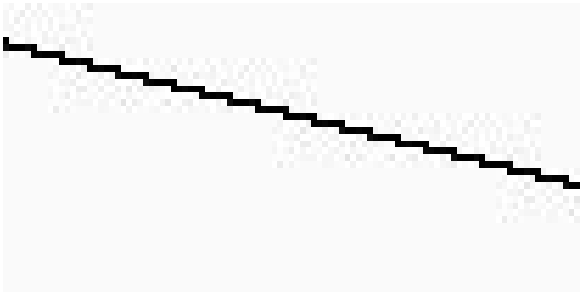

Figure 2: Linear movement from left to right in 8^{th} octant

3 Extension of Bresenham line algorithm for $3-D$ linear motion

We can use the basic concept of Bresenham algorithm to make $3-D$ linear movements. We will emphasize on the basic concept of Bresenham algorithm and that is to first find such axis along which there is total maximum displacement. If, for example, the total maximum displacement is along $y-axis$. Then we will transform a $3-D$ line into two $2-D$ lines with $y-axis$ as common axis. One line will lie in $X-Y$ and other will lie in $Y-Z$ plane. In every iteration, there will always be a movement along $y-axis$. The only question remains that whether there will be displacement along $x-axis$ or along $z-axis$. So we need to make 2 choices out of 4 options rather than making 2 choices from 26 possible moves.

In this approach we will deal with lines in $X-Y$ plane and $Y-Z$ plane as separate lines. This whole algorithm can be generalized for any octant. We will first determine the axis along which there is maximum movement. Then we will find the octants of these two lines separately and apply our previous knowledge of $2-D$ lines. Pseudo-code for this algorithm can be written as:

```
3DLinear(int x1, int y1, int z1, int x2,
           int y2, int z2);
begin
  [6]int incY = 1;
  int incX = 1;
  int incZ = 1;
  DeltaX = x2 - x1;
  DeltaY = y2 - y1;
  DeltaZ = z2 - z1;
  2DeltaX = DeltaX + DeltaX;
  2DeltaY = DeltaY + DeltaY;
  2DeltaZ = DeltaZ + DeltaZ;
  int currentX = x1;
  int currentY = y1;
  int currentZ = z1;
  [7]int ErrorY = 0;
  int ErrorX = 0;
  int ErrorZ = 0;
  if(DeltaX < 0);
  begin
    incX = -1;
    DeltaX = -DeltaX;
    2DeltaX = -2DeltaX;
  end
  if(DeltaY < 0);
  begin
    incY = -1;
    DeltaY = -DeltaY;
    2DeltaY = -2DeltaY;
  end
```

[6]These variables will tell whether to increase or decrease during iterations

[7]Initializing error parameter, just like in $2-D$ linear movement

```
  if(DeltaZ < 0);
  begin
    incZ = -1;
    DeltaZ = -DeltaZ;
    2DelatZ = -2DeltaZ;
  end
  if(DeltaZ > DeltaX & DeltaZ > DeltaY)
  begin
    [8]while(currentZ ≠ z2);
    begin
      [9]currentZ += incZ;
      [10] ErrorX += 2DeltaX;
      ErrorY += 2DeltaY;
      [11] if(ErrorX > DeltaZ);
      begin
        currentX += incX;
        errorX -= 2DeltaZ;
        moveX(incX);
      end
      if(ErrorY > DeltaZ);
      begin
        currentY += incremenetY;
        errorY -= 2DeltaZ;
        moveY(incY);
      end
    end
    ⋮
{Other two cases can be implemented in
similar way}
    ⋮
  end
endProcedure
```

[8]Because DeltaZ will have the maximum total displacement, so we will only check currentZ that whether it has reached final point or not

[9]currentZ will increase in every iteration

[10]Calculating error for two separate lines like we have found in $2-D$ linear movement

[11]Checking whether we will make movement along $x-axis$

4 Bresenham Circle Algorithm

Bresenham circle algorithm works in similar fashion. We divide the whole plane into octants. In G-Code, G02 is used for clockwise and G03 is used for counter-clockwise motions. Lets suppose that we want to move along a counter-clockwise circular arc with center at origin, with radius r and our motion lies in first octant, so our limits are from $(r, 0)$ to $(r/\sqrt{2}, r/\sqrt{2})$, where $x = y$. In first octant, we can see that movement along $y-axis$ is more than the movement along $x-axis$. So we can say that during each iteration y will always increase, but the only question, we need to answer is that whether x will decrease or not. We find answer to this question through mathematical manipulation.

We know that the basic circle equation with center at origin is given as:

$$x^2 + y^2 = r^2$$

We can rewrite this equation as:

$$x^2 + y^2 - r^2 = 0$$

If this is not true then it means that we are deviating from the circular trajectory. So we can call it an error, hence:

$$error(x, y) = |x^2 + y^2 - r^2| \tag{6}$$

In the next iteration, there will be a movement along $y-axis$ but we will have to decide whether we will move along $x-axis$ or not. We will find error for $x-1$ and for x. If the error for the latter will be less, then we will not move, and if not, we will move along $x-axis$. Let the two errors be $error(x-1, y+1)$ and $error(x, y+1)$. We will decrement x and move along $x-axis$ if and only if:

$$|error(x-1, y+1)| < |error(x, y+1)| \tag{7}$$

if and only if

$$|(x-1)^2 + (y+1)^2 - r^2| < |x^2 + (y+1)^2 - r^2| \tag{8}$$

if and only if

$$\{(x-1)^2+(y+1)^2-r^2\}^2 < \{x^2+(y+1)^2-r^2\}^2 \quad (9)$$

if and only if

$$2(1-2x)(x^2+y^2-r^2+(1+2y))+(1-2x)^2 < 0 \quad (10)$$

If $(1-2x) < 0$, then inequality sign will change to $>$ and we will eliminate $(1-2x)$ from above equation. Similarly, we can determine equation for $3rd$ octant. It can be written as:

$$2(1-2y)(y^2+x^2-r^2+(1-2x))+(1-2y)^2 < 0 \quad (11)$$

Here we will eliminate $(1-2y)$. If we examine equation (10) , equation (11) and equations in other octants, we find that sign with x and y shows that this quantity is increasing or decreasing in a particular octant. So before we actually implement this algorithm, we will first find that whether x and y are increasing or decreasing. If our circular arc lies in $1st, 4th, 5th$ or $8th$ octant, we will follow equation (10) format and if it lies in $2nd, 3rd, 6th$ and $7th$ octant, we will follow, equation (11) format. Figure (3) shows a circular arc in first octant.

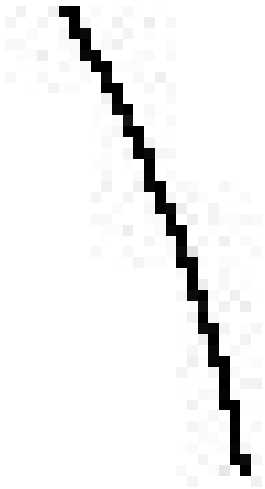

Figure 3: Circular movement in first octant

Pseudo-code for this algorithm can be written as:

```
arc(int x1,int y1,int x2,int y2,int r);
  int x = x1;
  int y = y1;
  while(!complete(x, y, x2, y2))
    nextStep(x, y, r);
endProcedure

nextStep(int x, int y, int r);
begin
  int incX = 1;
  int incY = 1;
  int incZ = 1;
  if(y > 0)
    incX = -1;
  else
    incX = 1;
  if(x>0)
    incY = 1;
  else
    incY = -1;
  if(|x| > |y|)
  begin
    if(incX*x>0)
      if(2*(1+incX*2*x)*(x*x+y*y-r*r+1+incY*2*y)
        +(1+incX*2*x)<0)
      begin
        x += incX;
        moveX(incX);
      end
    else if(incX*x<0)
      if(2*(1+incX*2*x1)*(x*x+y*y-r*r
        +1+incY*2*y1) +(1+incX*2*x1)>0)
      begin
        x += incX;
        moveX(incX);
      end
  y += incY;
  moveY(incY);
  end
  if(|x| < |y|)
  begin
    if(incY*y>0)
      if(2*(1+incY*2*y)*(x*x+y*y-r*r
        +1+incX*2*x) +(1+incY*2*y)<0)
      begin
        y += incY;
```

```
      moveY(incY);
    end
  else if(incY*y<0)
    if(2*(1+incY*2*y)*(x*x+y*y-r*r
      +1+incX*2*x) +(1+incY*2*y)>0)
    begin
      y += incY;
      moveY(incY);
    end
  x1 += incX;
  moveX(incX);
 end

endProcedure
```

```
[12]bool complete(int x,int y,int x2,int y2);
begin
 if([13]check(x, y, x2, y2));
  if(|x| > |y|);
    if(y = y2);
      return true;
  else
    if(x = x2);
      return true;
 else
  return false;
endProcedure;

bool check(x, y, x2, y2);
begin
 if((x>0&x2>0)||(x<0&x2<0);
  if((y>0&y2>0)||(y<0&y2<0);
    if(((|x|>|y|)&(|x2|>|y2|))||
      ((|x|<|y|)&(|x2|<|y2|)));
      return true;
 else
  return false;
endProcedure
```

The procedure "complete" informs the program, when to stop. It might be possible that due to even very small error, the final point might not lie on the circular arc and program iterates for ever. But this procedure forces the program to stop to the nearest point. In first octant, $(1 - 2x)$ decreases by two units in every iteration, so we can simply calculate its initial value and decrease it by two units in every iteration instead of actually calculating its value. This way we can make this algorithm efficient just like we did in Bresenham Line Algorithm. But one of the major difference between Bresenham Line Algorithm and Bresenham Circle Algorithm is that in circle algorithm, we have to calculate in every iteration that in which octant current point lies but in line algorithm we need to calculate octant in the beginning only.

[12]this procedure returns a boolean

[13]checking whether current and final points lie in same octant

5 Extending Bresenham Circle and Line algorithms to generate helical motion

Helical structure can be considered as a $2 - D$ line. One of the axis is along z direction and other one is along the circular arc. First we need to know that along which axis, there is maximum total displacement. To find number of steps along circular arc, we will make a dummy procedure, that works just like our previous circle algorithm except it does not generate any movement. Let n_c be the total number of iterations required to make a circular arc, and let n_z be the total number of iterations required to make displacement along $z - axis$. If n_z is greater than n_c then during every iteration, we will take a step along $z - axis$ but we will have to calculate whether we will take a step along circular arc or not. Similarly, if n_c is greater than n_z, then we will take a step along circular arc in every iteration. This line is treated as a normal line, and we will find its octant and then proceed further with normal operations. Pseudo-code for this algorithm can be written as:

```
helical(int x1, int y1, int z1, int x2,
         int y2, int z2, int r);
begin
```

```
[14]integer nc = dummy(x1, y1, x2, y2, r);
int nz = z2 - z1;
int 2nc = nc + nc;
int 2nz = nz + nz;
int currentZ = z1;
int currentCircleStep = 0;
int errorCircle = 0;
int errorZ = 0;
int incZ = 1;
if(nz < 0);
begin
  incZ = -1;
  nz = -nz;
  2nz= -2nz;
end
if(nz > nc);
begin
  currentZ += incZ;
  errorCircle += 2nc;
  if(errorCircle > nz);
  begin
    currentCircleStep += 1;
    errorCircle -= 2nz;
    [15]nextStep(currentX, currentY, r);
  end
  moveZ(incZ);
end
else
begin
  currentCircleStep += 1;
  errorZ += 2nz;
  if(errorZ > nc);
  begin
    currentZ += incZ;
    errorZ -= 2nc;
    moveZ(incZ);
  end
  nextStep(currentX, currentY, r);
end
endProcedure
```

Above mathematics is developed with the assumption that circle has its center at origin, which is usually not true. But above algorithms can still be applied with little change.

If (x_i, y_i) is current coordinate of a circle, then we simply subtract the center from these coordinates and apply above algorithm without any further change. If (x_c, y_c) is center of a circle, then it can be written that:

$$x = x_i - x_c \tag{12}$$

$$y = y_i - y_c \tag{13}$$

6 Conclusion

Bresenham algorithm is an efficient algorithm for mapping continuous motions onto discrete plane. This solution can easily be implemented in micro controllers and other processors in which there is no floating point unit. So by using Bresenham algorithms for $3 - D$ motions, we can save cost and improve performance of CNC machines.

References

[1] Kennedy, John, *Bresenham integer Only Line Drawing Algorithm,* (Santa Monica College, Santa Monica, CA 90405, rkennedy@ix.netcom.com)

[2] Kennedy, John, *A Fast Bresenham Type Algorithm For Drawing Circles,* (Santa Monica College, Santa Monica, CA 90405, rkennedy@ix.netcom.com)

[3] Glenn, Rowe, *Computer Graphics with JAVA*

[14]dummy procedure calculates total steps required to draw a circular arc

[15]same procedure which we used in circle algorithm