The CORDIC Method for Faster *sin* and *cos* Calculations

Michael Bertrand

CORDIC stands for Coordinate Rotation Digital Computer, an early device implementing fast integer sine and cosine calculations (Volder, 1959). Whenever trigonometry functions must be evaluated repeatedly, as in computer graphics, integer methods, such as CORDIC, should be considered. While integer methods are less accurate than the C Library functions `sin` and `cos`, the improved speed makes the tradeoff quite acceptable in some applications.

CORDIC Units

The key to the CORDIC method is the representation of both angles and trigonometric ratios as integers. In this implementation a 16-bit unsigned integer represents the angles around the circle as shown in Table 1.

With CORDIC angle units, or CAUs, the circle divides into 64K parts instead of 360 parts using degrees. Each degree measures about 182 CAU.

Sine and cosine values are represented as signed integers, with an implicit denominator of 16384 (*CordicBase* in the accompanying program). Calculated sines and cosines lie in the range -16384 to +16384, corresponding to a trigonometric ratio between -1 and +1. Table 2 contains sample correspondents in this fixed point scheme.

Suppose your application receives a value of 100 and must multiply it by *sin(54°)* to produce the nearest integer (a realistic example from computer graphics where input and output are pixel locations). The standard approach, using the C Standard Library `sin` call, amounts to:

```
100 × sin(54°) = 100 × 0.8090 = 80.90 -> 81
```

Both the floating-point multiplication and the `sin` are expensive.

The CORDIC version of this calculation substitutes a fast *sin* and long integer multiplication (where 54° = 9830 CAU and sin(9830 CAU) = 13255 CORDIC fixed-point units):

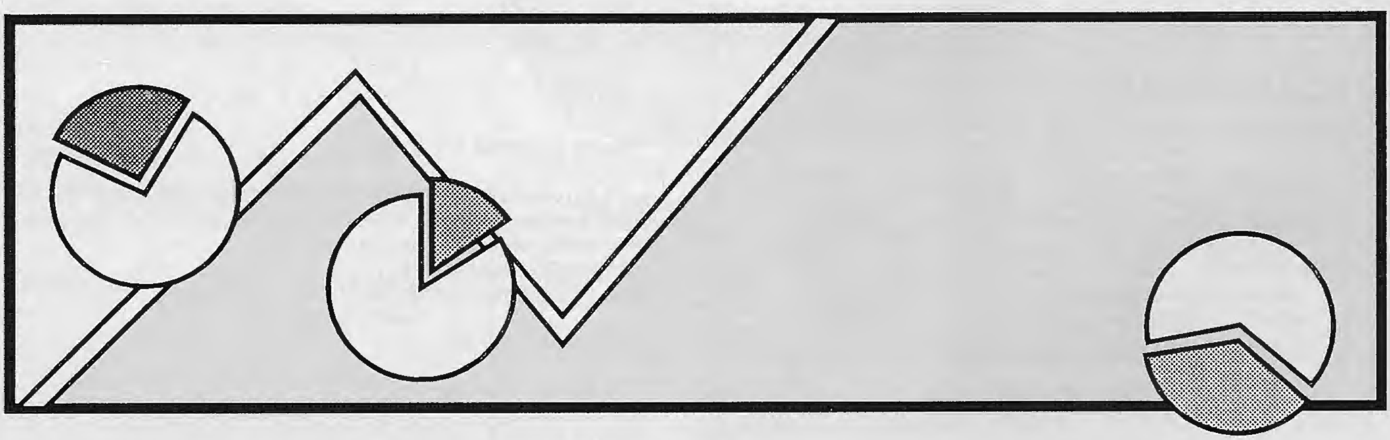
```
100 × sin(9830) = (100 × 13255 + 8192) / 16384 =
81.40 -> 81
```

The 8192 is added for integer rounding (8192/16384 = 0.5). The division by 16384 is done with an inexpensive right shift. CORDIC's speed is due to the fast sine calculation and the complete avoidance of floating-point calculations.

CORDIC Special Angles

The CORDIC algorithm depends on representing a given angle by a set of special angles, $\{\arctan(2^{-i})\} = \{\arctan(1), \arctan(1/2), \arctan(1/4), \ldots\}$:

```
arctan(   1) = 45.00° = 8192 CAU
arctan( 1/2) = 26.57° = 4836 CAU
```



Mike teaches Mathematics and programming at Madison Area Technical College, Madison, WI 53704.

```
arctan( 1/4) = 14.04° = 2555 CAU
arctan( 1/8) =  7.13° = 1297 CAU
arctan(1/16) =  3.58° =  651 CAU
                 .
                 .
                 .
```

Using these special angles, 54° is represented to a finer and finer precision as follows:

```
54° = 45.00                                       = 45.00
54° = 45.00 + 26.57                               = 71.57
54° = 45.00 + 26.57 - 14.04                       = 57.53
54° = 45.00 + 26.57 - 14.04 - 7.13                = 50.40
54° = 45.00 + 26.57 - 14.04 - 7.13 + 3.58         = 53.98
```

This approximation has a physical interpretation. Think in terms of a vector 16384 units long and emanating from the origin in a standard x-y coordinate system. Starting at 0° along the positive x axis, the vector rotates through each of the special angles one step at a time. Rotation at each step may be clockwise or counter-clockwise, whichever is needed to bring the vector closer to 54°. The special angles represent rotations by smaller and smaller amounts, with positive signifying

Table 1 *16-bit unsigned integers representing the angles around the circle*

degrees	CORDIC angle units (CAU)
0	0
45	8192
90	16384
180	32768
270	49152

Table 2 *Sample correspondents in a fixed-point scheme where calculated sines and cosines lie in the range -16384 to +16384, corresponding to a trigonometric ratio between -1 and +1*

decimal	CORDIC fixed point units
-0.8	-13107
-0.5	-8192
0.0	0
0.25	+4096
+0.5	+8192

Listing 1 CORDIC.C

```
/*  CORDIC.C : Demonstrates the integer-based CORDIC
 *    system for calculating sines and cosines. The
 *    vertices of a regular hexagon are calculated using
 *    CORDIC trig and the hexagon itself is rotated.
 *    Make in Borland C++'s internal environment.
 *    (c) 1991 Michael Bertrand.
 */

#include <stdio.h>     /* printf */
#include <math.h>      /* sqrt, atan */
#include <conio.h>     /* getch */
#include <graphics.h>  /* BGI */

typedef struct
{
  int x;
  int y;
} POINT;

typedef unsigned int WORD;

void CalcHexPtsCORDIC(POINT center, POINT vertex);
void DrawHexagon(POINT center, POINT vertex);
void DrawCross(POINT pt, int colr);
void SinCosSetup(void);
void SinCos(WORD theta, int *sin, int *cos);

#define ESC   0x1B

/* Quadrants for angles. */
#define QUAD1 1
#define QUAD2 2
#define QUAD3 3
#define QUAD4 4
```

counter-clockwise rotation and negative signifying clockwise rotation.

Starting at 0° along the positive x axis, +45.00° is a counter-clockwise rotation into the middle of the first quadrant. The next step again rotates counter-clockwise by another 26.57°, but this results in 71.57°, which overshoots the mark. The third rotation is therefore clockwise, signified by the minus 14.04°, bringing the result back to 57.53°. This continues as many times as there are bits in 16384—14 times.

The x and y coordinates of the rotating vector are the cosine and sine of the vector's angle at each step, assuming that the vector's length, 16384, doesn't change during rotation. As the vector's angle approaches 54° more closely, its x and y coordinates provide better approximations of cos(54°) and sin(54°).

CORDIC Equations

Figure 1 illustrates the geometry of a counter-clockwise rotation at the i^{th} step. Rotate vector *p* through an angle of *arctan*(2^{-i}) to new vector *p1* such that the indicated angle at *p* is a right angle. The task is to express the new (cosine, sine) approximations (x1, y1) in terms of the old ones (x, y). The two shaded triangles are similar because they are right triangles with acute angles that are equal. In the right triangle connecting *p* and *p1* to the origin, the two legs *R** and *R* are in the proportion 1 : 2i by the definition of *arctan(2-i))* :

$$2^{-i} = \tan(\arctan(2^{-i})) = (R^*)/R$$

R and *R** are the hypotenuses of the two similar triangles, so these triangles are in the proportion 1 : 2^{-i}. This implies that the shorter horizontal leg of the small triangle is $y/(2^i)$ and the longer, vertical leg is $x/(2^i)$, leading to the counter-clockwise equations:

```
ccw rotation        x1 = x - y/(2^i)
                    y1 = y + x/(2^i)
```

Rotating *p1* clockwise leads to the clockwise equations, identical except for a sign reversal:

```
cw rotation         x1 = x + y/(2^i)
                    y1 = y - x/(2^i)
```

These are fast operations involving integer addition, subtraction and shifting.

Expansion Factor

A problem arises with *p1*, which is further from the origin than *p* was at each step, so the vector does not simply rotate around a circle of radius 16384, but also expands. The expansion can be exactly measured by applying the Pythagorean theorem to the right triangle containing *p* and *p1*:

$$\begin{aligned} p1^2 &= R^2 + (R/2^i)^2 = R^2 + R^2/2^{2i} \\ &= R^2 * (1+1/2^{2i}) \\ &= R^2 * (2^{2i} + 1) \; / \; 2^{2i} \end{aligned}$$

or:

$$p1 = R * \sqrt{2^{2i} + \frac{1}{2^{2i}}}$$

The first rotation (45°) expands the vector by a factor of $\sqrt{2}$ = 1.414; the second rotation (26.57°) expands it further by a factor of $\sqrt{\frac{5}{4}}$ = 1.118; the third rotation (14.04°) expands by $\sqrt{\frac{17}{16}}$ = 1.031, and so on. Each of the 14 rotations entails such an expansion, although the later ones are negligible.

The net effect is an expansion by a factor of 1.414 × 1.118 × 1.031 × ... = 1.646760. The original vector will expand to 1.646760 × 16384 = 26981 in the course of rotating. To offset this expansion, the original vector is contracted before starting the process to bring its final length, after the rotation/ expansions, back to 16384. Instead of initializing the vector to 16384, it is initialized to *xInit* = 16384 /1.646760 = 9949, which expands to 16384 after 14 steps. At the last step the vector's x and y coordinates will be the cosine and sine in CORDIC fixed-point units (based on 16384).

Implementation Notes

The central routine is *SinCos* (See Listing 1), which calculates the sine and cosine of an incoming angle. Both incoming

Listing 1 continued

```
/* NBITS is number of bits for CORDIC units. */
#define NBITS 14

/* NUM_PTS is number of vertices in polygon. */
#define NUM_PTS 6

int ArcTan[NBITS];       /* angles for rotation */
int xInit;               /* initial x projection */
WORD CordicBase;         /* base for CORDIC units */
WORD HalfBase;           /* CordicBase / 2 */
WORD Quad2Boundary;      /* 2 * CordicBase */
WORD Quad3Boundary;      /* 3 * CordicBase */
POINT HexPts[NUM_PTS+1]; /* calculated poly points */

void main(void)
{
  int    driver;    /* for initgraph() */
  int    mode;      /* for initgraph() */
  WORD   theta;     /* CORDIC angle */
  int    sine;      /* sine of CORDIC angle */
  int    cosine;    /* cosine of CORDIC angle */
  POINT  center;    /* center of hexagon */
  POINT  vertex;    /* hexagon's original base vertex */
  POINT  vertex1;   /* hexagon's changing base vertex */
  POINT  del;       /* vertex - center (radial spoke) */
  int    radius;    /* radius of circumscribing circle */

  driver = VGA;      /* for initgraph() */
  mode   = VGAHI;    /* mode 0x12 : 640x480 16 color */

  if (registerbgidriver(EGAVGA_driver) < 0)
    {
    printf("couldn't find VGA driver");  return;
    }
  initgraph(&driver, &mode, NULL);

  printf("Press ENTER to rotate, ESC to quit.");

  center.x = 320;  center.y = 240;
  vertex.x = 470;  vertex.y = 240;
  radius = vertex.x - center.x;
  /* Calculate the radial spoke : vertex - center. */
  del.x = vertex.x - center.x;
  del.y = vertex.y - center.y;

  setwritemode(XOR_PUT);

  /* Draw circumscribing circle. */
  setcolor(RED);
  circle(center.x, center.y, radius);

  /* Draw small cross at center. */
  DrawCross(center, YELLOW);
  setcolor(WHITE);

  /* Setup CORDIC system, initialize theta = 0. */
  SinCosSetup();
  theta = 0;

  /* Draw initial hexagon. */
  DrawHexagon(center, vertex1 = vertex);

  /* Rotate hexagon. vertex is fixed; vertex1 rotates
   * clockwise around vertex in increments of 650
   * CORDIC units (3.57 deg). CORDIC sines/cosines are
   * used to find vertex1. DrawHexagon() also uses
   * CORDIC sines/cosines to calculate the remaining
   * vertices for each hexagon so they can be drawn.
   */
```

Figure 1 *Illustrates the geometry of a counter-clockwise rotation at the i^{th} step*

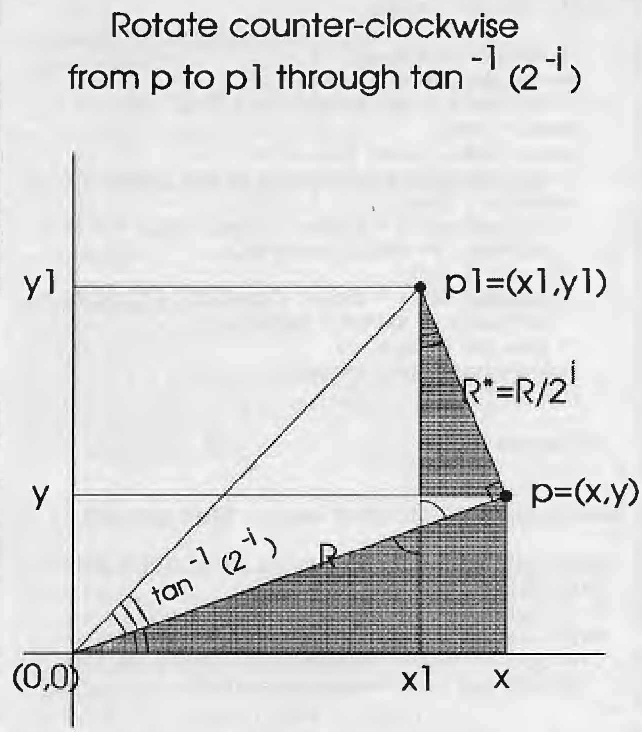
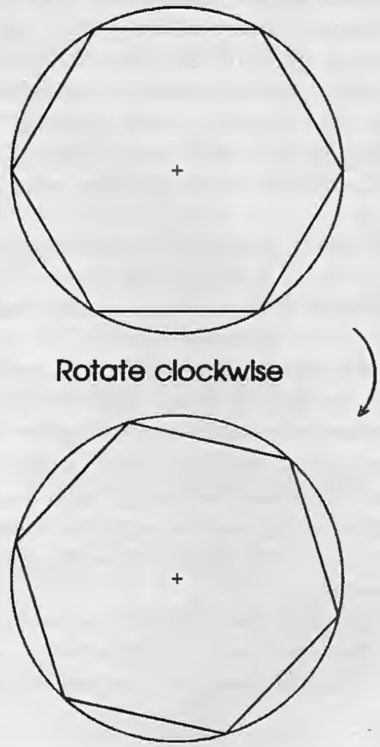


Figure 2 *Hexagon rotator*



angle and calculated sine and cosine are assumed to be in CORDIC units. *SinCosSetup* initializes needed global variables, including the special arctan angles and *xInit*, the initial contracted vector length. *SinCosSetup* must be called once only for initialization before calling *SinCos*. The CORDIC algorithm in *SinCos* works on first quadrant angles only (0 – 90, or 0 – 16383 CAU). *SinCos* translates angles from other quadrants into the first quadrant before applying the algorithm. The calculated sine and cosine will be correct, except possibly for the sign, which is adjusted before returning from the routine.

A hexagon rotator is included to demonstrate the CORDIC system (see Figure 2). A center point and initial vertex remain fixed while another point, *vertex1*, rotates clockwise around the original vertex in increments of 650 CAU (3.57). For each *vertex1*, the five associated regular hexagon vertices are calculated and the hexagon is drawn using Borland C++'s line drawing commands. The routine calculating the vertices, *CalcHexPts*, calculates the sine and cosine of 60° (10923 CAU) only once and calculates each vertex from the previous one. This practice cannot be used for continual rotating because of accumulating errors from integer rounding as well as inexact sines and cosines.

Timing tests show the CORDIC method to be over 20 times faster than the standard method when calculating the vertices of a regular hexagon, a characteristic computer

Listing 1 *continued*

```
  while (getch() != ESC)
    {
    /* Erase last hexagon. */
    DrawHexagon(center, vertex1);
    /* Inc theta by 650 CORDIC units (3.57 deg). */
    theta += 650;
    SinCos(theta, &sine, &cosine);
    /* Calc new vertex by rotating around center. */
    vertex1.x = (int)
       (((long) del.x * cosine - (long) del.y * sine +
       HalfBase) >> NBITS) + center.x;
    vertex1.y = (int)
       (((long) del.x * sine + (long) del.y * cosine +
       HalfBase) >> NBITS) + center.y;
    /* Draw new hexagon. */
    DrawHexagon(center, vertex1);
    }

  closegraph();
}

void CalcHexPtsCORDIC(POINT center, POINT vertex)
/*
  USE:  Calc array of hex vertices using CORDIC calcs.
  IN:   center = center of hexagon.
        vertex = one of the hexagon vertices
  NOTE: Loads global array HexPoints[] with other 5
   vertices of regular hexagon. Uses CORDIC routines
   for trig and long integer calculations.
*/
{
  int   sine;      /* sine of central angle */
  int   cosine;    /* cosine of central angle */
  int   corner;    /* index for vertices of polygon */
  POINT del;       /* vertex - center (radial spoke) */

  /* 60 deg = 10923 CORDIC units. */
  SinCos(10923, &sine, &cosine);

  /* Set initial and final point to incoming vertex. */
  HexPts[0].x = HexPts[NUM_PTS].x = vertex.x;
  HexPts[0].y = HexPts[NUM_PTS].y = vertex.y;

  /* Go clockwise around circle to calc hex points. */
  for (corner = 1; corner < NUM_PTS; corner++)
    {
    /* Calculate the radial spoke : vertex - center. */
    del.x = vertex.x - center.x;
    del.y = vertex.y - center.y;
    /* calc new vertex by rotating around center. */
    vertex.x = (int)
       (((long) del.x * cosine - (long) del.y * sine +
       HalfBase) >> NBITS) + center.x;
    vertex.y = (int)
       (((long) del.x * sine + (long) del.y * cosine +
       HalfBase) >> NBITS) + center.y;
    /* Store new vertex in array. */
    HexPts[corner].x = vertex.x;
    HexPts[corner].y = vertex.y;
    }
}

void DrawHexagon(POINT center, POINT vertex)
/*
  USE:  Draw Hexagon given center and one vertex.
  IN:   center = center of hexagon.
        vertex = one of the hexagon vertices
  NOTE: Call CalcHexPtsCORDIC() to load global array
   HexPts[] with hexagon vertices.
*/
```

Listing 1 *continued*

```
{
  CalcHexPtsCORDIC(center, vertex);
  drawpoly(NUM_PTS+1, (int far *)HexPts);
}

void DrawCross(POINT pt, int colr)
/*
  USE:  Draw cross on screen at pt with given color.
*/
{
  int oldColor;

  setwritemode(COPY_PUT);
  oldColor = getcolor();
  setcolor(colr);
  moveto(pt.x - 2, pt.y); lineto(pt.x + 2, pt.y);
  moveto(pt.x, pt.y - 2); lineto(pt.x, pt.y + 2);
  setcolor(oldColor);
  setwritemode(XOR_PUT);
}

void SinCosSetup(void)
/*
  USE : Load globals used by SinCos().
  OUT : Loads globals used in SinCos() :
   CordicBase    = base for CORDIC units
   HalfBase      = Cordicbase / 2
   Quad2Boundary = 2 * CordicBase
   Quad3Boundary = 3 * CordicBase
   ArcTan[]      = the arctans of 1/(2^i)
   xInit         = initial value for x projection
  NOTE: Must be called once only to initialize before
   calling SinCos(). xInit is sufficiently less than
   CordicBase to exactly compensate for the expansion
   in each rotation.
*/
{
  int i;          /* to index ArcTan[] */
  double f;       /* to calc initial x projection */
  long powr;      /* powers of 2 up to 2^(2*(NBITS-1)) */

  CordicBase = 1 << NBITS;
  HalfBase = CordicBase >> 1;
  Quad2Boundary = CordicBase << 1;
  Quad3Boundary = CordicBase + Quad2Boundary;

  /* ArcTan's are diminishingly small angles. */
  powr = 1;
  for (i = 0; i < NBITS; i++)
    {
    ArcTan[i] = (int)
      (atan(1.0/powr)/(M_PI/2)*CordicBase + 0.5);
    powr <<= 1;
    }
  /* xInit is initial value of x projection to comp-
   * ensate for expansions.  f = 1/sqrt(2/1 * 5/4 * ...
   * Normalize as an NBITS binary fraction (multiply by
   * CordicBase) and store in xInit. Get f = 0.607253
   * and xInit = 9949 = 0x26DD for NBITS = 14.
   */
  f = 1.0;
  powr = 1;
  for (i = 0; i < NBITS; i++)
    {
    f = (f * (powr + 1)) / powr;
    powr <<= 2;
    }
  f = 1.0/sqrt(f);
  xInit = (int) (CordicBase * f + 0.5);
}

void SinCos(WORD theta, int *sin, int *cos)
/*
  USE : Calc sin and cos with integer CORDIC routine.
  IN  : theta = incoming angle (in CORDIC angle units)
  OUT : sin = ptr to sin (in CORDIC fixed point units)
        cos = ptr to cos (in CORDIC fixed point units)
  NOTE: The incoming angle theta is in CORDIC angle
   units, which subdivide the circle into 64K parts,
   with 0 deg = 0, 90 deg = 16384 (CordicBase), 180 deg
   = 32768, 270 deg = 49152, etc. The calculated sine
   and cosine are in CORDIC fixed point units : an int
   considered as a fraction of 16384 (CordicBase).
*/
{
  int quadrant;   /* quadrant of incoming angle */
  int z;          /* incoming angle moved to 1st quad */
  int i;          /* to index rotations : one per bit */
  int x, y;       /* projections onto axes */
  int x1, y1;     /* projections of rotated vector */

  /* Determine quadrant of incoming angle, translate to
   * 1st quadrant. Note use of previously calculated
   * values CordicBase, etc. for speed.
   */
```

graphics task. The savings are due to the elimination of floating-point calculations as well as fast sine and cosine evaluation.

An exhaustive test of all 16384 CAUs shows that the worst error in a sine or cosine is 0.00064 and the average error is 0.00011. This is over 13 bits of accuracy on average, or better than one part in 8000, quite adequate for many screen-related computer graphics applications.

The original papers on this topic read very well, and are accessible through an excellent reprint by IEEE (*Computer Arithmetic*). Also helpful are articles cited below in *Graphics Gems*. Both volumes are essential for computer graphics workers. □

References

Linhardt, R. J., and H. S. Miller 1969. "Digit-by-Digit Transcendental Function Computation". *RCA Rev.* 30:209-247. (Reprinted in E. Swartzlander (ed.) 1990. *Computer Arithmetic*. IEEE Computer Society Press, 233-271.)

Ritter, Jack. "Fast 2D-3D Rotation". In A. S. Glassner 1990. *Graphics Gems*. Academic Press, 440-441.

Turkowski, Ken. "Fixed-Point Trigonometry with CORDIC Iterations". In A. S. Glassner 1990. *Graphics Gems*. Academic Press, 494-497.

Volder, Jack E. 1959. "The CORDIC Trigonometric Computing Technique". *IRE Trans. Electron. Comput.* EC-8:330-334. (Reprinted in E. Swartzlander (ed.) 1990. *Computer Arithmetic*. IEEE Computer Society Press, 226-230.)

Listing 1 *continued*

```
  if (theta < CordicBase)
    {
    quadrant = QUAD1;
    z = (int) theta;
    }
  else if (theta < Quad2Boundary)
    {
    quadrant = QUAD2;
    z = (int) (Quad2Boundary - theta);
    }
  else if (theta < Quad3Boundary)
    {
    quadrant = QUAD3;
    z = (int) (theta - Quad2Boundary);
    }
  else
    {
    quadrant = QUAD4;
    z = - ((int) theta);
    }

  /* Initialize projections. */
  x = xInit;
  y = 0;

  /* Negate z, so same rotations taking angle z to 0
   * will take (x, y) = (xInit, 0) to (*cos, *sin).
   */
  z = -z;

  /* Rotate NBITS times. */
  for (i = 0; i < NBITS; i++)
    {
    if (z < 0)
      {
      /* Counter-clockwise rotation. */
      z += ArcTan[i];
      y1 = y + (x >> i);
      x1 = x - (y >> i);
      }
    else
      {
      /* Clockwise rotation. */
      z -= ArcTan[i];
      y1 = y - (x >> i);
      x1 = x + (y >> i);
      }

    /* Put new projections into (x,y) for next go. */
    x = x1;
    y = y1;
    }  /* for i */

  /* Attach signs depending on quadrant. */
  *cos = (quadrant==QUAD1 || quadrant==QUAD4) ? x : -x;
  *sin = (quadrant==QUAD1 || quadrant==QUAD2) ? y : -y;
}
/* End of File */
```