UNDERSTANDING AND PLOTTING POLAR COORDINATES

By -
Steelweaver52 -- Tom Nally

This article first appeared in the Liberty BASIC Newsletter Issue #96, May, 2002.

Companion demonstration programs:

- Atan2.bas
- Polar1.bas

POLAR PHUN: AN INTRODUCTION

I suspect that I was daydreaming in math class the first time polar coordinates were discussed in my presence. And I'm darned if I can remember whether that was junior high school, high school or college. Suffice it say that my discussion of polar coordinates herein will probably not resemble what students are typically taught in school. Instead, I will share my own understanding of the polar coordinate system, and how one might draw polar graphs in Liberty BASIC programs.

If any of this information resembles what we are taught in school, consider that purely coincidental.

Basically, here are the six questions that I will discuss:

A. What are polar coordinates?
B. What units are used in polar coordinates?
C. How does a programmer convert from polar to Cartesian coordinates?
D. How does a programmer convert from Cartesian to polar coordinates?
E. How does one plot polar coordinate points in an Liberty BASIC GraphicBox?; and
F. How are polar coordinates useful in the real world?

A. WHAT ARE POLAR COORDINATES?

The polar coordinate system is a way to determine the location of a point on a plane given a known point and a known reference line. We might refer to the known point as the "pole", the "origin" or the "center of the circle". (In this discussion, I will call it the "origin".) The known reference line is any line that passes through the origin that we are willing to assign an angle of zero. Typically, if we are drawing a polar coordinate system on a sheet of paper, the reference line is a vector which starts at the origin, and goes to the right. If this sheet of paper happens to be a map with North at the top, then the reference line would start at a known point, then head in a cardinal direction, such as due East or due North.

Any point, Q, on a polar coordinate plane can be located by finding it's angular distance from the reference line, and its linear (or "radial") distance from the origin. What is the angular distance to point Q? Well,

imagine yourself standing at the origin of our polar coordinate system with your right arm pointed along our reference line. Then, sweep your arm in a counterclockwide direction until you are now pointing at point Q. The angle through which your arm swept is the angular distance to point Q. This value is also called the "angular coordinate of Q", and we often name this angle by giving it a greek letter, such as phi or theta. Herein, we will call the angular coordinate of a point "phi".

The radial distance to point Q is the straight-line distance that you would have to walk at the angle "phi" in order to arrive at point Q. This distance is often called the "radial coordinate of Q". This coordinate is often given the name "R".

So, in a polar coordinate system, any point Q can be defined by its polar coordinates, phi and R. A common notation used to define Q by use of Q's polar coordinates looks like this:

Q(phi, R)

The polar coordinate system can be compared to the "cartesian coordinate system", which is a system with which most programmers have some familiarity. The cartesian system also uses two values to define the location of a point--point Q, again--on a plane. One of these values is Q's offset from the Y axis, which is known as Q's "x coordinate" because it represents Q's distance from the origin along a line which is parallel to the X axis. The second value is Q's offset from the X axis, which is known as Q's "y coordinate" because it represents Q's distance from the origin along a line parallel to the Y axis. In the cartesian system, the notation used to define Q by reference to it's cartesian coordinates looks like this:

Q(x, y)

To help us understand these systems a little further, I will provide my robot, Bob, instructions for finding the stump in my back yard using each of these two systems. First, the polar coordinate system.

I will choose my back porch as the origin of my polar system. My arbitrary reference line will be a vector which begins at my porch, and goes directly through my bird bath, which is due East of my porch.

Bob, here are my instructions for finding the stump in my backyard:

(1) Bob, stand at the origin, my porch, and face the bird bath.
(2) Bob, rotate your robot frame 36.87 degrees counterclockwise.
(3) Bob, along your new heading, motor outward a distance of exactly 50 feet.
You are now at the stump.

Bob just found the stump by virtue of his ability to follow instructions given in a polar coordinate frame of reference. Let's put Bob to work again, but use a cartesion system this time.

Once more, I will choose my back porch for the origin of this system. For the X axis, I will choose the vector that begins at the origin and passes through my bird bath, which is due East of the porch. The Y axis of my system is a vector that begins at my porch, and is located on a line which is 90 degrees counterclockwise from my X-axis.

Bob, here are my new instructions for finding the stump in my backyard:

(1) Bob, stand at the origin, and motor along the X axis a distance of exactly 40 feet.
(2) Bob, rotate your robot frame counterclockwise so that you now have a heading which is parallel to the Y axis.
(3) Bob, motor along this new heading exactly 30 feet. You are now at the stump.

Bob has now successfully navigated to the stump twice after being given instructions using both coordinate systems.

Bob, take a bow. (But don't get cocky.)

B. WHAT UNITS ARE USED IN POLAR COORDINATES?

Above, we stated that a polar coordinate consists of a pair of values. One value is an angle (called "phi" in this article), while the second value is a distance from the origin (refered to as "R" herein).

There is really no convention for the distance units to use when describing R. It could be milimeters, meters, inches, feet, furlongs or miles. It could even be astronomical units or parsecs if we are using a polar coordinate system to discuss objects in space. Since any reader of this article may at some point use polar coordinates to plot data in Liberty BASIC, the R unit for that programming endeavor will probably be, well, pixels. In other words, the problem you are solving will define what units to use for the R value.

With regard to the angular unit, I typically start my solution to the problem by thinking about phi in terms of degrees. This is simply because degrees are so familiar to most of us.

If I use a computing language to solve my problem, however, at some point I will probably need to convert any degree measurements into an alternate measure of angular distance called "radians". This is because trig functions which need an angle as an argument will want to see that angle expressed in radians. Likewise, those trig functions that "output" an angle --such as ATN()-- will return an angle expressed in radians.

So, just what is a "radian"?

To understand radians, let's conduct this "thought exercise". On a sheet of paper, use a compass to draw a circle. Make the radius any size you want. In this thought exercise, it doesn't matter how big the circle is, as long as it fits on the paper. Now, cut seven pieces of string whose length is exactly the same length as the radius of the circle. We are going to use these pieces of string as instruments of measure. In fact, we will call them "radians" because the length of each piece corresponds exactly to the length of the radius of the circle.

Next, let's attempt to measure the length of the circumference of the circle by snuggly wrapping these radians end to end around the circumference of the circle.

Of course, we couldn't really wrap these strings "snuggly" around the circumference of a circle on a flat sheet of paper. That's why this is a "thought exercise" rather than a real exercise. But if we could do that, we would find that it takes exactly 2*pi (or 6.2832) radians to wrap exactly one time around the circumference of the circle. In other words, the 360 degree arc of the circle corresponds exactly to 2*pi radians. This will be true regardless of the radius of the circle.

Next in our thought exercise, we want to find out what portion of a radian corresponds to a 30 degree arc when pulled snuggly along the circumference of the circle.

To accomplish this, we need to draw a reference line on our circle, then draw an angle which is 30 degrees to our reference line. Next, we hold one end of our radian where the reference line intersects the circumference of the circle, and pull it snuggly around the circle's circumference until it wraps past the 30 degree line. If we were able to do this accurately, we would find out that it takes exactly 0.5236 radians to wrap along an arc corresponding to 30 degrees. Again, this will be true regardless of the radius of the circle.

If we repeated this exercise for any number of angles, we would observe that each unique angle sweeps out a unique length of arc at the circumference of the circle, and that this arc length can be expressed in terms of radians. It follows, then, that any angle itself can be expressed in terms of radians. This is how radians serve as a substitue for the degree measure of an angle.

Fortunately, we don't need to cut a piece of string every time we want to find the radian measure of an angle, provided that we already know the degree measure of the
angle. Recall that we have already established that 360 degrees corresponds exactly with 2*pi or 6.2832 radians. Therefore, 1 degree corresponds to 6.2832/360, or 0.01745 radians. Here is a conversion example: a 27 degree angle would be identical to 27*0.01745 or 0.4712 radians.

Since 0.01745 is a conversion factor that's pretty hard to remember, when I convert from degrees to radians, I typically just multiply the degree measure by 2 * pi, then divide by 360 to get radians. Works every time:

```
27 degrees = (27/360) * (2 * pi) radians
```

C. HOW DOES A PROGRAMMER CONVERT FROM POLAR TO CARTESION COORDINATES?

I am happy to report that it will only take me a few words to describe how polar coordinates can be converted to cartesian coordinates. First, recall that the notation for a polar coordinate pair looks like this:

```
Q(phi, R)
```

To convert this polar coordinate to a cartesian coordinate requires two operations. One operation finds the

X coordinate in the cartesian system:

```
X = R * cos(phi)
```

The second operation finds the Y coordinate of the cartesian system:

```
Y = R * sin(phi)
```

Not too hard, that.

D. HOW DOES A PROGRAMMER CONVERT FROM CARTESIAN TO POLAR COORDINATES?

Converting from cartesian coordinates to polar coordinates is not quite as simple. The easy part is finding "R" which is simply accomplished by using the Pythagorean Theorem, also called the "distance formula":

```
R = sqrt(x^2 + y^2)
```

Given only x and y, finding phi is a little more difficult. The ATN() function is useful but not sufficient. ATN() only returns radian values from -pi/2 through +pi/2, which corresponds to -90 through 90 degrees.

Here's an example which will illustrate the problem. If

```
x = -1
```

and

```
y = -1
```

then

```
y/x =1
```

and the

```
ATN(y/x) = pi/4 or 45 degrees
```

This is a problematic answer because we know that the cartesian coordinate pair Q(-1,-1) actually corresponds to an angle of 5 * pi/4 or 225 degrees.

The difficulty is caused by the fact that ATN() only takes a single argument. If the argument is positive, then ATN() returns a value between 0 and pi/2 (0 and 90 degrees). If the argument is negative, then ATN returns a value between -pi/2 and 0 (-90 and 0 degrees). Therefore, ATN() gives disproportionate attention to those angles in only half of the circle, while blatantly ignoring the feelings of those angles in the other half where low self-esteem, sadly, is a chronic problem.

Some languages (MS Excel comes to mind) get around this problem by offering a more comprehensive arctangent function called ATAN2(). ATAN2() takes two arguemtns, x and y, which is sufficient information to return the correct value for phi. ATAN2() would be used like this in a program:

```
X = 25
Y equals -60
phi = ATAN2(X, Y)
```

In the companion demo, ATAN2.bas, I provide a custom ATAN2() function written for Liberty BASIC. It takes x and y as arguments, and will return phi in radians. In this function, x or y (or both) must be non-zero. If both x and y are zero, then ATAN2() should not be called, because the point in question exists at the origin of the system where phi is meaningless.

Here's an exercise for the ambitious: write an ATAN2$() function which returns the string value of phi if Q is not at the origin, but returns "Q is at the origin" if Q is at the origin. The usefulness of this function is that there would be no restrictions on the arguments.

E. HOW DOES ONE PLOT POLAR COORDINATE POINTS IN A Liberty BASIC GRAPHICBOX?

I find polar systems fun primarily because some trig functions plotted within polar systems produce interesting curves. (Play with the Liberty BASIC program "PolarPhun" to see a few of 'em.) So, how is the plotting of polar data accomplished? Plotting polar data is identical to plotting cartesian data, except that at some point during the routine, the data must be converted from polar coordinates to cartesian coordinates.

Here are the five steps that I go through, followed by a discussion of each step. In a second companion program to this article, Polar1.bas, I provide source code which implements these steps.

1. Generate the polar data by defining an equation using trig functions, then generating a series of polar coordinate pairs within a For/Next loop.
2. Convert the polar data into cartesian data within another For/Next loop.
3. Scale the data up or down by multiplying the x and y cartesian values by a scalefactor, so that the data will plot nicely.
4. "Shift" or "offset" the data points so that they print about the center pixel of the GRAPHICBOX.

5. Plot the data within the GRAPHICBOX.

1. The first step in generating the data is defining an equation containing one or more trig functions. A simple function with which to start might look like this:

```
R = 2.5 + 2*sin(12*phi)
```

This function will plot a daisy-like graph with 12 petals.

In general, I prefer to use only the sin() and cos() trig functions in my polar equations, while excluding the tan() function. One reason for this is that the sin() and cos() functions will always return a value between -1 and 1. That enables the programmer to control the magnitude of R. In the equation shown just above, R will never be greater than 4.5 (which occurs when sin(12*phi) = 1), nor will it ever be less than 0.5 (which occurs when sin(12*phi) = -1). On the other hand, the tan() function will return a value between minus infinity and plus infinity. This not only makes graphing the output more difficult, but it also risks a program crash due to an overflow error. For example, the tangent of pi/2 is infinity.

A simple programming routine to generate a set of polar data pairs may look like this:

```
DIM R(360)
DIM x(360)
DIM y(360)
DIM xscaled(360)
DIM yscaled(360)
DIM xplot(360)
DIM yplot(360)
pi = 3.14159

for degree = 0 to 360
    phi = (degree/360) *(2*pi)
    R(degree) = 2.5 + 2*sin(12*phi)
next degree
```

2. Converting polar coordinate pairs to cartesian coordinate pairs is accomplished exactly as described in section C above. The source code for that operation might look like this:

```
for degree = 0 to 360
     phi = (degree/360) * (2*pi)
     x(degree) = R(degree) * cos(phi)
     y(degree) = R(degree) * sin(phi)
next degree
```

When this loop has concluded, we will have a set of 361 cartesian coordinate pairs that have been

converted from polar coordinate pairs.

3. As indicated above, R in this equation will always occur within a range of 0.5 to 4.5. If you examine the x and y values coming out of the loop just above, you will observe that x and y will always occur within a range of -4.5 to +4.5. Note that if we plotted our data without scaling it first, all the points would plot within a small, tight box which will be 9 pixels high by 9 pixels wide. That's a pretty small box to hold 360 data points. Certainly, the plotted data would have more meaning for us if we could scale it up so that all the plotted points were discernable. But how large should we scale it?

Say that we are plotting to a Graphicbox that is 200 pixes wide by 200 pixels high. If we want to make full use of the real estate provided by this Graphicbox, it might be nice to plot data to within 10 pixels of the border of the graphic box. Therefore, any scaled value for x and y should be no larger than 90, nor smaller than -90. Why these two values? Because we are going to place our origin at the center of the Graphicbox, which is the pixel with the coordinates

```
(x=100, y=100)
```

Therefore, the largest plotted x value will be no greater than

```
100 + 90 = 190
```

and the smallest plotted x value will be no less than

```
100 - 90 = 10
```

The same goes for the plotted y values.

To review, the largest x value as it comes out of the function will be 4.5, but the largest x value after scaling will be 90. So, the scale factor that must be applied to all x and y values is 20.

Whatever the scale factor happens to be, we don't want to calculate it "by hand", we want the computer to calculate the scale factor. To do that, we have to find the most extreme value of all x and y, whether that value be positive or negative. Here's one way to do that:

```
xmax = 0
xmin equals 0
ymax = 0
ymin equals 0

for degree = 0 to 360
    if (x(degree) > xmax) then xmax = x(degree)
    if (x(degree) < xmin) then xmin = x(degree)
```

```
    if (y(degree) > ymax) then ymax = y(degree)
    if (y(degree) < ymin) then ymin = y(degree)
next degree
```

At this point, we've determined the largest and smallest of all x and y. Now we have to find the value which is largest in magnitude. The value which is largest in magnitude will be stored in a variable called BiggestXY. The coding below will take care of that task:

```
BiggestXY = 0
If (abs(xmax) > BiggestXY) then BiggestXY = abs(xmax)
If (abs(xmin) > BiggestXY) then BiggestXY = abs(xmin)
If (abs(ymax) > BiggestXY) then BiggestXY = abs(ymax)
If (abs(ymin) > BiggestXY) then BiggestXY = abs(ymin)
```

Of the 361 pairs of (x, y) values, we've now found the most extreme value of all of them, and we've placed it in a variable called BiggestXY. Now, we can establish the scalefactor which, when multiplied against BiggestXY, will yield a value of 90.

```
Scalefactor = 90 / BiggestXY
```

At this time, we can multiply all of x() and y() by the scalefactor. This will produce a dataset with magnitudes that will use most of the real estate of the Graphicbox.

```
for degree = 0 to 360
    xscaled(degree) = Scalefactor * x(degree)
    yscaled(degree) = Scalefactor * y(degree)
next degree
```

4. We are getting closer to plotting our data, but we are not quite there yet. First, realize that our Graphicbox has a natural origin defined by it's own coordinate (x = 0, y equals 0). This Graphicbox coordinate exists in the upper left hand corner of the Graphicbox. If we plotted all values of Q(xscaled(), yscaled()) about the natural origin of the Graphicbox, then we would only see the lower right quadrant of the data set. Any point Q whose xscaled() or yscaled() value is less than zero will not appear within the Graphicbox.

To correct for this, we need to plot the data not around the Graphicbox's natural origin, but around the Graphicbox's geometric center. We might think of this operation as shifting all the data points to the geometric center. As indicated above, the center of this Graphicbox is located at

```
To
 enable this data shift, simpl
y add 100 to all of the xscaled() values and
```

```
100 to all of the yscaled() values:
[[code format="vb"]]
For degree = 0 to 360
    xplot(degree) = 100 + xscaled(degree)
    yplot(degree) = 100 + yscaled(degree)
next degree
```

This would be a good time to consider another issue related to plotting data on computer screens. This issue is the direction of the Y axis. When we plot data in school, we usually draw the Y axis pointing upward in our notebooks. In most computer languages, however, y values increase in a downward direction. If we want positive y values to be plotted upward of the Graphicbox's geometric center, then we need to "reverse" the direction of y. To accomplish this, the four lines of code shown above can be replaced by these:

```
for degree = 0 to 360
    xplot(degree) = 100 + xscaled(degree)
    yplot(degree) = 100 - yscaled(degree)
next degree
```

5. So far, we've generated our polar data; we've converted it to cartesian coordinates; we've scaled it up (or down); and we've shifted it to the geometric center of the graphic box. Now, we want to create a routine which actually plots the data.

We learned in geometry that a "point" represents a location in space, but otherwise has no extents. That is, we can talk about "where" a point is, but not "how big" a point is because it does not have "bigness". In computer graphics, we often represent points with pixels. Pixels do have "bigness", though not very much of it. So, instead of just using pixels to plot our data, lets plot some kind of a "marker" at each data point. In PolarPhun, a small circle with a radius of 2 pixels seemed to serve as a pretty good marker. Shown below is source code which plots a small, circular marker at each data point:

```
'We assume that the window is called "main" and that
'the Graphicbox is called "PlotArea"

for degree = 0 to 360
     print #main.PlotArea, "place " + str$(xplot(degree)) + " " _
                              + str$(yplot(degree))
     print #main.PlotArea, "circle 2"
next degree
```

On many occasions, we will want our plot to appear as a relatively smooth curve instead of a sequence of markers. To do that, we can plot lines between successive data points. The code below will accomplish this:

```
for degree = 1 to 360
    xplot1 = xplot(degree - 1)
    yplot1 = yplot(degree - 1)
    xplot2 = xplot(degree)
    yplot2 = yplot(degree)

    print #main.PlotArea, "line " + str$(xplot1) + " " + _
                          str$(yplot1) + " " + _
                          str$(xplot2) + " " + str$(yplot2)
next degree
```

F. HOW ARE POLAR COORDINATES USEFUL IN THE REAL WORLD?

Ok, I give up. How?

Polar coordinate systems probably have many wonderful uses. But the truth is, I personally can't identify very many of 'em. Since I first started plotting polar graphs back in the days of the Commodore 64, I've mainly used them to make interesting shapes.

I know that polar systems are used quite often in aviation. In fact, if you look at the regional maps inside your Microsoft Flight Simulator Pilot's Handbook, you will see compass circles overlaid on top of airports. Each compass circle will have a vector drawn from the center of the circle to due North. The vector is the "reference line" that we discussed in other sections of this article. Starting at due North on each compass circle, the circles are marked every 5 degrees. During flight planning, if a pilot draws a line from the center of the compass circle to his destination airport, that line will cut the circle at a specific degree mark. This is the heading at which the pilot must steer his plane in order to arrive at his destination.

I expect that polar systems are used for navigation at sea, too. Certainly, polar systems are used in land surveying. For other important uses of polar coordinate systems, I must humbly defer to others.

Tom Nally