Drawing with Trigonometry

Trigonometry is one of those subjects that leaves students asking "What's in it for me?" After all, knowing a lot of stuff about ancient Greeks and how to prove theorems isn't much use to a lawyer or a musician. Graphic arts is another story, at least if you want to write your own programs. There are no theorems to prove, but some of the basic ideas are necessary all the time. This paper[1] covers basic concepts and show how to put them to work.

Cartesian coordinates.

Trigonometry lives on graph paper:

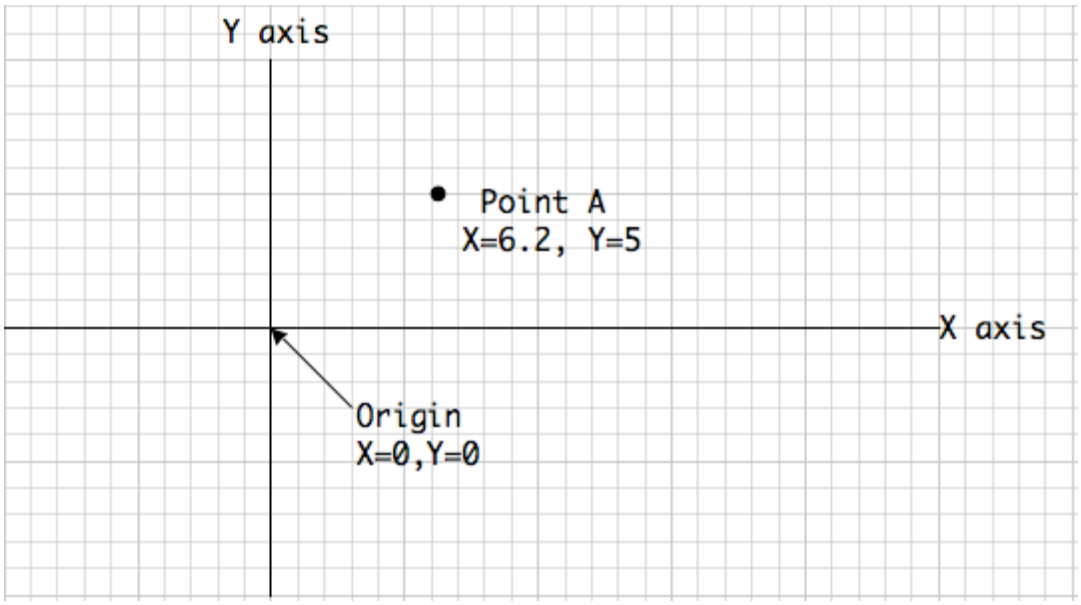


Figure 1. Cartesian space is measured with grid lines.

Figure 1 shows the usual layout. Each line of the grid has a number, and locations on the grid are defined by the intersection of the two nearest grid lines. It's a lot like my saying I live at the corner of Laurel and Cleveland, or I'll meet you at third and Beach. Since this is math, we count to name the lines, and we count starting with 0. The intersection of 0 and 0 is the origin, and lines left of the origin or below it have negative numbers. Also since this is math, we assign letters to indicate horizontal or vertical lines: X for counting to the left and Y for counting up. The horizontal numbered 0 is called the X axis, and the vertical numbered 0 is the Y axis. Any point on the paper can then be defined with an X and Y value. If the point is between lines, it can have fractional values. We always put X before Y, so it makes sense to say point A is at (6.2,5).
Further points about Cartesian graphs:

[1] There is an earlier paper called Notes on trigonometry, which contains the math descriptions of this article, without the Max examples. It may be handy if you are drawing with another program.

- There can be three dimensions-- the Z axis points out of the paper, and is counted with positive numbers in front of the page. More dimensions are possible in math, but hard to draw in our universe.
- You are free to move the origin around, as long as you adjust all of the other numbers to compensate.
- This system is named after Rene Descartes, who lived in the early 17th century. He didn't invent it, but his writings about it are so important that other mathematicians copied his notation.

The application of this to computer graphics should be obvious: it's how we describe the pixels on a screen. The origin is the upper left corner, and pixels are counted down and left. Since Y is positive going up and pixel addresses are positive going down, graphics have to be turned upside down before they are displayed. Some programs do this for you, some do not.

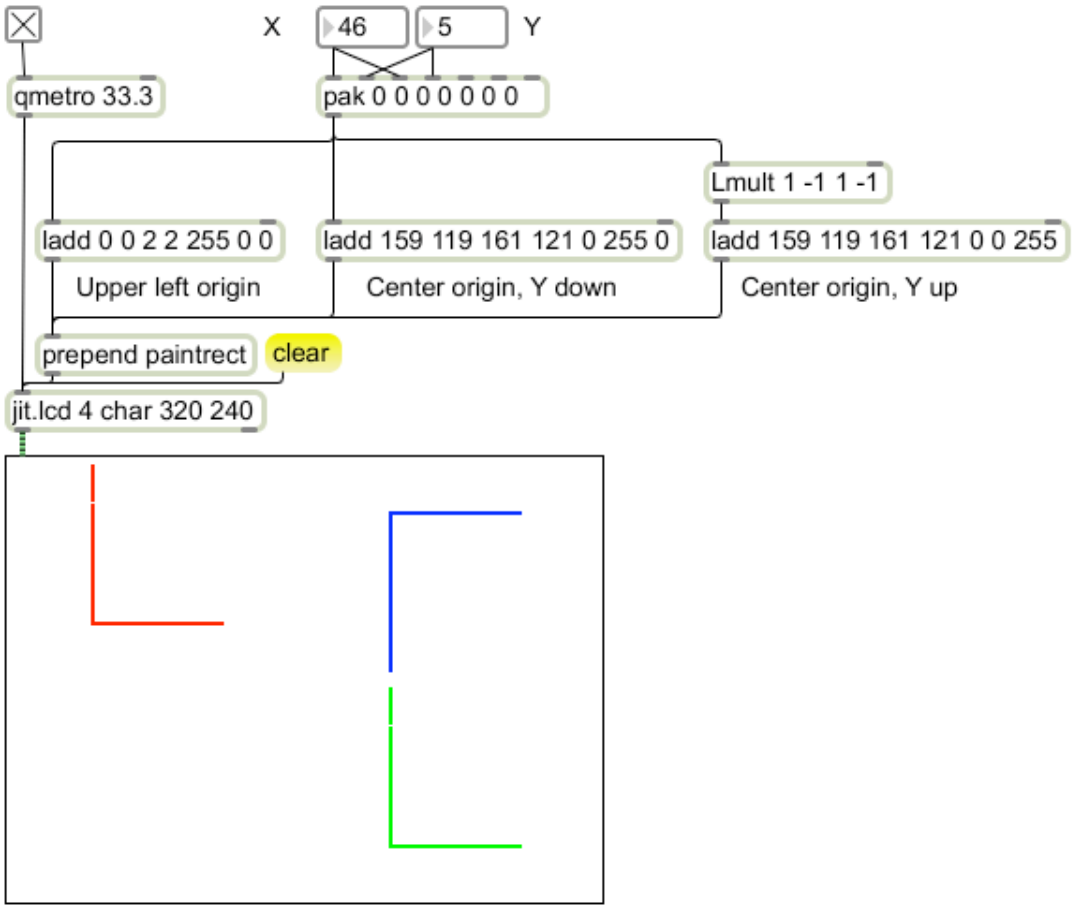


Figure 2. Coordinate systems- Red= upper left origin, Green = center origin, Y down, Blue = center origin, Y up.

Figure 2 compares three common coordinate systems. The same input was used to draw all three shapes. The red shape was plotted using the standard upper left origin, the green

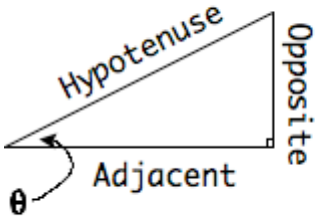
shape was plotted with a center origin with the Y value increasing down the screen, and the blue shape was plotted using a center origin but flipped vertically so Y value increases going up. All three methods use an Ladd[2] object to produce an argument list for paintrect. First a pack object is used to duplicate the X and Y coordinates in a list. (Most jit.lcd drawing commands require a target specification in the form left, top, right, bottom. For the few that only need X and Y, the third and forth list entries are unnecessary.) Following that, Ladd treats each differently:

- For upper left origin, all that is necessary is to add the dimensions of the rectangle.
- For center origin, the location of the center pixel, plus and minus the dimensions of the rectangle are added. This makes the X, Y values the coordinates of the center of the rectangle.
- To draw with Y going up, an Lmult object is required to change the sign of the Y values. This only works with center origin drawings. If you wanted to create a coordinate system with the origin at the lower left, you would subtract the Y value from the window height. (The Lcomp object would do this.)

Each system has its advantages. The standard coordinates are obviously simpler, but most drawing will involve extra objects to place the image. If you wish to change resolution of the screen, you would need to find all of these objects and recalculate their values. With the center origin, only the Ladd object needs to be changed, and that can be done with a list to build a multiple resolution patch. Drawing with Y going up may be conceptually easier in some cases, but there are some operations (such as placement of picts) that require top down thinking, so mixing systems may be more trouble than it is worth. If you are drawing images that will be shown in the jit.gl world, it is worth the trouble.

Trigonometric Functions

Trigonometry originated in surveying, looking at problems like determining the height of a tree without cutting it down. Many of the problems are about angles and distance, and can be solved by the basic functions of sine, cosine or tangent.



$$\text{Sine } \theta = \frac{\text{Opposite}}{\text{Hypotenuse}}$$

$$\text{Cosine } \theta = \frac{\text{Adjacent}}{\text{Hypotenuse}}$$

$$\text{Tangent } \theta = \frac{\text{Opposite}}{\text{Adjacent}}$$

Figure 3.

[2] Ladd and Lmult are part of my Lobjects, available at ftp://arts.ucsc.edu/pub/ems/

As figure 3 shows, these functions are defined via a triangle with a 90° corner, or right triangle. The sides are identified as to whether they are opposite or adjacent to the angle we know. (The side opposite the right angle is the hypotenuse.)

For any angle, the sine of the angle is defined as the ratio of the length of the opposite side to the length of the hypotenuse- since the hypotenuse is always the longest side, sines will be a fraction. The sine of a particular angle is always the same- to find what it is, you can just look it up in a table.[3]

The cosine of an angle is the length of the adjacent side over the hypotenuse. If you look at the drawing, you can see that the cosine of the angle θ is the sine of the unmarked angle, and vice versa.

The tangent is the ratio of the opposite to adjacent sides. Most of the time, we know this ratio and want to find the angle, so we look for the tangent in the table. This process is called finding an arctangent.

There are other functions, such as secant (hypotenuse/adjacent) but they are aren't used as often. The big three are all we really need to memorize.

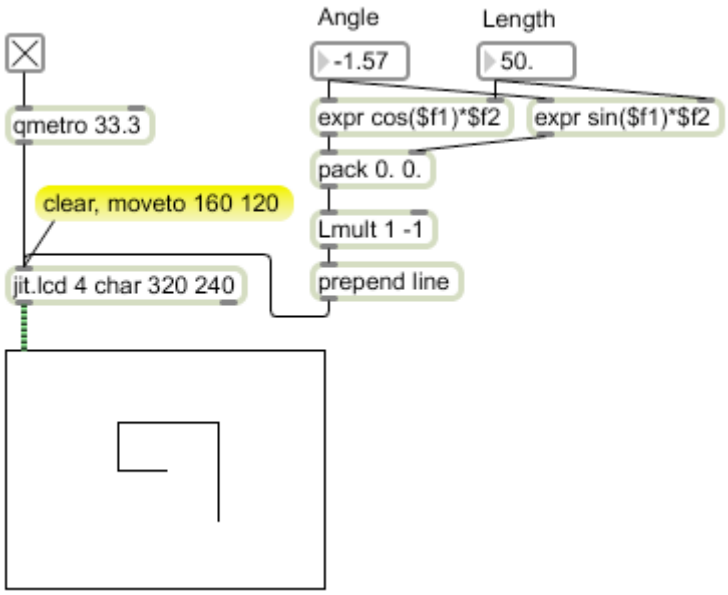


Figure 4.

Figure 4 puts trigonometry to work to draw a line in a desired direction at a desired angle. The horizontal motion is the length times the cosine of the angle, and the vertical motion is the length times the sine of the angle. The length was always 50 pixels, and the angles were 3.14, 1.57, 0, 0, -1.57, and -1.57 radians.

[3] Many mathematicians spent their careers constructing tables of sines.

Plotting Functions

There is an endless variety of beautiful shapes that can be drawn using trig functions. Figure 5 shows a patch for trying things out.

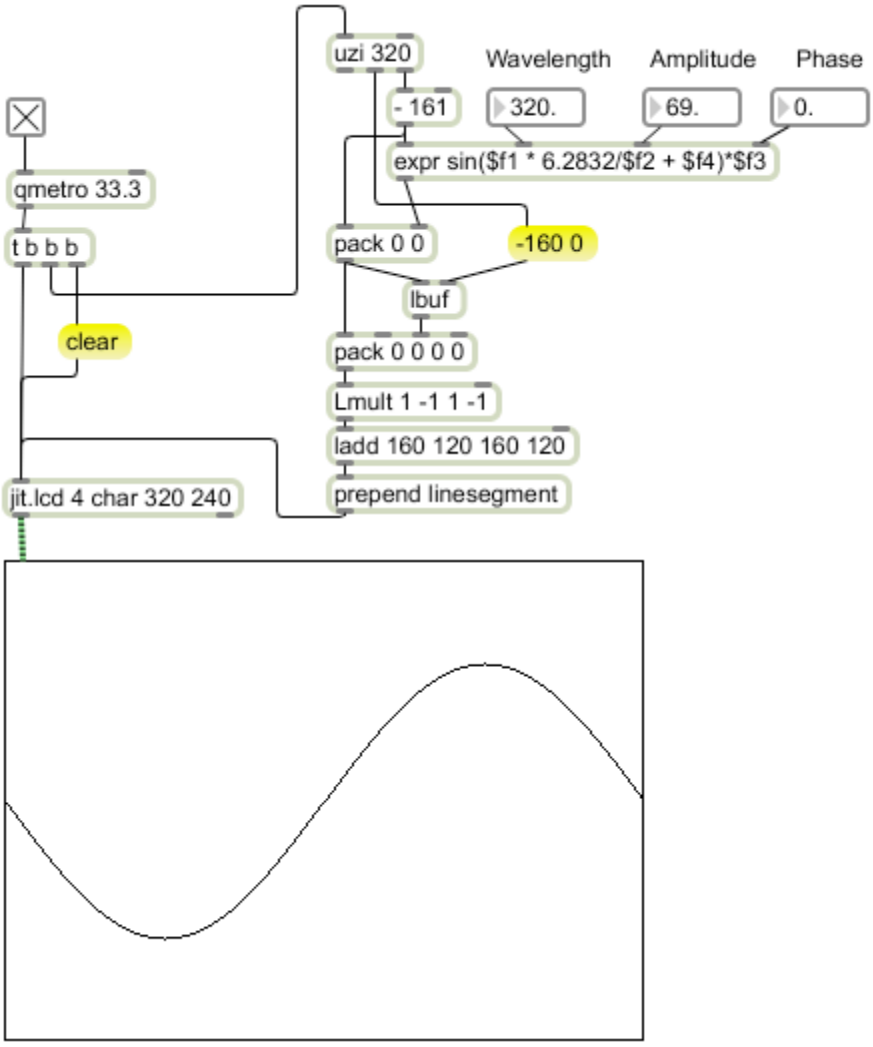


Figure 5.

The basic method of curve drawing is to plot a dot and connect it with a line from the last dot. To do that, we need to keep track of the last coordinates calculated. The Lbuf object (an Lobject) will do this- it holds a list until a new list comes in, whereupon it outputs the list and stores the new one. It's like bucket for lists. The basic strategy here is to calculate X and Y, pack them into a list of two, use the list to fill the first two places in a four part list and the output from Lbuf to finish the list. Linesegment will then draw the line.

This patch uses a center origin, Y up coordinate system. The point corresponding to sin(0) will be in the middle of the screen. This is not exactly like textbook curves, which put the origin at the left edge, but it is handy for graphics. We are plotting Y as a function of X, so an uzi will produce 320 dots across the window. The X value is calculated by

subtracting 161 from the uzi (remember, uzi starts counting with 1). That will place the first dot right at the edge. Note that lbuf is initialized with the list -160 0 and reset to that list after the drawing is complete. This prevents the drawing from beginning with a line from the last point on the right[4]. A complete curve is drawn each frame, which means a clear is sent to jit.lcd before the uzi is triggered.

The math to calculate Y is contained in the expr object. The function sin(theta) returns the sine of the angle theta as a number between -1 and 1. Theta is in radians and needs to increase on each input from the uzi. If we want to show exactly one cycle across the window, theta should step to through 2π in one sweep, so the step size will be 2π/320. Any particular step is at $f1*2π/320. We can replace the 320 with $f2 to give a variable distance between peaks, which I have labeled wavelength.

A range of -1 to 1 doesn't do much good in jit.lcd, so we need to multiply the sine value by the height or amplitude we want to draw. This is provided in the expr by $f3.

Phase is a specification of where the curve starts, or in figure 5 what part of the waveform is at the center (where X = 0). This simply added to the terms within the sin() function parentheses as $f4. The complete expression is then

sin($f1*6.28/$f2+$f4)*$f3

Phase is in radians, so a phase of 3.14 will turn the curve over. Phase can easily be animated to produce waves that sweep across the window.

Polar Coordinates

There is a second way to look at location. In polar notation, a point is defined by a radius and angle, as if it were on a circle:

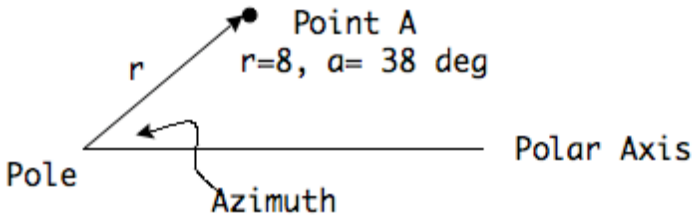


Figure 6.

The center of the circle is called the pole, and is equivalent to the origin in the Cartesian system. (Everything is measured from here). A horizontal line from the pole to the right is called the polar axis. To locate a point, draw a line from the pole. The length of this

[4] There is a vertical line at the beginning of the drawing, but that is concealed by the jit.window border.

line is the radius r, and the angle the line makes with the polar axis is the azimuth, a[5]. Angles may be measured in degrees or radians. These are both defined in terms of how many are in a complete circle: 360 degrees or 2π radians.[6]

This may seem like an awkward way to describe locations, but polar notation makes some operations very easy. For instance, rotation around a point just requires a change to the angle. We usually work in Cartesian coordinates, but occasionally convert to polar for some operation, and immediately convert back. To see how to convert, we must revisit the bane of all trig students, the Trigonometric Functions.

Cartesian to Polar and Back

Figure 7 shows how Cartesian and Polar coordinates are related.

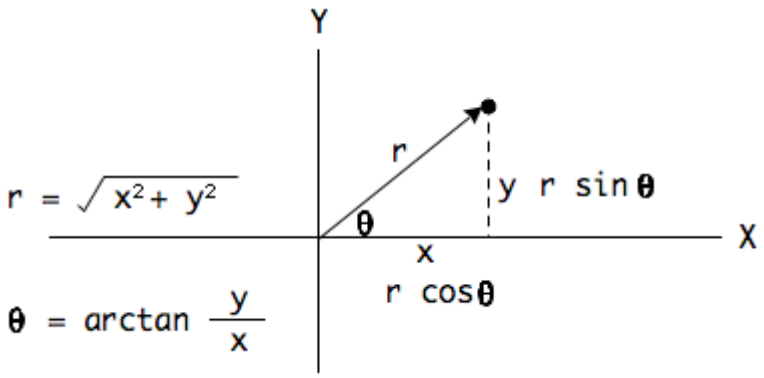


Figure 7.

To go from Cartesian to Polar, the radius is the square root of the sum of the squares of the other two sides. The angle is the arctangent of the ratio of opposite to adjacent. In C or Java code that would be:

```
r = sqrt(x*x + y*y);
theta = atan(y/x) or arctan2(y,x)[7];
```

Of course in Max the whole thing is calculated in cartopol. The angle is reported in radians. To go back after the process:

```
y = r*sin(theta);
x = r*cos(theta);
```

[5] Mathematicians generally use Greek letters for azimuth and angles in general. The most common is θ (theta). My graphic program doesn't do Greek, and Greek in Word doesn't print reliably, so I use roman letters.

[6] I thought radians were awful until I started doing math with them. Any formula that includes both π and angles in radians is easy to work out because the π terms will cancel out.

[7] On computers arctan(y/x) will generate an error if x=0. Also arctan() does not tell which way the angle goes. Artctan2 deals with all of this. (Notice the order of arguments)

All of which is in the poltocar object. One other interesting use of these formulas is that if you plot them with a steady radius and a theta that hits most of the numbers between 0 and 2π, you will get a circle. Figure 8 shows how to generate a set of angles and use them to draw a shape.

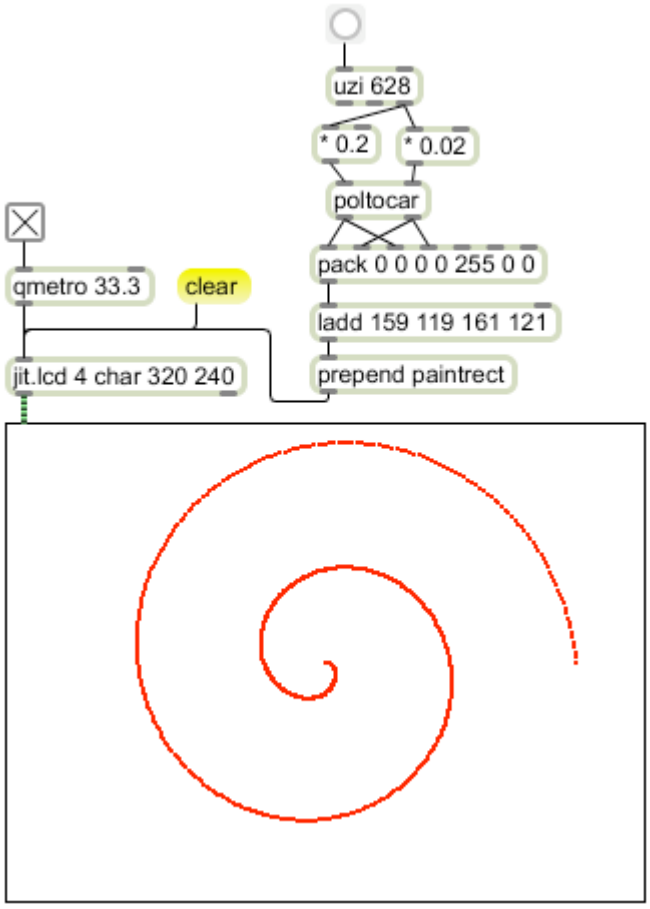


Figure 8.

The driving engine here is uzi, firing 628 values. These are multiplied by 0.02 to create angles in steps of 0.02 radians-- that will go twice around a circle. For each angle, there must be a radius. In this case the same stream of numbers is multiplied by 0.2, giving the relationship

R= 10 * theta

Any time the radius is a function of the angle, you will get some sort of spiral. Calculating angles from uzi works well when drawing static forms, although we have to remember that uzi starts counting from 1 instead of 0. When I need circular motion, I use Lcount, set up like figure 9.

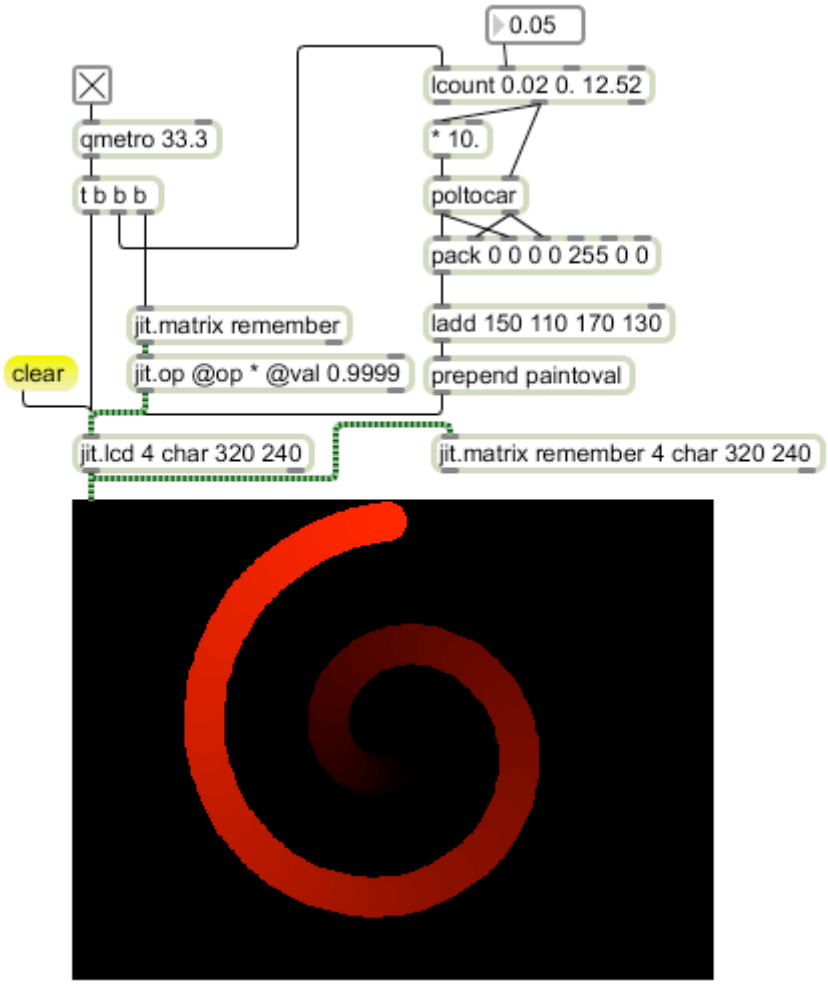


Figure 9.

The spot in figure 9 follows the same path drawn in figure 8. (I'm using the feedback trick to help visualize this[8].) Lcount will produce any increment desired, so no processing is needed on the angle side. Of course the end count needs to be changed to get two rotations-- thus the 12.52 for the argument. That points up the essential difference between counter and Lcount. With counter, you specify the number of steps and process the output to get you the right step size and reach the target. With Lcount you specify the target and step size. It's easy to change the rate of motion with Lcount- larger increments mean more rapid motion.

[8] See my basic drawing tutorial.

Modulation in Drawings

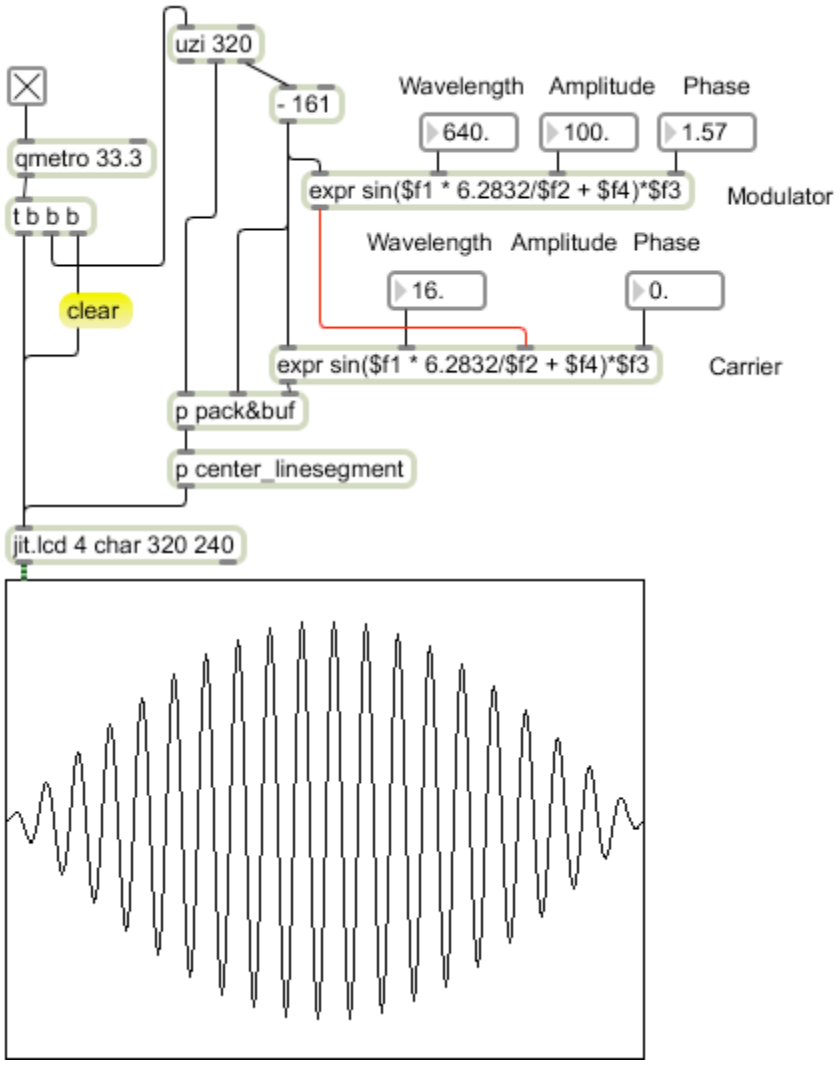


Figure 10.

When we play with audio, we often modulate one waveform with another. If we use the output of one sine wave to change the amplitude or frequency of another, we get some wild and wonderful sounds. These same operations can produce some interesting shapes as well. Figure 10 shows how to use a sine wave to modulate the amplitude of another sine wave. This is based on figure 5. (Some parts have been encapsulated to save space.) All that is necessary is the replacement of the constant amplitude value with one that is recalculated on each step. In figure 10, the recalculation comes from a second expr with the same sine function. (The connection is shown with a red patch cord.) In keeping with audio practice, the function that is drawn is labeled carrier, and the function that modifies the carrier is the modulator. If we plotted the modulator, we would see that only half of a wave is in the frame. This gives the overall shape we see to the carrier. Figure 11 illustrates this with the modulator in red and the result in blue. Figure 11 also shows modulator wavelengths of 320 and 80 pixels on the same carrier.

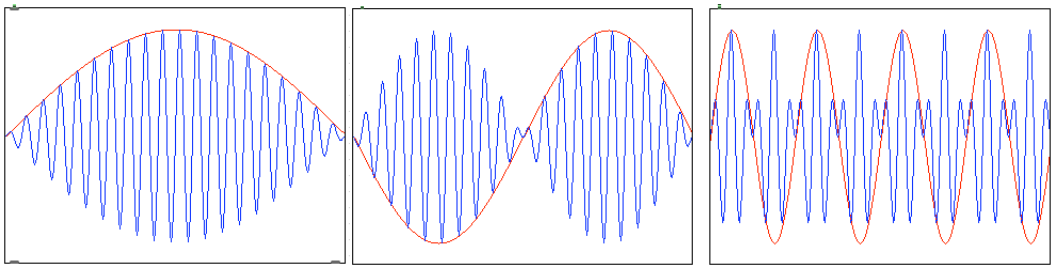


Figure 11.

Drawing Between Waveforms

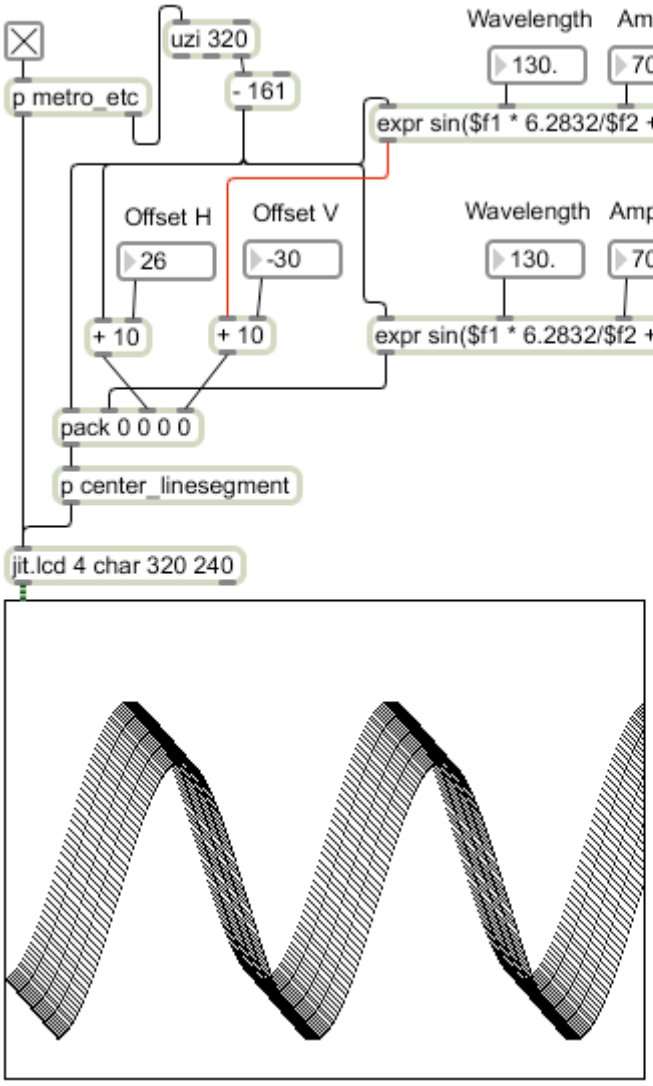


Figure 12.

Figure 12 shows another way to combine two waveforms. This uses the same linesegment command, but each end of a line segment comes from a different sine function. If the functions are the same wavelength and amplitude, changing the offset of one function from the origin produces images similar to the one shown. Figure 13 shows

some variations made by changing amplitude, wavelength and phase. (A color effect has been added to clarify the lines.)

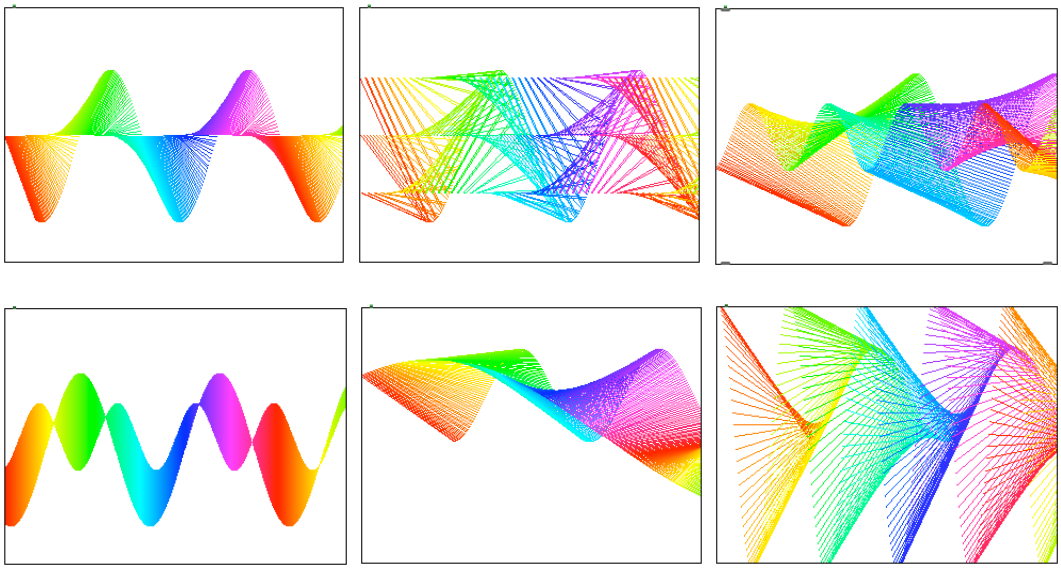


Figure 13

Lissajous Patterns

As I mentioned before, plotting x=cos(theta) and y = sin(theta) will result in circles. If you use different but related angles in the sine and cosine terms you can generate many complex forms based on Lissajous patterns (See my tutorial on Lissajous art). Figure 14 shows some of the basic patterns.

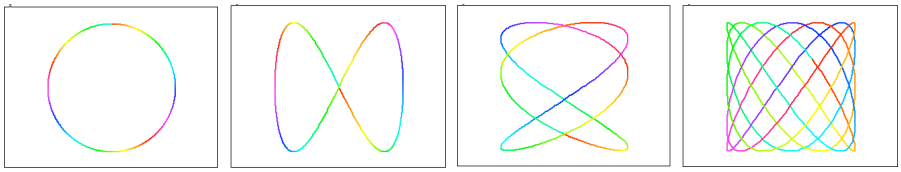


Figure 14. Lissajous patterns

Figure 15 shows a mechanism to generate Lissajous figures. It calculates coordinates from sine and cosine of a series of angles. The Lcount object provides the angles and the exprs contain the functions. The other objects allow the frequency ratios and phase of the two functions to vary as well as provide overall control of the amplitude.

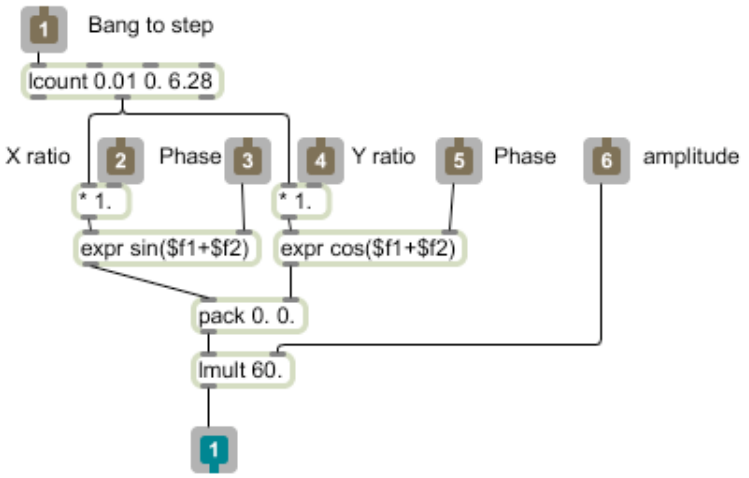


Figure 15.

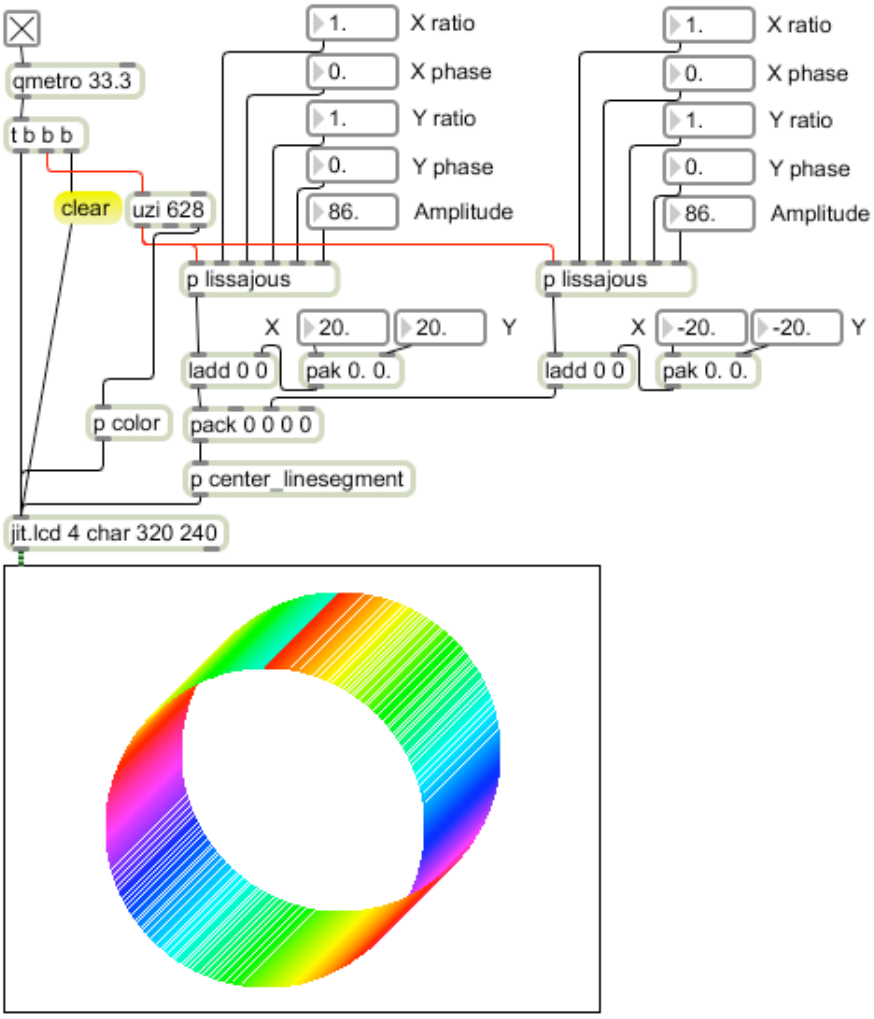


Figure 16.

Figure 16 uses a Lissajous figure for each end of a set of line segments. Independent settings for all four functions involved, plus the ability to offset each Lissajous figure

provide a rich assortment of images, some of which are shown in figure 17. Animation of any parameter in this structure will provide complex mutations of the shapes.

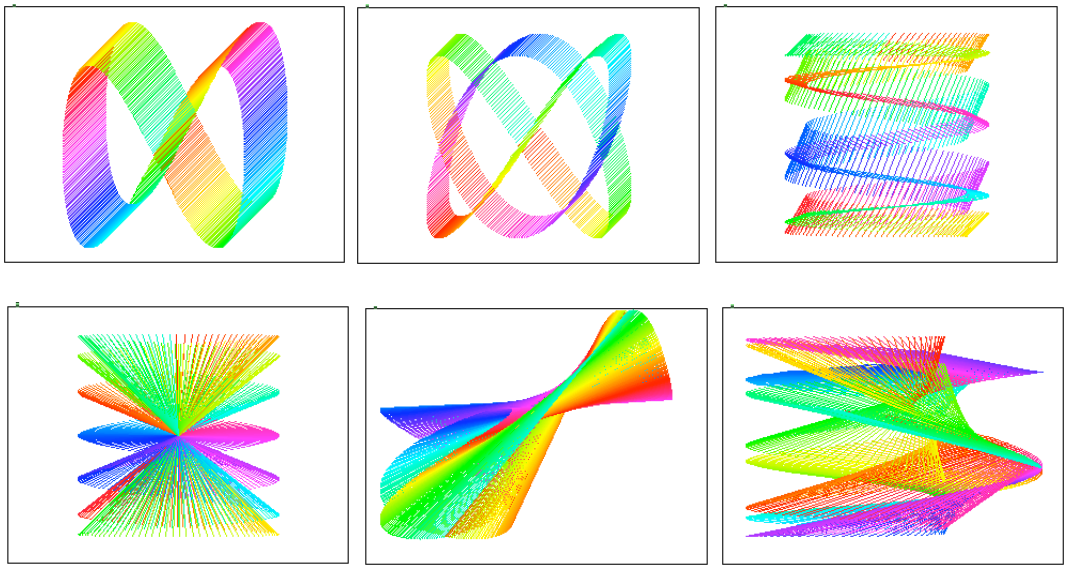


Figure 17.

Polar Flowers

If you use polar coordinaets you can make flowers by manipulating the radius. Figure 18 applies that technique. There are two Lcount objects, each generating a series of angles for each frame. Note that there is no synchronization of the count to the frame, so (depending on the increment) a frame may contain only a part of a circle or go around more than once. The right hand Lcount generates a set of angles for the poltocar object, while the sine of the left hand angle is added to the radius. This is a modulation of the radius. Controls are provided to set the base radius and amount of modulation. Changing the increments of the Lcount objects is tantamount to changing the frequency ratios of the counters and will produce images such as those in figure 19.

Most of the images consist of simple loops. The number of loops depends on the ratio of the increments in the two lcounts. If the radius equals the modulation amplitude, a ratio of 10 to 1 (0.4 to 0.04) will produce a 10 loop flower. If the base radius is less than the modulation amount, a second set of loops will appear, generated by the negative swing of the sine function. If the ratio is even, the negative loops will appear between the main loops. If the ratio is odd, the loops will be superimposed.

If the angle increment is not an exact division of 6.28, the image will rotate.

Large steps on the right lcount object will produce geometric figures such as the square shown in figure 19. These will pulse and flicker as the increment of the modulation is unlikely to be a precise ratio with the angle increment. If the modulation increment is made larger than the angle increment, the image will become a star shape.

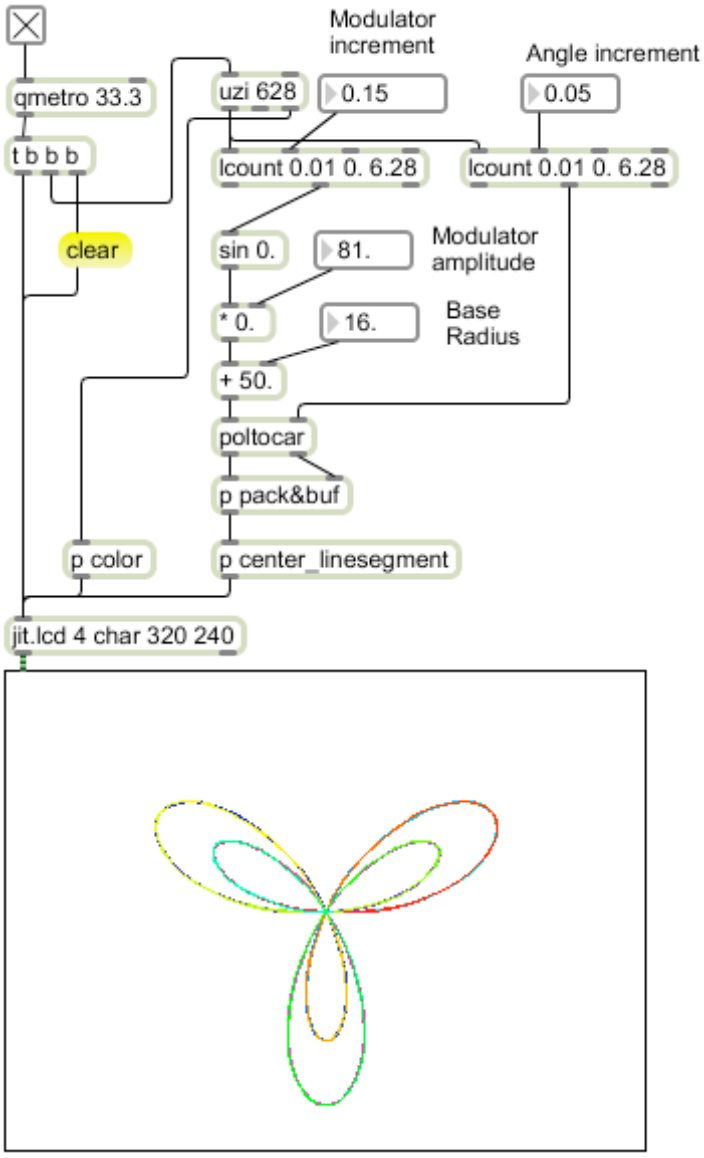

Modulator increment
Angle increment
qmetro 33.3
uzi 628
0.15
0.05
t b b b
lcount 0.01 0. 6.28
lcount 0.01 0. 6.28
clear
sin 0.
81.
Modulator amplitude
* 0.
16.
Base Radius
+ 50.
poltocar
p pack&buf
p color
p center_linesegment
jit.lcd 4 char 320 240

Figure 18.

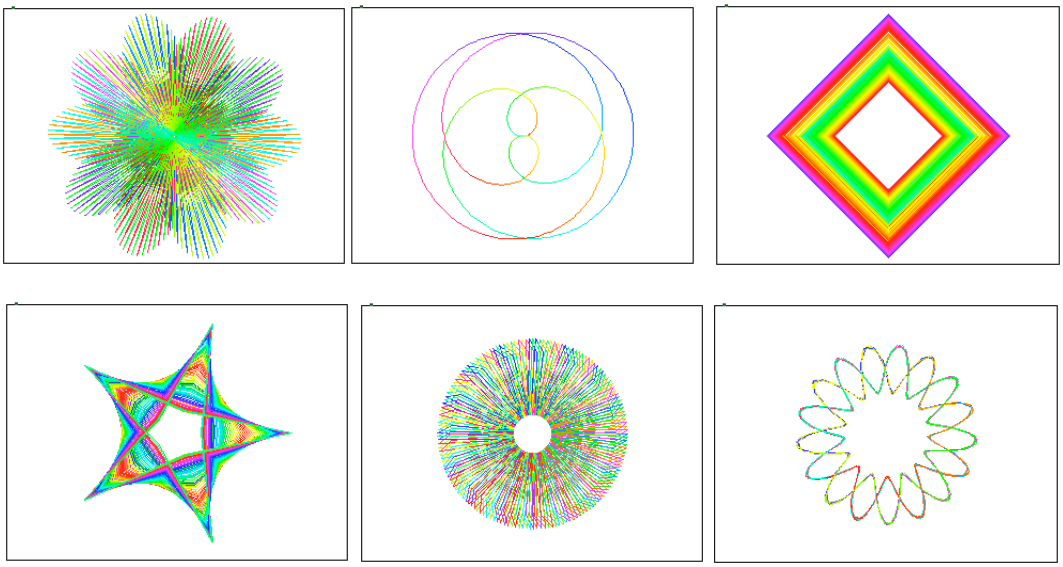


Figure 19.

The application of the ends-of-line-segment trick to this process can add dimension and interesting opportunities for animation. Figure 20 shows how.

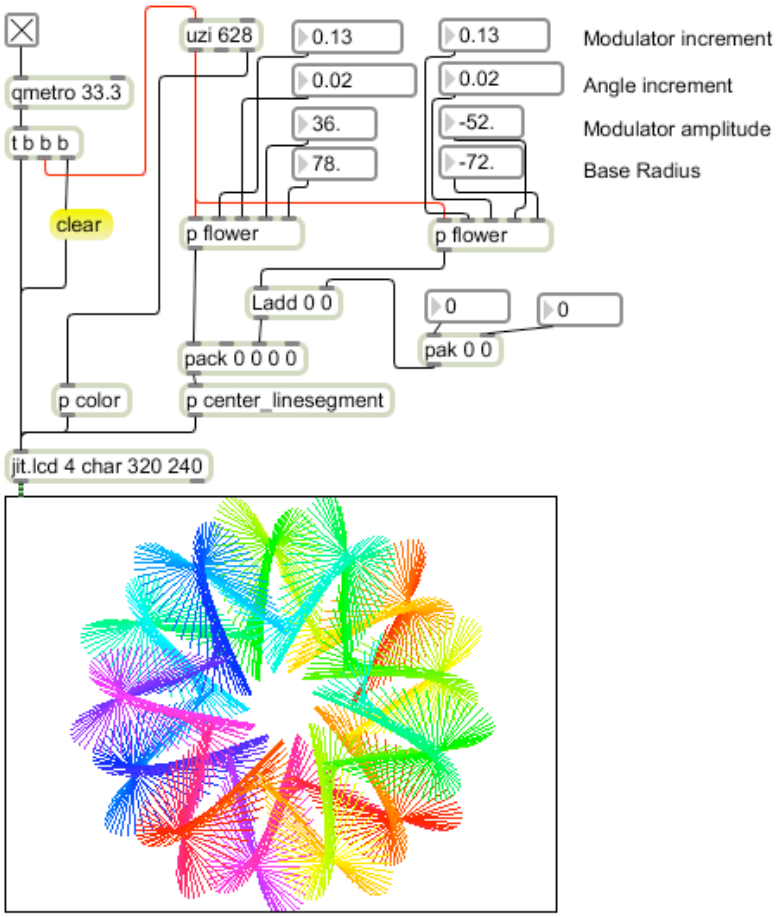


Figure 20.

The essential mechanism of figure 18 has been encapsulated in the subpatcher called flower (including a pack to produce a list of X Y output), A second subpatch sets the end of a line segment as before. Thus the lines are drawn from a point on one flower to a point on another.

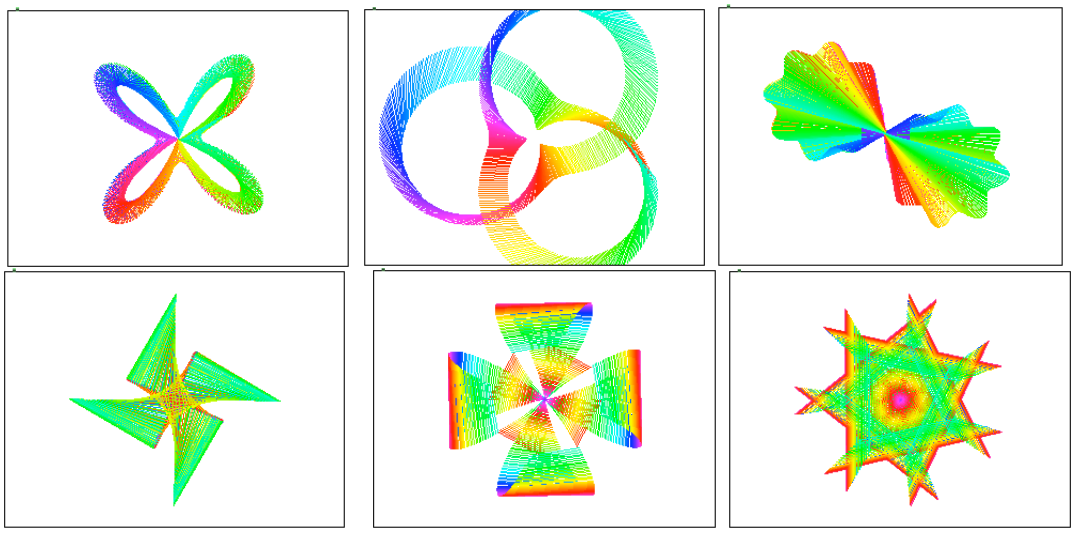


Figure 21.

Since the two Lcount objects are not synchronized, the lines will often cross, making complex spider webs. It gets interesting when the angle increments do not quite match, say 0.02 and 0.02008. then the figures will twist and turn into elaborate knots. Figure 22 shows snaps from one cycle.

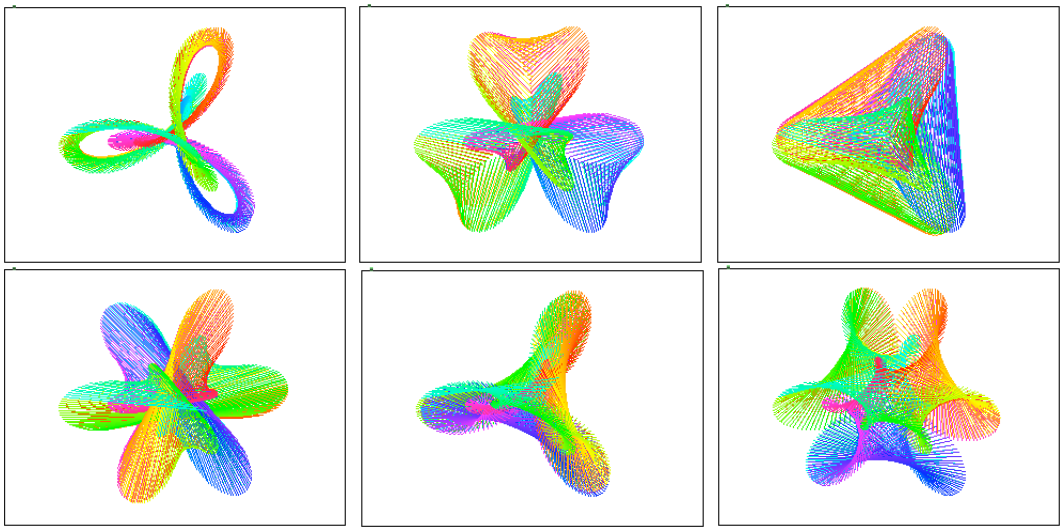


Figure 22.

Arabesques

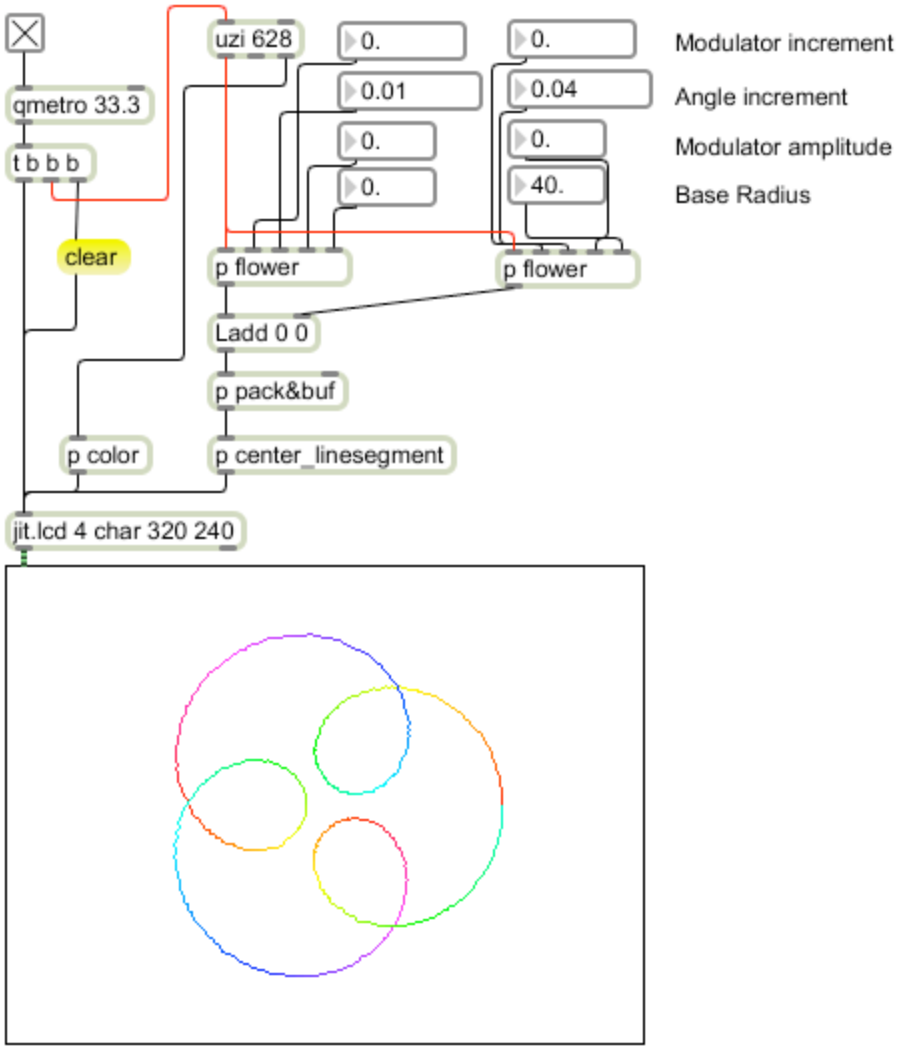


Figure 23.

In figure 23, the coordinates from the two flower subpatchers are added. If modulation is turned off, the result is interlinked circles, with the number of loops determined by the ratio of the increments minus 1. As before, if the ratio is slightly off an integral value, the image will writhe and spin. Adding modulation to either side results in lopsided forms that rotate and turn inside out. Figure 24 only hints at the possibilities.

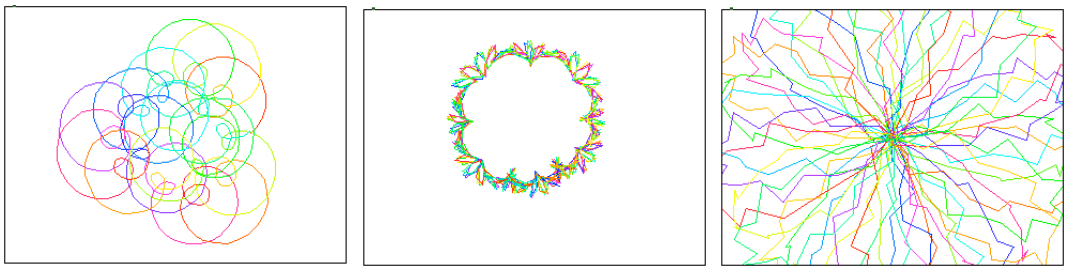


Figure 24.