## Meet jit.repos

Repos stands for reposition, an efficient way to mangle images. We can use it to turn pictures inside out, or build fun house mirrors, strange lenses, and kaleidoscopes. The principle behind jit.repos is simple. It holds a control matrix with 2 planes and the dimensions of the image to process. The values in the control matrix are the source for each pixel in the final output. For example, if the control matrix contains [0 0] at position [160 120], pixel 160 120 in the output will contain the colors found at [0 0] of the input. This can produce an endless variety of strange images.

The secret to using jit.repos is in making control matrices. This can be a tedious process, so it is best to build them ahead of time and save them to disk. Once they are made up, it is easy to use jit.xfade and jit.matrixset to move them around and animate the process.

The first control matrix to make does nothing at all. Each cell contains its own cell position, so jit.repos will just pass the image through. (I'll call it the pass.jxf matrix) The method used to derive it is unnecessarily complex, but we only have to do it once, and this patch will become our control matrix factory.

We'll use a simple test patch to check the results:

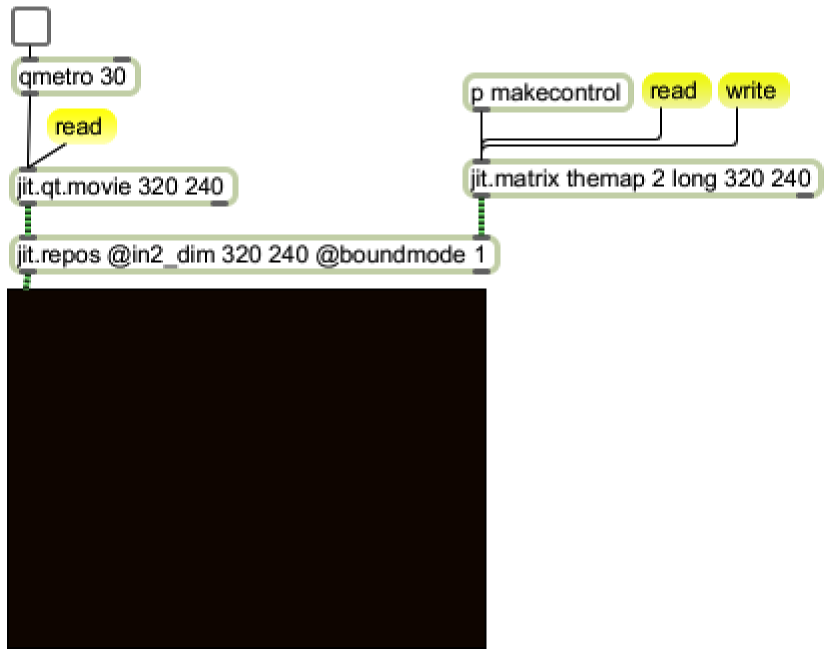

qmetro 30
read
jit.qt.movie 320 240
p makecontrol
read
write
jit.matrix themap 2 long 320 240
jit.repos @in2_dim 320 240 @boundmode 1

Figure 1.
Even with a movie loaded in and the metro running, there is no output until a control matrix is applied to the right inlet. The makecontrol subpatcher has the basic construction driver.

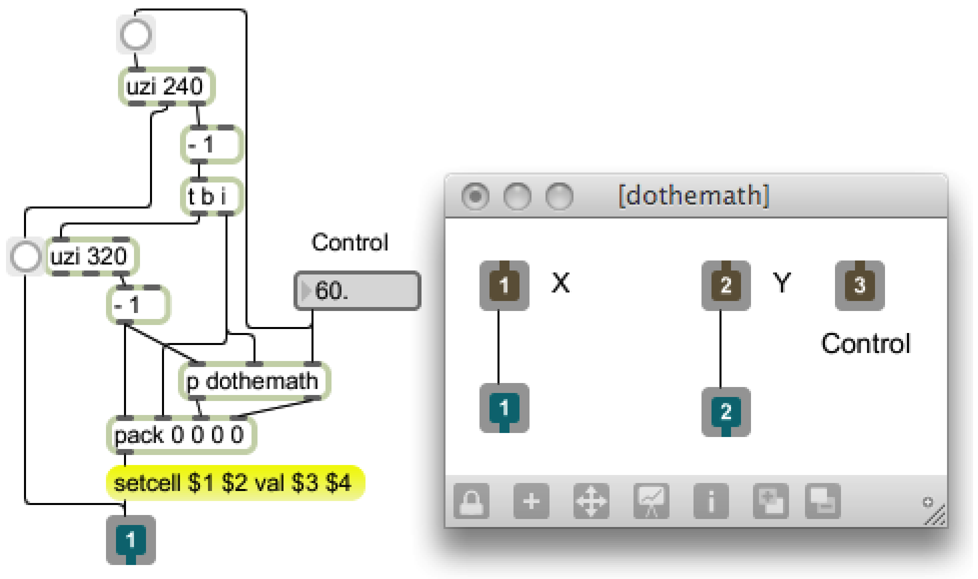


Figure 2 makecontrol
This simply stuffs a setcell command with the same numbers for address and value. The result in this case is a perfectly normal image. (It's not useless, though. With this control matrix, the offset_x and offset_y attributes of repose will slide the image around.)

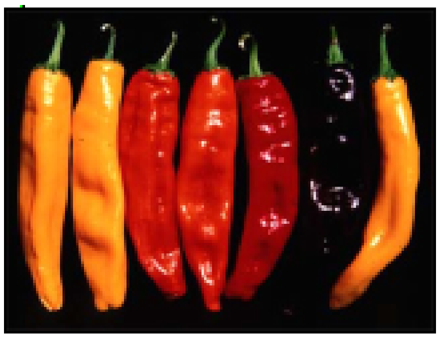
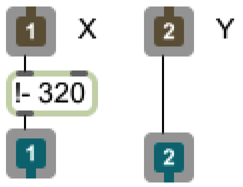


Figure 3.
Save the matrix with the write command. Now a simple process creates a different control matrix:



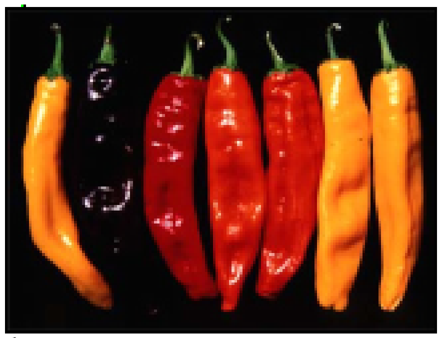


Figure 4. The math: the result.

Plane 0 of the control matrix now has the numbers reversed across each row. The result is a left to right swap. The same process on Y will produce controls for the results in figure 5.

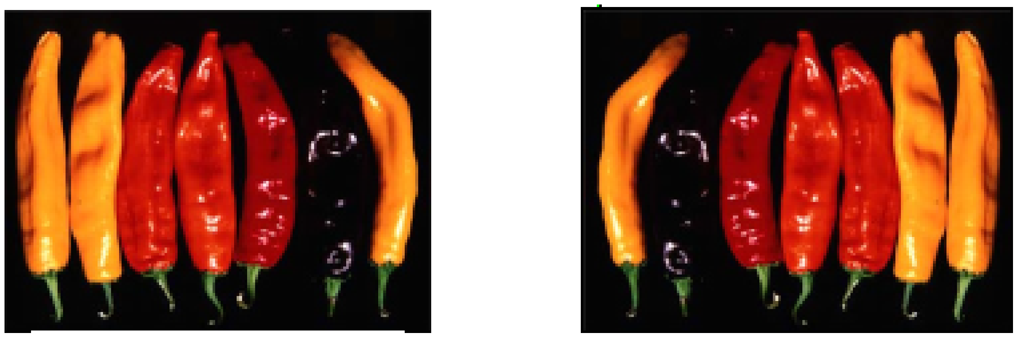


Figure 5

## *Partitioning*

Figure 6 illustrates another simple process. This is all available in jit.split, but it's a bit more efficient here, and the process of making the controls is a stepping stone to some unique things. If we use the remainder 80 across the top and remainder 60 down the side, the upper corner of the image is repeated in a 4 x 4 grid. We could focus on other parts of the image by adding an offset after the remainder. This would point to the upper left pixel of the section we want.

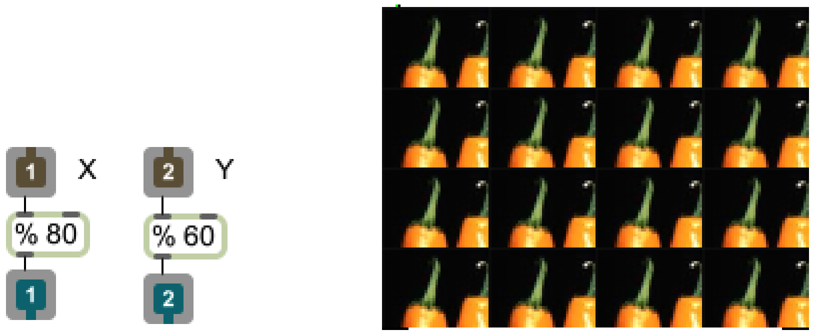


Figure 6

In figure 7, the entire image is squeezed into a grid at a smaller scale. The small images are made by stepping through the whole image, taking every 4th pixel.

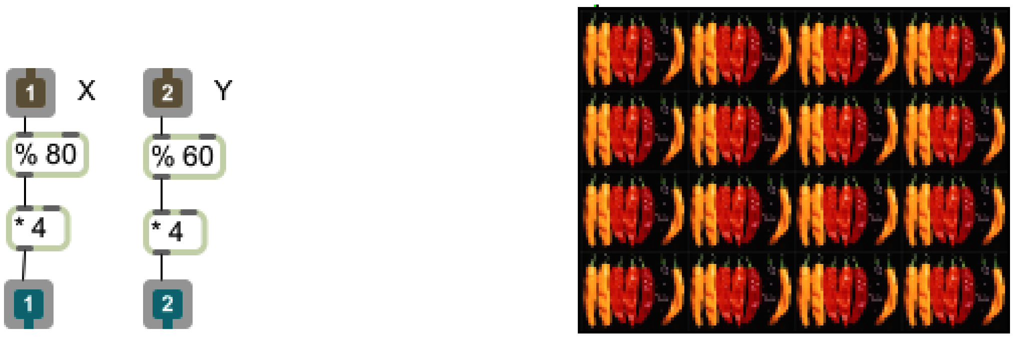


Figure 7.

Figure 8 shows how to modify the process for different areas of the image. A gate chooses one of two possible processes depending on the value of X. The int box is necessary to insure that the Y value is correctly processed according to the current conditions. In the simple patches, Y is only processed once per each 320 X values.

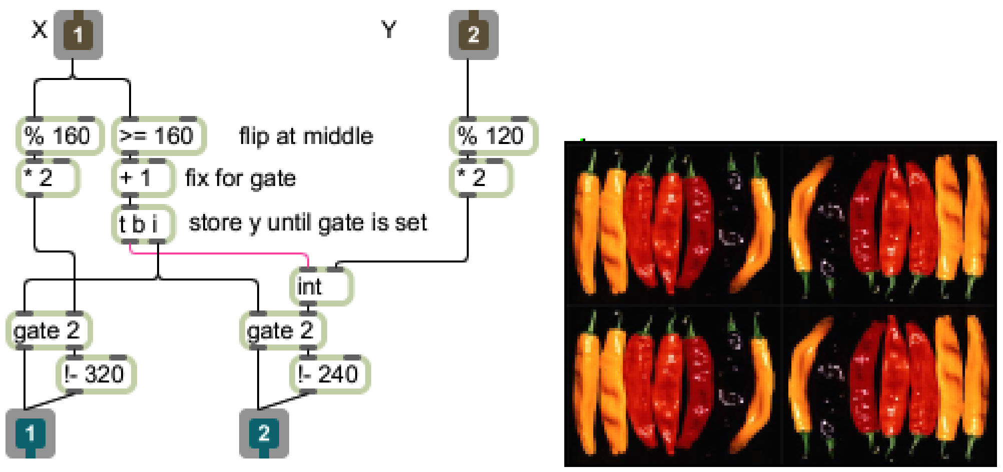


Figure 8

Figure 9 goes a bit further. The split point is taken from Y, so it moves diagonally across the image. Y is scaled by 1.33 to conform to the 4:3 aspect of the movie. You should experiment with changing the sense of the comparison and other ratios for the y scaling.

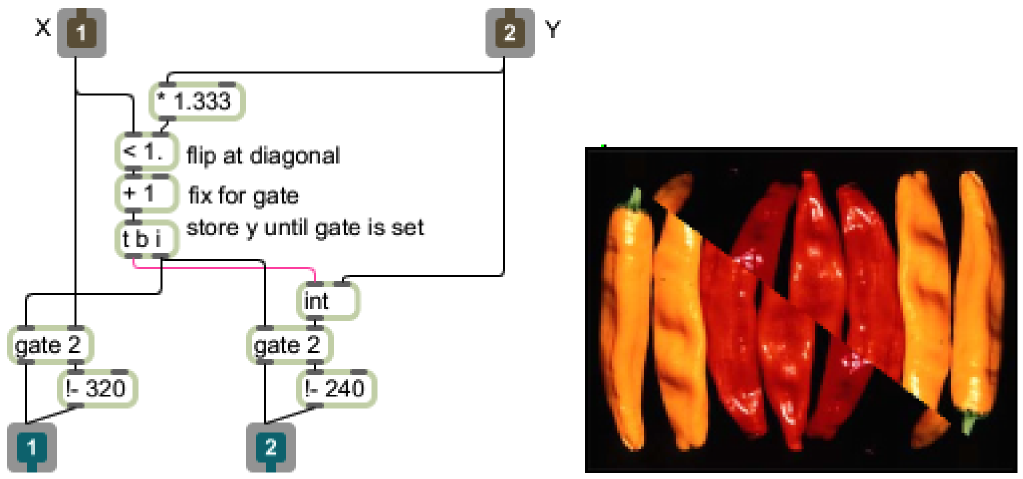
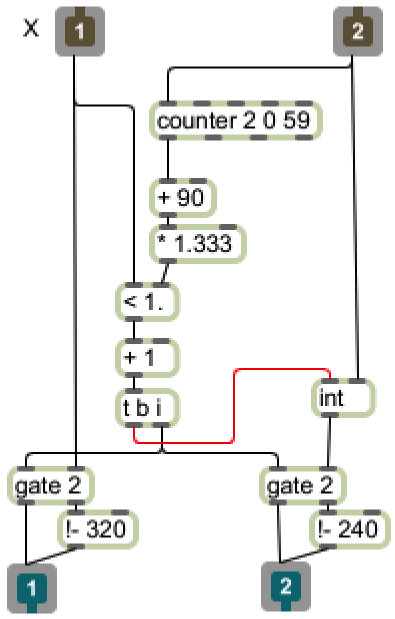


Figure 9.

In figure 10, a counter gives an even more complex seam.



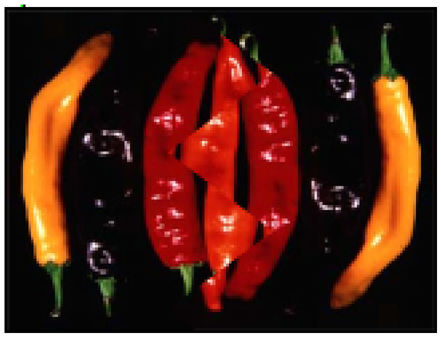


Figure 10.

## *Rotations*

To do rotations, we use our old friend cartopol, as in figure 11.

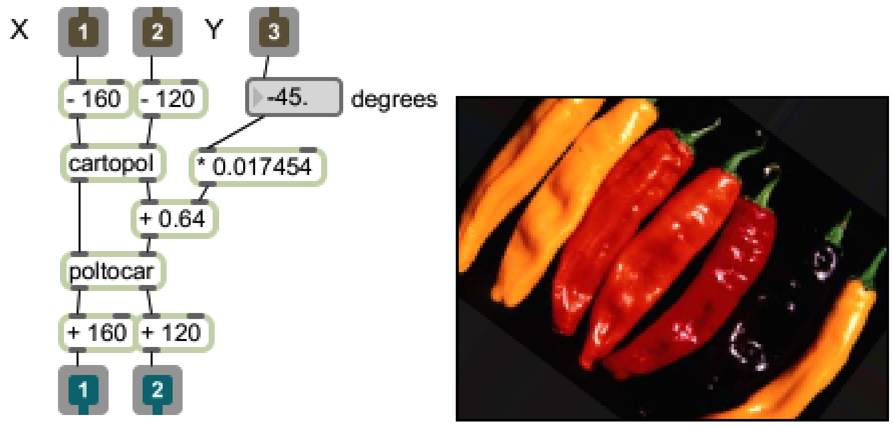


Figure 11.

First, translate the coordinates to the center (or other anchor of your choice) by subtracting the center position from the incoming X and Y. Then cartopol converts the Cartesian coordinates to polar. Add an angle and convert back with poltocar. Finally, move the origin back to the corner. Notice that cartopol considers a positive angle to be a counter-clockwise rotation. This example uses the control inlet, and converts the incoming value (in degrees) to radians.

The rotation need not be constant. In figure 12, it is dependent on Y as well as a control angle.

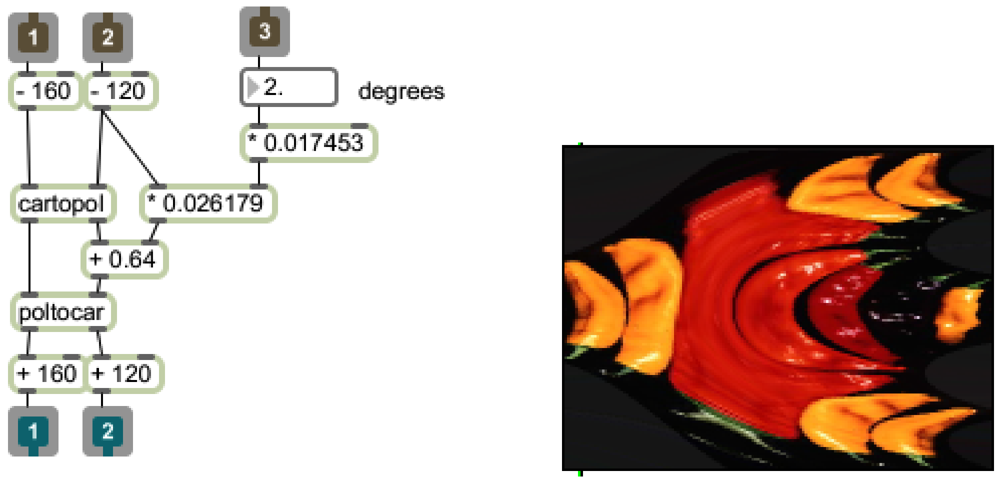


Figure 12.

Some other rotation ratios are shown in figure 13

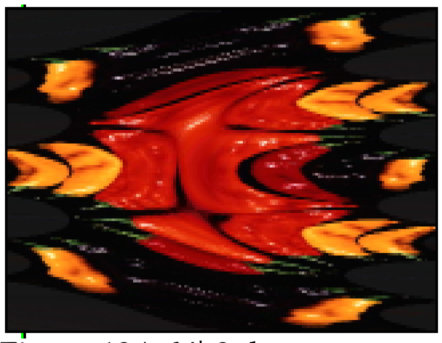
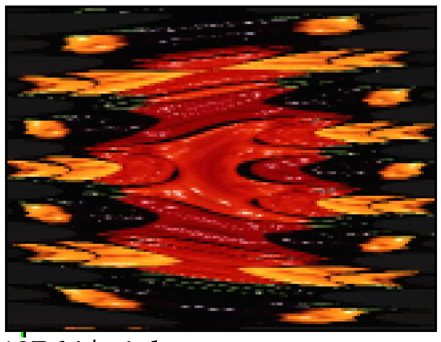
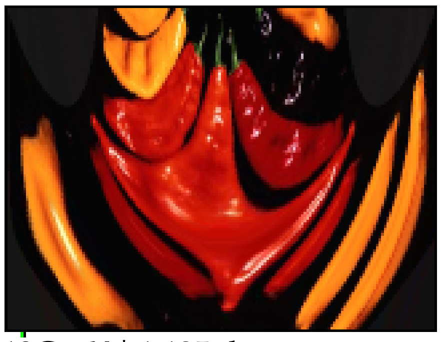


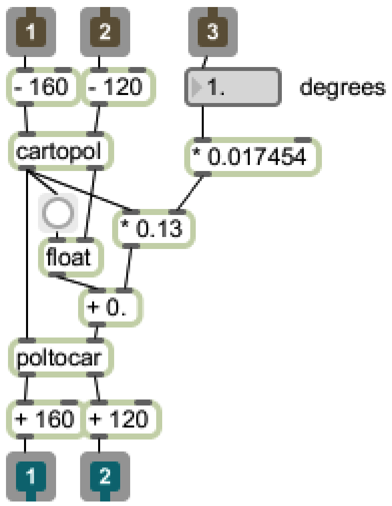
Figure 13A  Y* 3 degrees


13B Y *  6 degrees


13C    X * 1.125 degrees

13D  (X+Y) * 0.28 degrees

Making the rotation dependent on the radius after the cartopol will generate spirals.



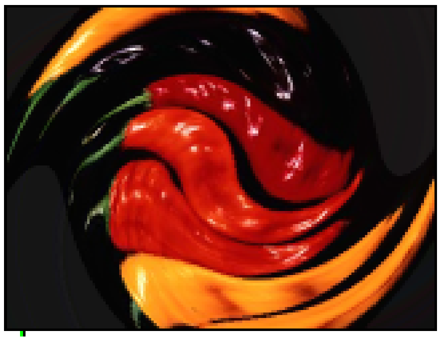


Figure 14.

The angle controls the number of times around. Larger values give tighter spirals.

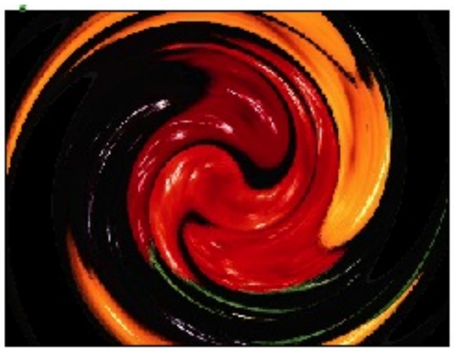
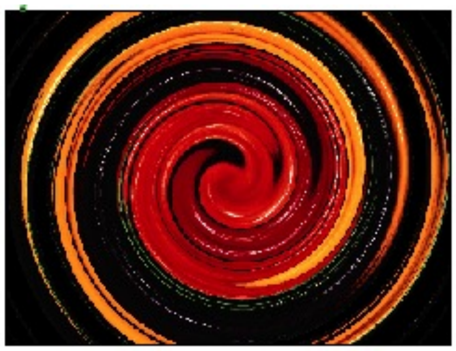
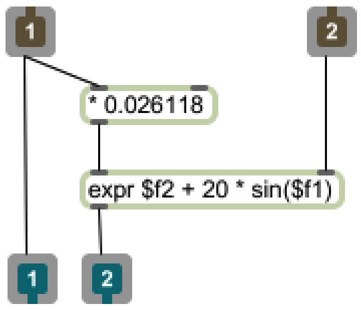


Figure 15A control 3 degrees


15B control 6 degrees

## *Trigonometric transformations*

Simple trig functions will give waves in the image, not unlike amplitude modulation of an audio wave.



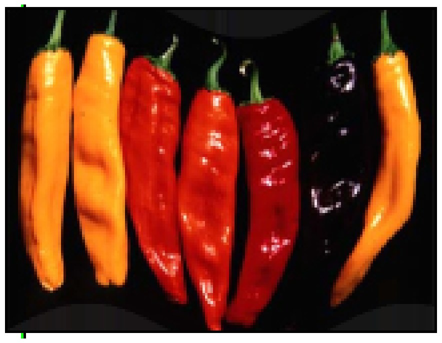


Figure 16.

In figure 16, the sine of the X value is used to modulate the Y. The multiplier is based on pi/ 320. Other multiplier give similar results. Figure 17A shows a high frequency ripple, while 17B shows the effect of modulating both X and Y. In 17C the amplitude of the modulation is increased to run from -100 to 100. 17D suggests the effects of tangent functions.

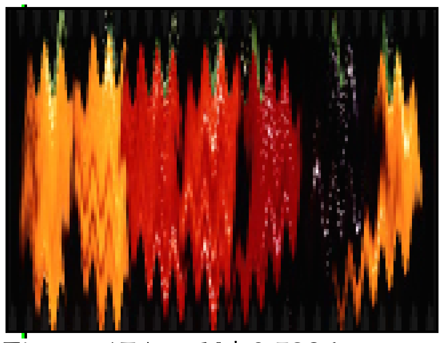
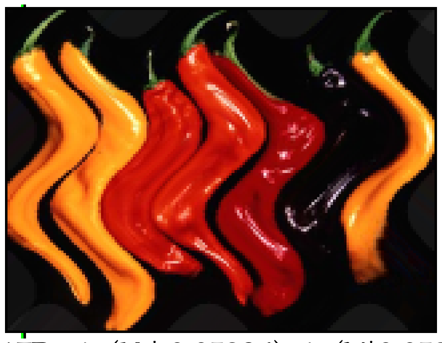
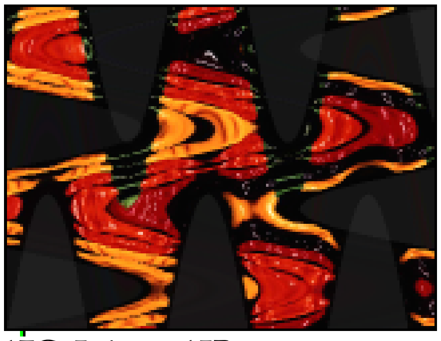
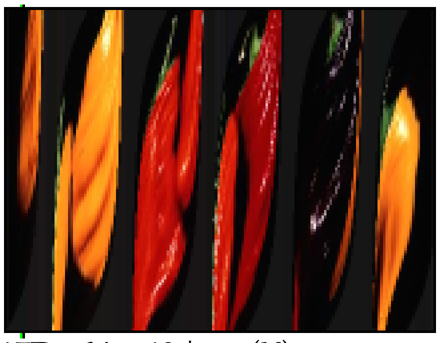


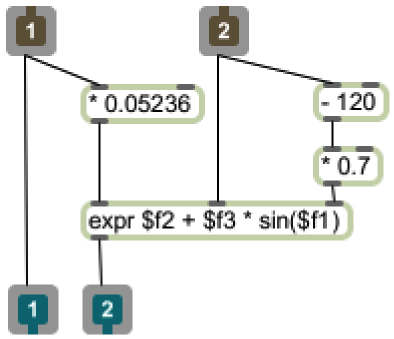
Figure 17A X * 0.5236


17B sin(X * 0.05236) sin(Y*0.05236)


17C  5 times 15B


17D    Y = 40 * tan(X)

The distortion can be bilateral if you use the Y value to scale the sin function.



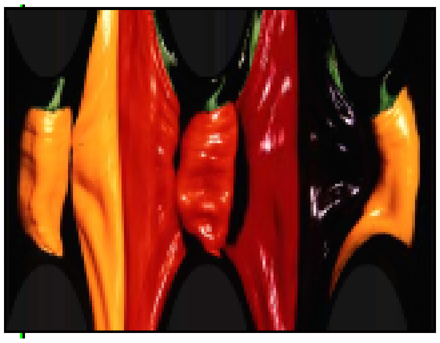
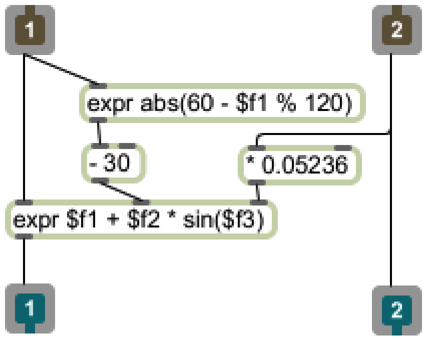


Figure 18

Or if you use X.



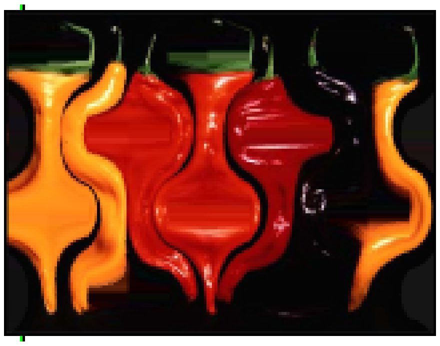


Figure 19
The expr object in figure 19 produces a triangle function, a useful trick for getting smooth transitions in the middle of a control matrix.

## *Reflections*

We can fake a simple reflection by combining rotation and partitioning:

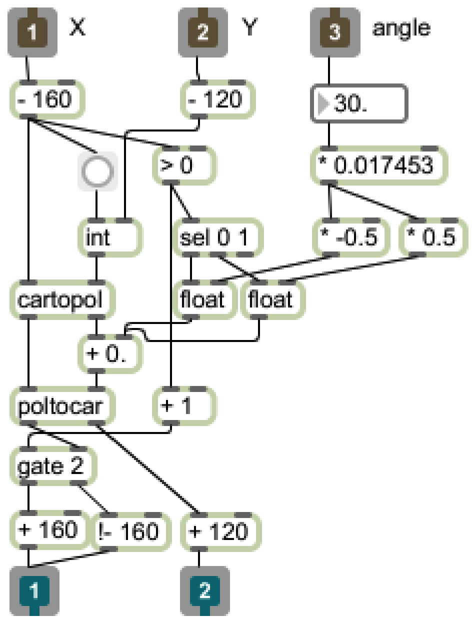
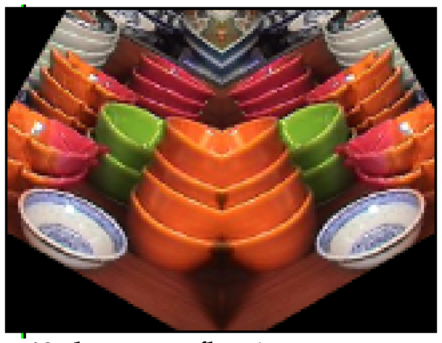
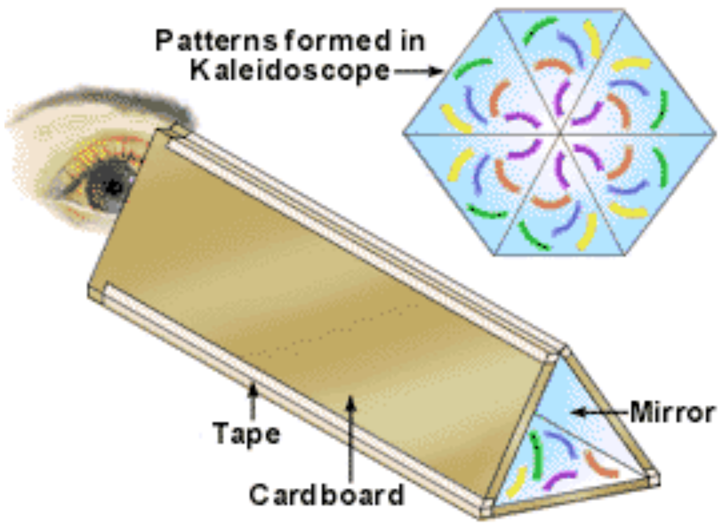


figure 20.



a 60 degree reflection

## *Repos and the Kaleidoscope*



www.arborsci.com

Kaleidoscopes are some of my favorite toys. The traditional design has a pair of mirrors mounted at an angle in a tube so that as you look through the tube you see a slice of something at the other end, plus its reflection as well as reflections of the reflections all the way around a circle. To do this with repos we build a control matrix that divides into segments, and for each segment get either the image rotated or the reflection rotated.

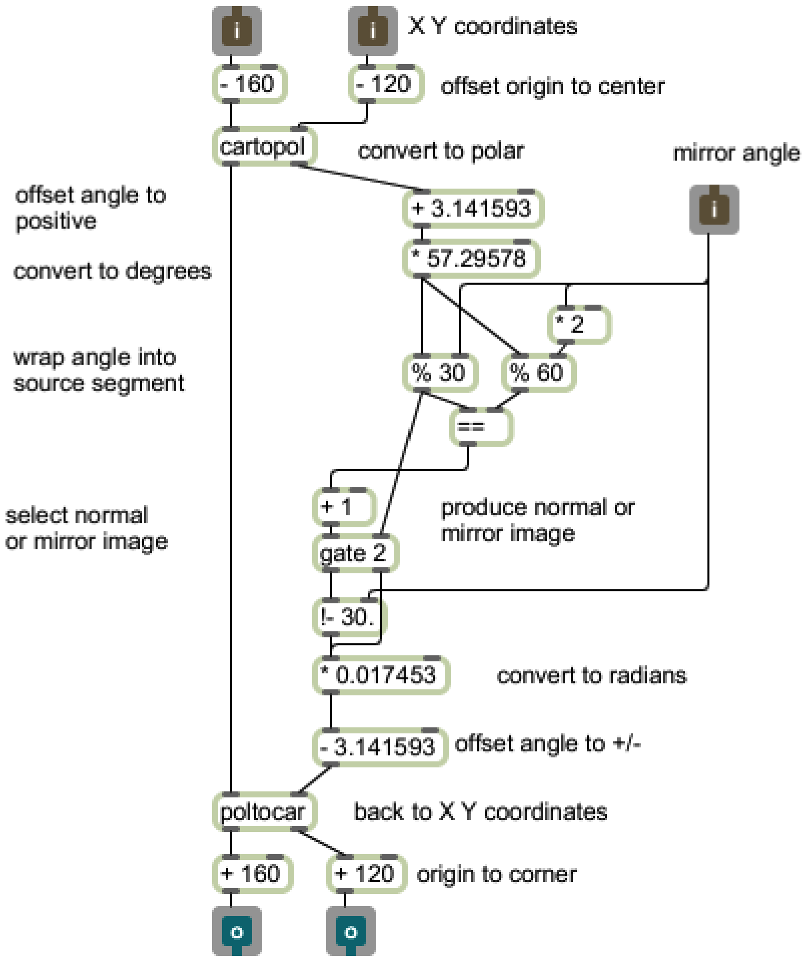
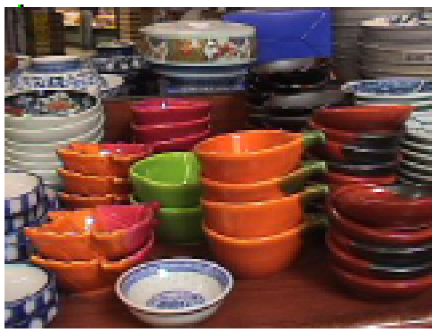
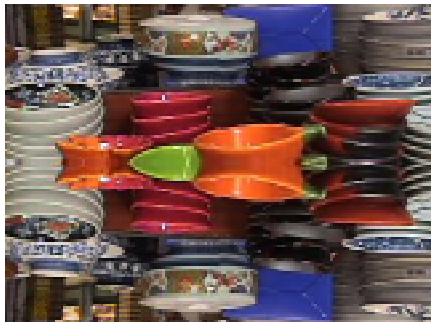


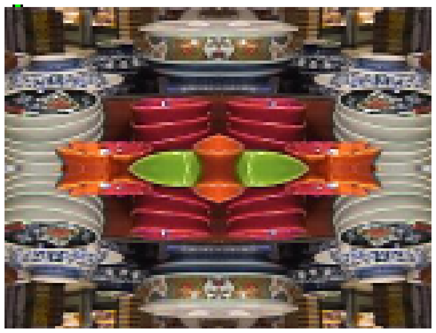
Figure 21

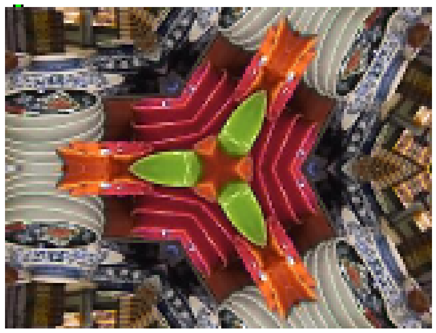
Figure 21 will do the job. At the top it converts the XY coordinates to polar coordinates, just like the rotation subpatches. We can visualize this as scanning all of the pixels in a circular fashion. Next the angle is offset so it runs 0 to 6.28 instead of –3.14 to 3.14. This makes it possible to use a simple modulus operation to determine what each segment needs. In this example I convert the radians of the angle outlet to degrees, primarily because it makes the math easier to follow. (It still works if you leave angles in radians.) The next step is a little tricky. The angle this cell should look at (in polar coordinates) is the remainder that is obtained when the mirror angle is divided into the original angle of the cell. This is provided by the % operation. Half the segments need to be a reflected version, which is easily produced by subtracting the remainder from the mirror angle. In effect this makes the scan go backwards. The choice is made by comparing the remainder found by dividing the lookat angle by both the mirror angel and twice the mirror angle. If they are the same, we want the normal image. If they are different, the reflected image is appropriate.

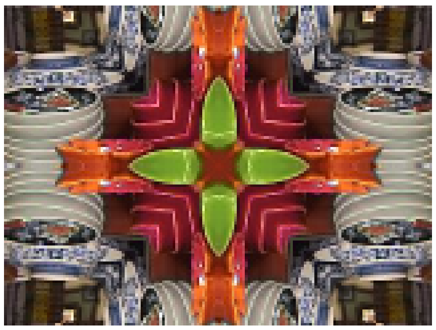
After this is done the angle is converted back to radians from –pi to pi, and the polar coordinates back to Cartesian. Here are some examples with various mirror angles.


original


180 degrees


90 degrees


60 degrees


45 degrees

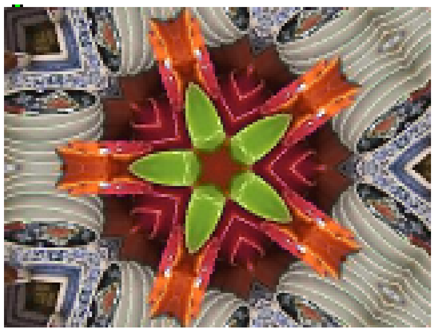
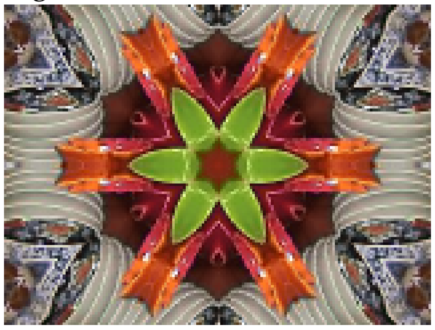
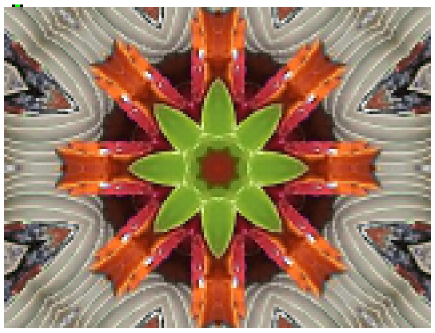
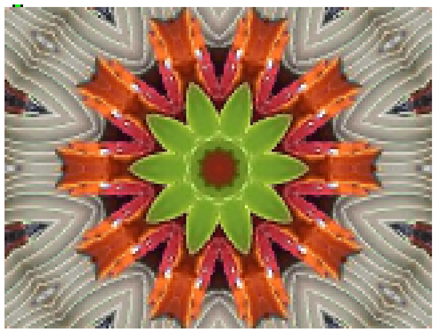
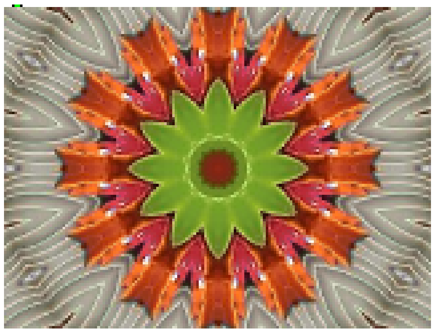

36 degrees

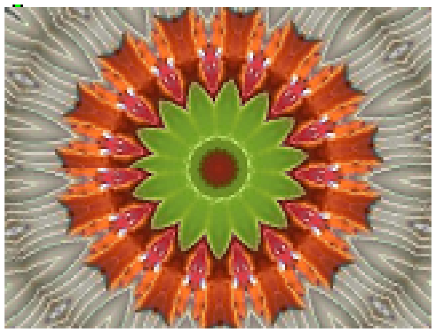
Figure 22.

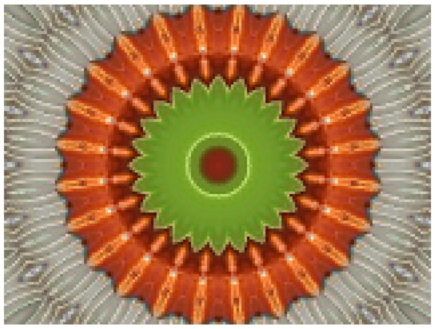

30 degrees

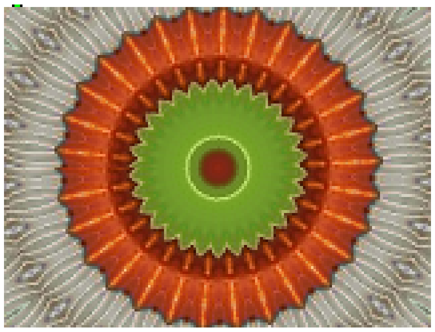

22.5 degrees


18 degrees


15 degrees


12.25 degrees


7.5 degrees


6.125 degrees

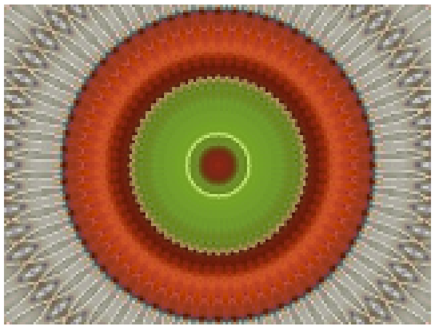

3 degrees

Figure 23.

## *Using relative mode*

Jit.repos has two modes, which determine the precise effect of the control matrix. Heretofore we have used absolute mode (0), where the source of an output cell is the location specified in the control matrix. In relative mode (1) the source of an output cell is the address of the cell plus the contents of the same cell in the control matrix. If you are in relative mode, and the control matrix is filled with 0s, the input is passed straight through. Relative mode can be responsive in real time, as follows.

First I will create a control matrix that contains some simple trig functions. The make control patcher has to be modified to handle floats as shown in figure 24.

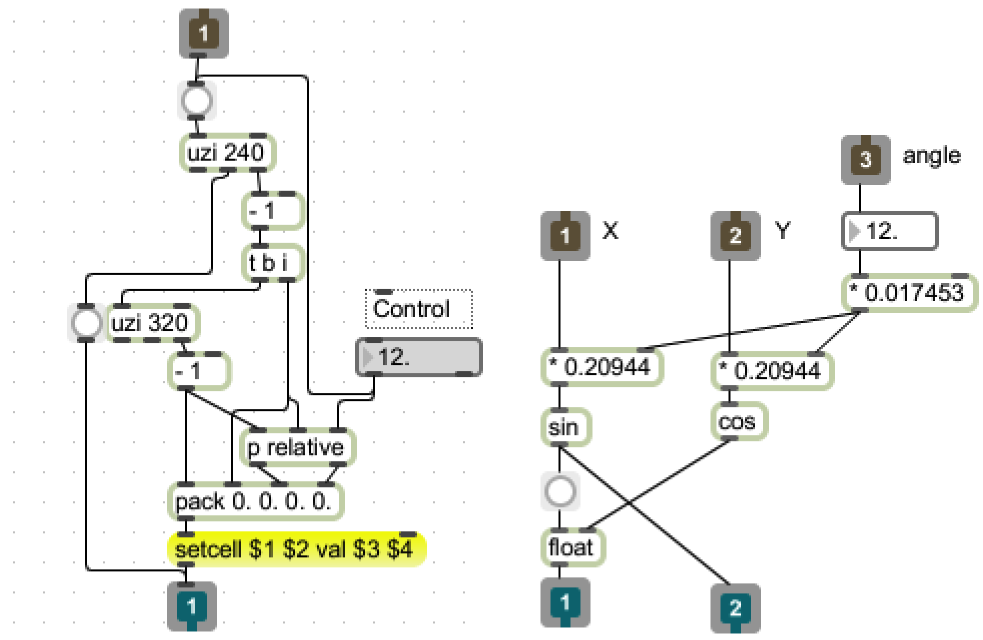


Figure 24.
Now the control matrix is set up like this:

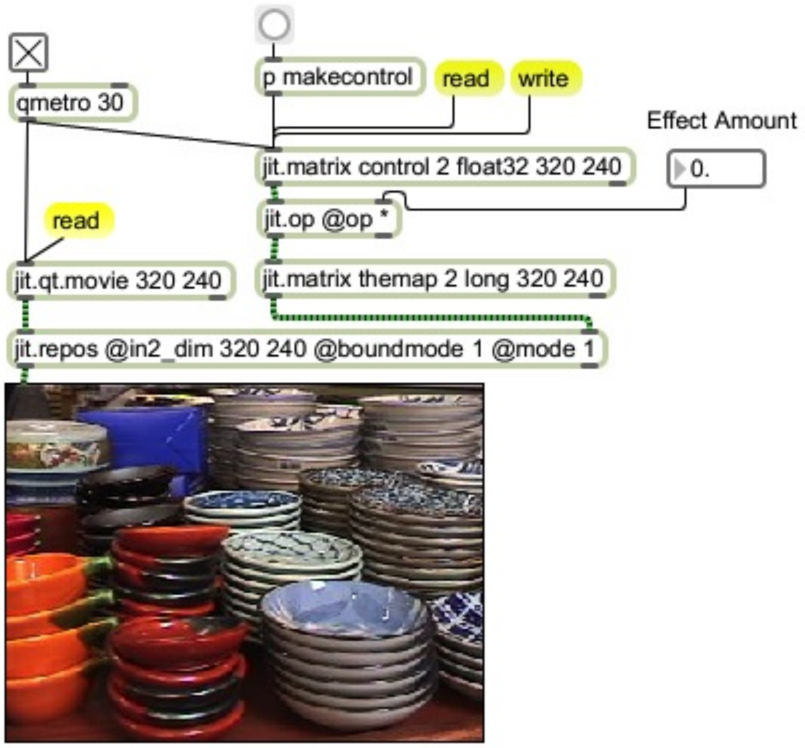
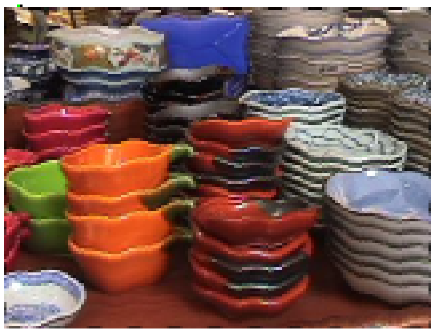
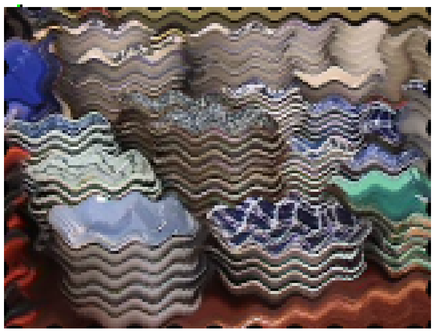
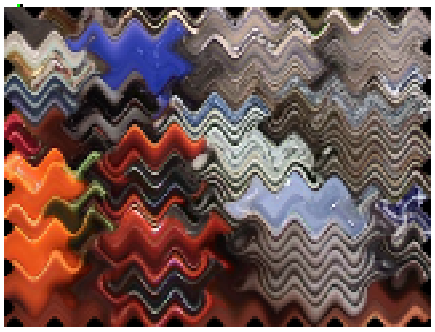

qmetro 30
p makecontrol
read
write
Effect Amount
jit.matrix control 2 float32 320 240
0.
jit.op @op *
read
jit.qt.movie 320 240
jit.matrix themap 2 long 320 240
jit.repos @in2_dim 320 240 @boundmode 1 @mode 1

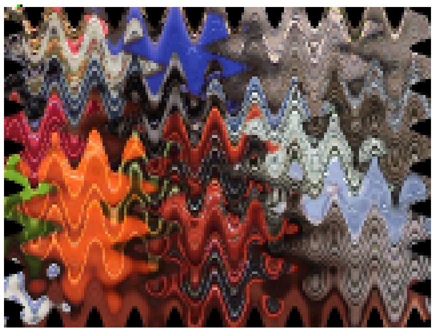
Figure 25
The matrix applied to jit.repos must be char or long. Since chars cannot take negative values, long is used the most. In figure 25, the float trig functions from the control matrix are run through jit.op for scaling and converted to longs. The multiplier in jit.op will determine how much effect is applied, and since this is

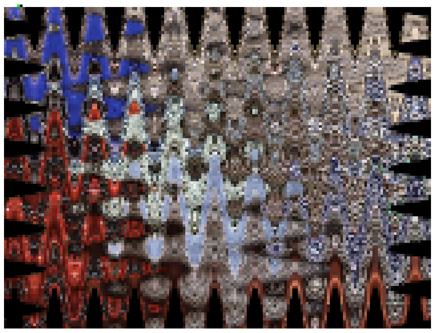
just a multiply on a pre-computed matrix, it will respond instantly. Here are some examples.

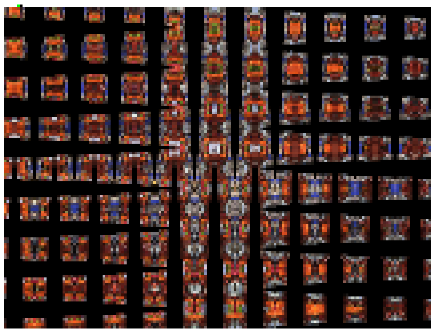

Amount 2.


Amount 4.


Amount 8.


Amount 16.


Amount 32


Amount 128.

## *Dynamic Repos*

### Using quicktime

Relative repos control matrices can be stored as a quicktime movie, enabling some nifty animations. This example creates an expanding ripple. It starts with a function generator:

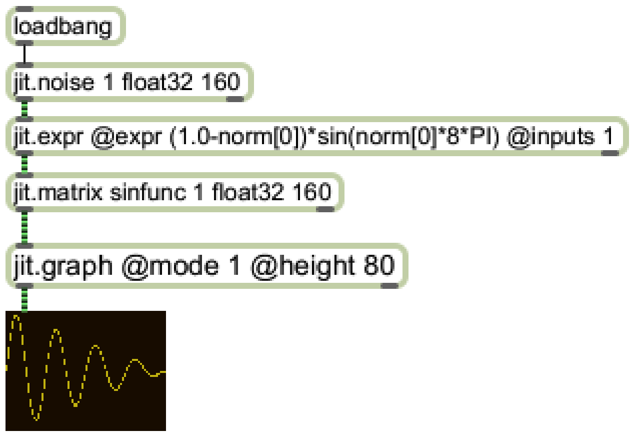

loadbang
jit.noise 1 float32 160
jit.expr @expr (1.0-norm[0])*sin(norm[0]*8*PI) @inputs 1
jit.matrix sinfunc 1 float32 160
jit.graph @mode 1 @height 80

Figure 26.
This part of the patch is dedicated to filling the matrix with the function

$$F[x] = (1-kx)\sin(x)$$

with x ranging from 0 to 8π. That will be a sine wave that fades over 4 cycles, as shown.

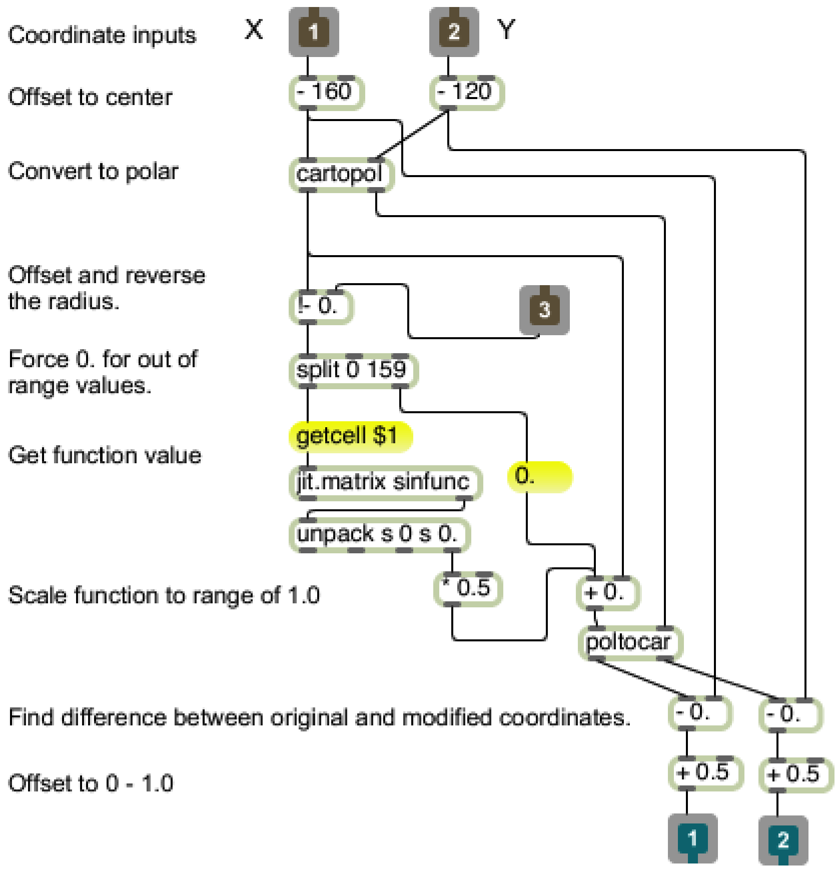


Figure 27.
Figure 27 is the mechanism for applying the function. This will plug into a modified makecontrol patch (figure 29.) The X and Y coordinates are centered, then converted to polar form. The angle is unmodified, but the radius has a value from the function added. The radius is only changed a bit, but that will eventually be magnified. The value obtained from the function is offset by a control that varies with each frame. Note that the offset is reversed by subtracting the radius from the control. The control value will range from 0 to 320, so the offset radius will sweep from -160 to 0 at the start and from 160 to 320 at the end. In conjunction with the split object, this will present three possibilities for each radius operation. At the start, all values will be negative, and produce no change to the image. As the offset expands, pixels from the center to the offset will be modified by the function. Note the function is highest in amplitude at the beginning. Eventually, the offset becomes larger than the image and the center area is unaffected. The control matrices are visualized in figure 28. Gray represents no change.

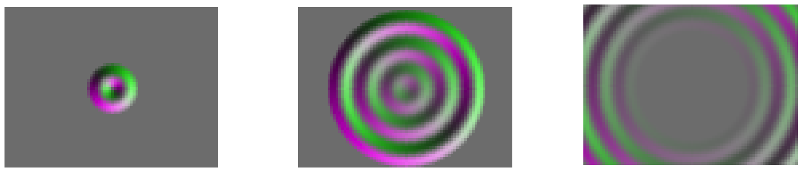


Figure 28

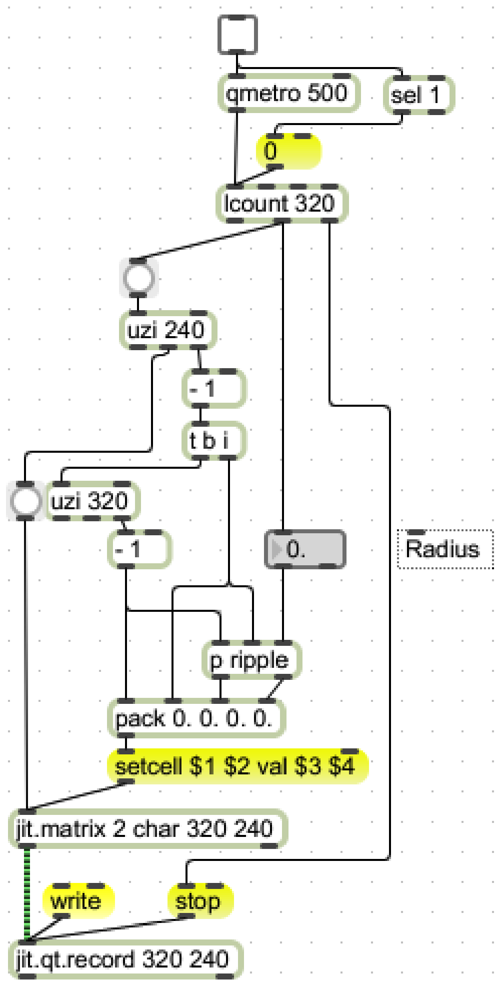


Figure 29.

The patch of figure 29 generates the series of matrices. For each frame, a set of uzis produce coordinates for figure 27 (included in the ripple sub-patch) and the results are stuffed into a matrix of char that can be recorded by jit.qt.record. The float values produced by the mechanism of figure 27 are interpolated  to chars in the range 0 255. These values need some massage to control the repos as shown in figure 30. These operations reverse the steps. The char matrix output by jit.qt.movie is converted to float32, 0.5  is subtracted, then the effect is multiplied as desired.

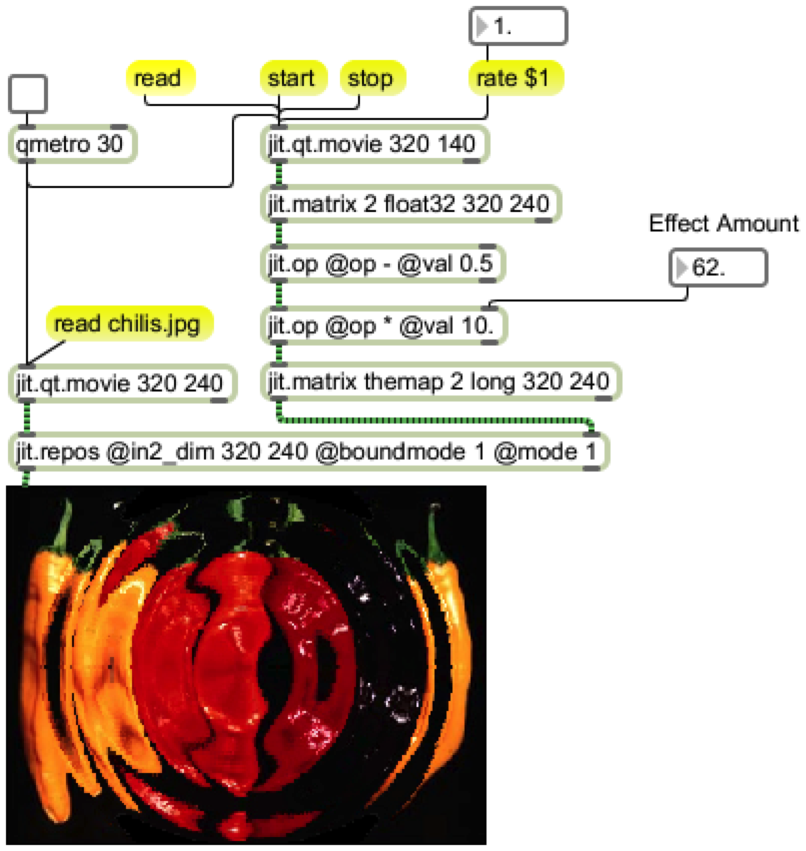


Figure 30.

Similar animations can be created by capturing control frames in a matrix set, but the movie approach is more flexible.

## Using Matrixsets

There are some limits to the use of QuickTime movies to store repos control matrices. Since the QuickTime format converts all data to 24 bit color, cell values greater than 255 cannot be stored. This pretty much limits QuickTime control to relative mode. For more grandiose animated transformations, matrixsets are the best choice. These can be quite large, so memory could be an issue. The mechanism for recording repos action into a matrix set is very much like figure 29.

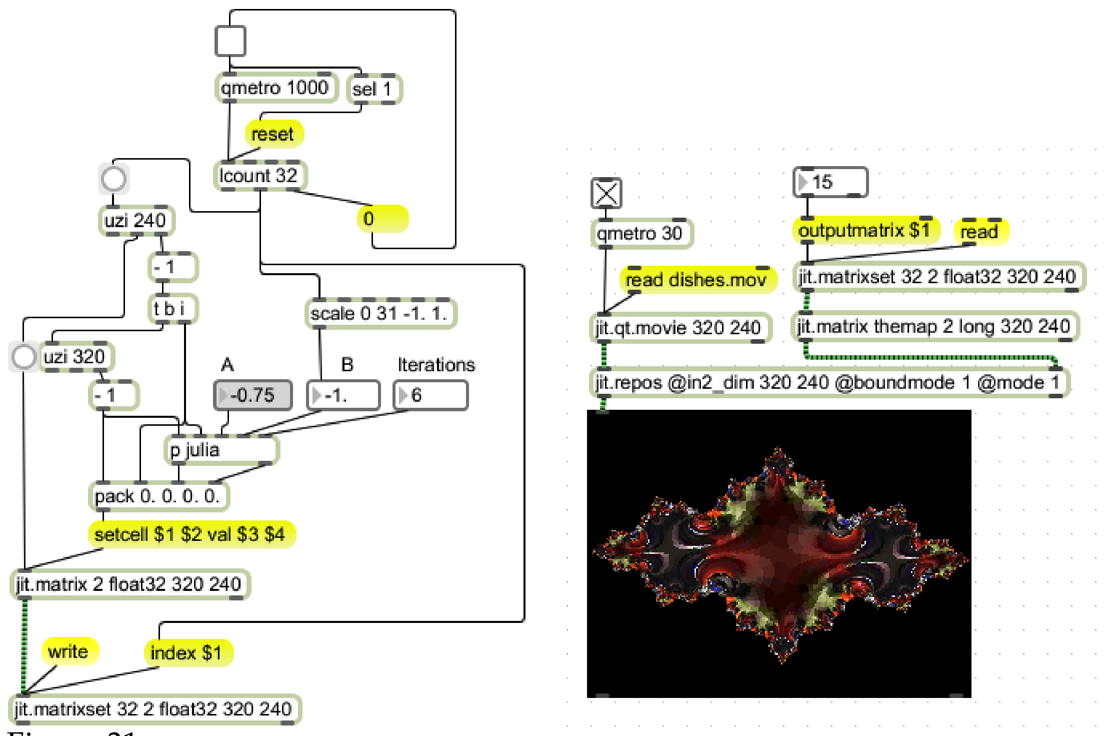


Figure 31.

Playback is simple too. The image shown in figure 32 is derived from Julia sets. To find out more, read on.

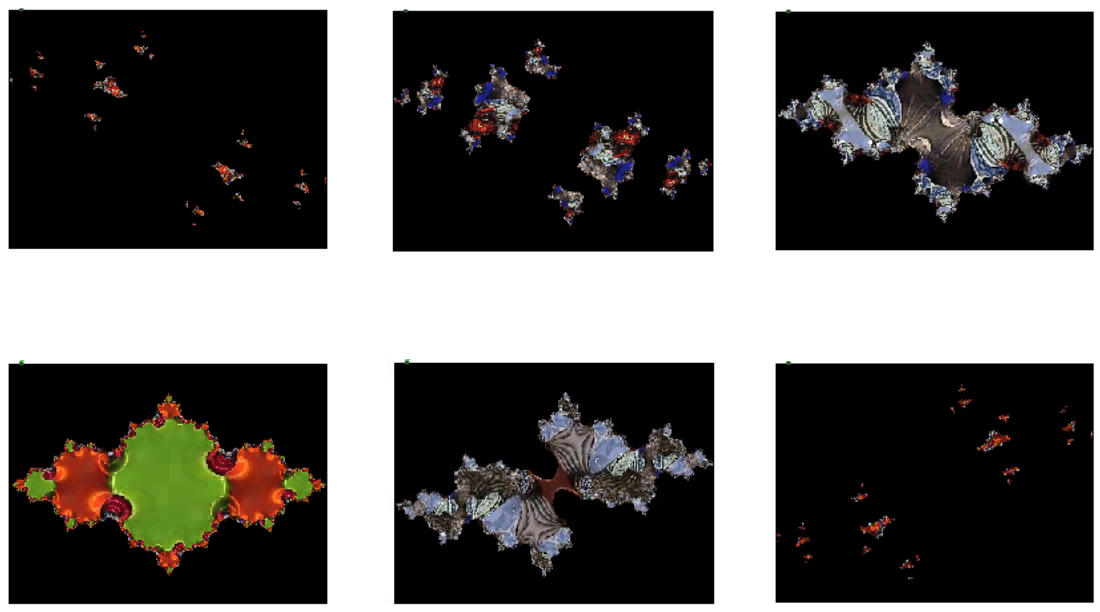


Figure 32. Several stages of repose controlled by matrixset.

## Repos and the Julia set

The Julia set is an interesting construct. It is a grouping found by iterating this formula

$$Z_{i+1} = Z^2 + C$$

Equation 1.

Where Z and C are complex numbers. An equivalent formula pair that's appropriate to our needs is:

$$X_{n+1} = X^2 - Y^2 + A$$

$$Y_{n+1} = 2*X*Y + B$$

Equation 2.

These map the real part of Z to X and the imaginary part to Y, with A the real part of C and B the imaginary part of C. The typical application for these is to try hundreds of iterations for various values of A and B. If the X and Y values stay on the screen, the value pair (or C ) is a member of the set. This is the famous Mandelbrot set. (figure 33.)

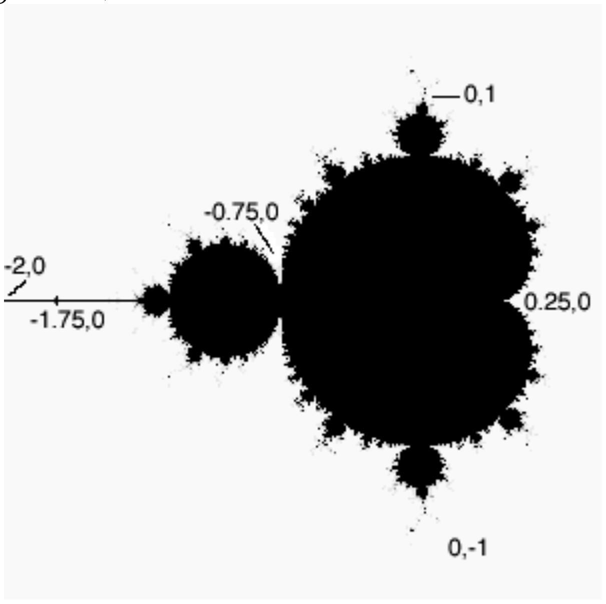
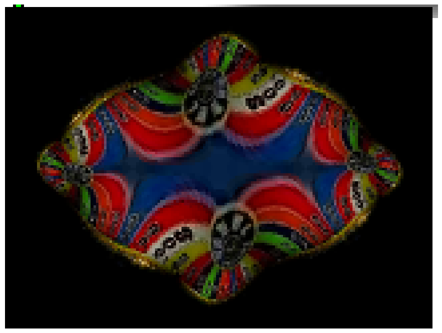
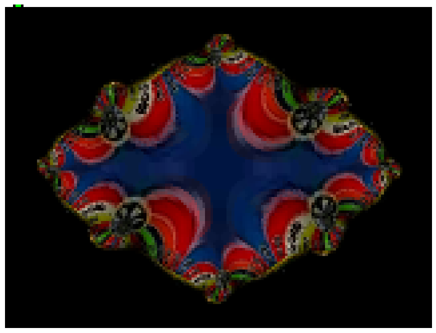


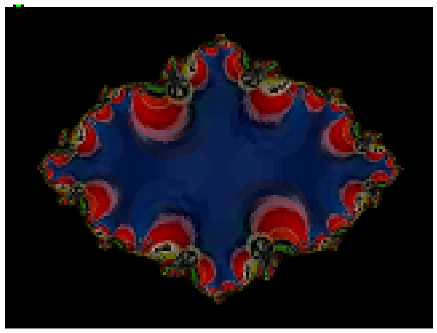
Figure 33. The Mandelbrot set

Most mathematicians are interested in a simple case for any starting pair-- does it stay on the page as the formula is iterated, or do the values increase to infinity.

A Julia set is the set of Z (X,Y pairs) that stay on the screen for a given C. These sets may be spots all over the screen, or may form uniquely twisted shapes. Figure 34 shows one example for C = -0.38. The screen area is -1 to 1 in the Y direction. The black areas of the image are values that have jumped out of range (they will never return) and the blue areas are a value that are approaching 0,0.


A = -0.38  B=0. 1 iteration


2 iterations


3 iterations

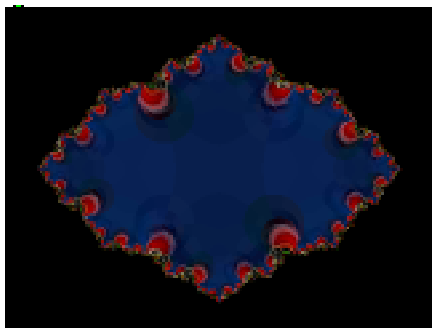
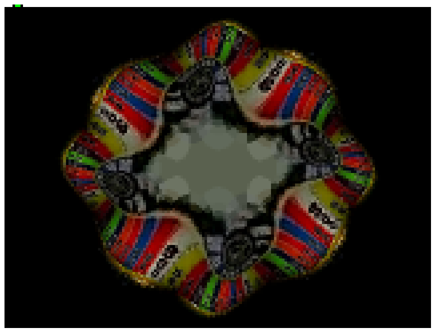

5 iterations

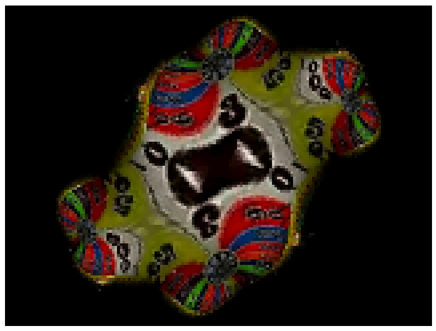
Figure 34.

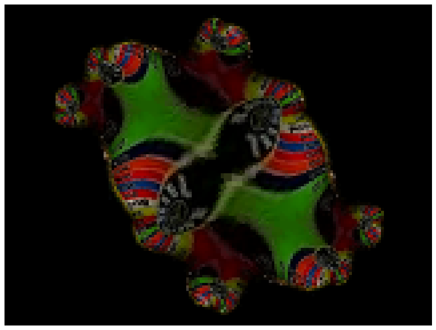
The colored portion of the image is a Julia set. Further iterations make the red and green regions thinner and the edge more crinkly.

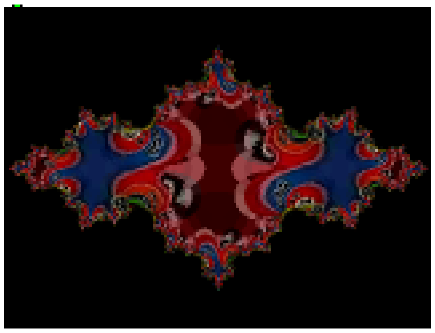
Even if you don't understand the math concepts here, note that these formulas take X and Y in and give a new X and Y out. That's grist for jit.repos. For each cell in the control matrix, transform its own coordinates with equation 2. Even if we just apply a few iterations, we see transformations that are uniquely strange.

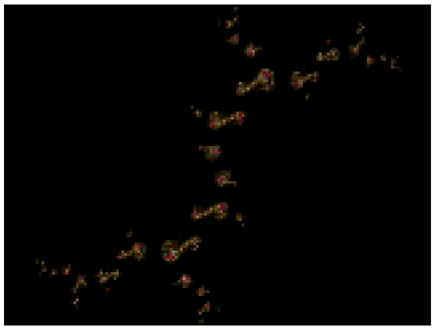
What generally happens is that one iteration gives a nearly recognizable twisted image, and additional iterations begin to approach the shape of the set. Different A and B values give very different shapes:


A = 0.075  B = 0.   1 iteration


A = 0.075   B = 0.43  1 iteration


A = 0.075   B = -0.43 2 iterations


A = - 0.75  B = 0  6 iterations


A = 0     B = 1   6 iterations

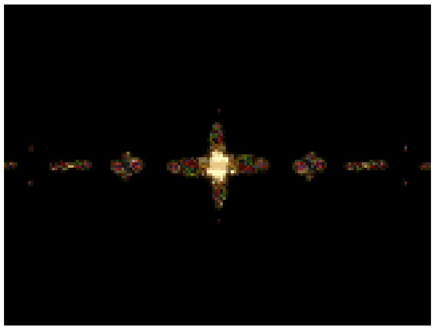
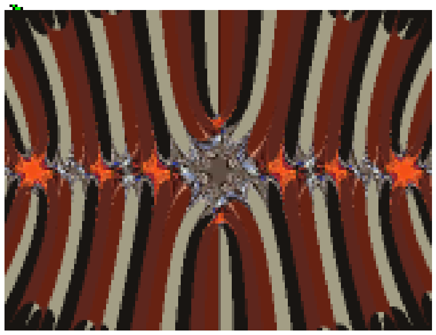
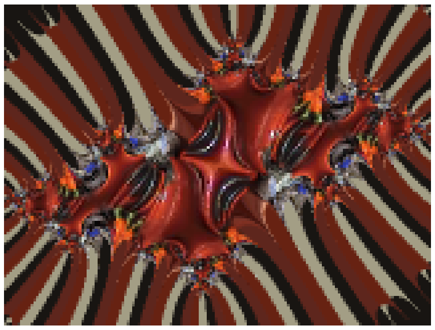
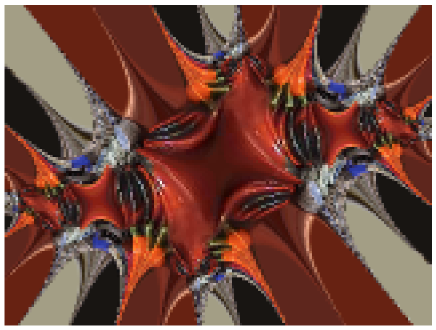

A = -1.75   B = 0  4 iterations

Figure 35.

There are too many shapes and variations to give examples of every possibility. One nice feature is that slight changes of A or B (on the order of 0.01) give slight variation s in the shape. It's possible to set up a slow evolution  of the shape that way. Since these are repositionings, different source materials will show up in strikingly different ways. The source here is the wheel movie, which has a black outer edge. If a frame is colored at the  edge some interesting transitions show up. Here's the dishes movie with boundmode = 3


A = -1.75 B = 0  4 iterations


A= -0.69   B = 0.44  6 iterations


A = -0.69 B = 0.44  2 iterations

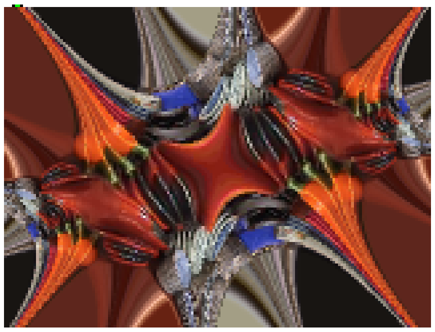

A = -0.69 B = 0.44  1 iteration

Figure 36

Most of the usable A and B values are found within the Mandelbrot set, although values off the set will give interesting forms with only one or two iterations.

A in  the set ranges from –2 to 0.35 and B from – 1 to 1. The really twisted images come from the edge zone.

Here's how the Julia set generator works:

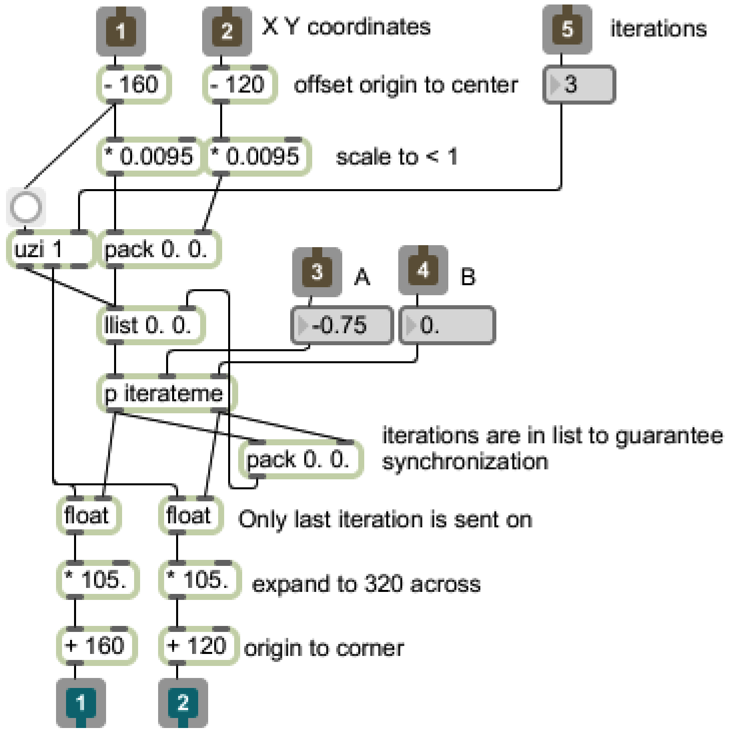


Figure 37.

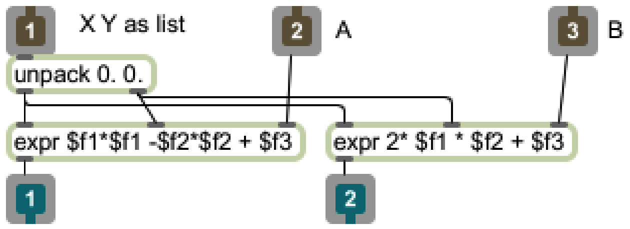


Figure 38. Contents of iterateme

The uzi in figure 37 determines the number of iterations. The values from A and B set the constants in the iteration. The first X and Y values for th computation are taken from the coordinates on the screen. The X and Y results of each iteration are then reapplied to the formula as many times as desired.