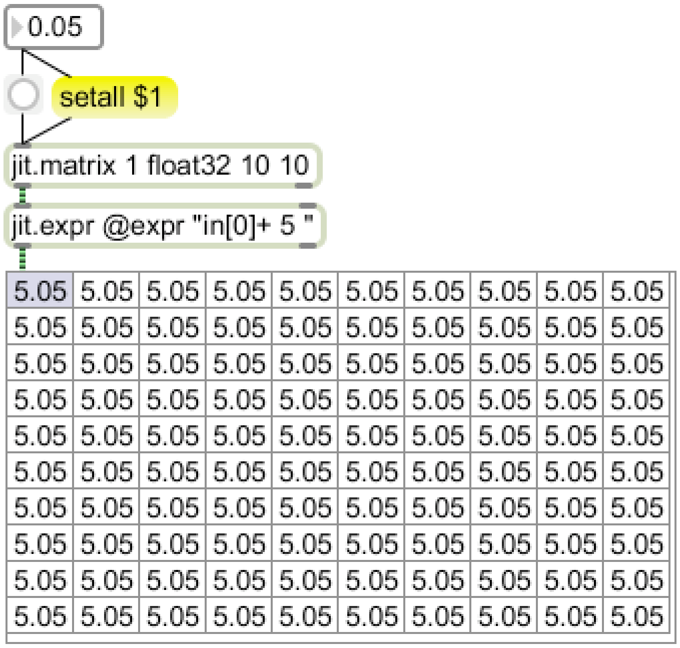
Advanced Math in Jitter

Jit.expr

Jit.expr is one of the more enigmatic Max objects. Like expr, and vexpr it exists to allow code level manipulation of data. Unlike expr, it works on entire matrices in one go.

The key argument to jit.expr is the @expr attribute, which is the math expression to apply to each cell. The expression may be contained in double quote marks. This is not necessary if there are no spaces in the expression, but I find spaces make math easier to read[1]. Figure 1 shows the archetypical jit.expr test patch. A small matrix is constructed with the same value in each cell. This is fed to specimen jit.expr expressions and the results read from a jit.cellblock.



5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05
5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05
5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05
5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05
5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05
5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05
5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05
5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05
5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05
5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05	5.05

Figure 1.

You will notice from figure 1 that jit.expr has a unique syntax. In place of the $i1 and $f1 tokens to denote input data, jit.expr uses the variable expression in[*] where the bracket contains the inlet number. You will also notice that the leftmost inlet is numbered 0. The object does not automatically provide enough inlets for the expression-- if more than two are needed, the @inputs attribute will produce them. There can be at least 64 inlets, although I can't imagine any math expression complex enough to use that many.

When jit.expr receives a matrix, it applies the expression to each cell in the input matrix and outputs a matrix of the same size. If you want to define a constant size for the output

---

[1] If you use quotes, but don't have any spaces, the quotes will vanish when the object is completed.

matrix, use the @dim attributes. When that is done, an incoming matrix is interpolated across the defined dimensions as with any other matrix. (Srcdim and destdim functions are not available, however.)
Jit.expr does all math in float format. If you apply chars to the input, they are normalized to the range 0 to 1.0 Since the char data type is unsigned, make sure operations do not involve negative values.

If there is only one expression, it is applied to all planes of the input matrix. You may also supply one expression per plane if you like. The in[*].p[*] variable can be used to access a value by input and plane. You can also specify a jit.matrix by name as input, and add .p[*] to specify a plane in that matrix.

Many operations require the coordinates of the cell. These are available in three forms:

- cell[†] gives an integer coordinate, where † denotes the dimension[2].
- norm[†] gives a coordinate normalized to the range 0.0 to 1.0.
- snorm[†] gives a coordinate normalized to the range -1.0 to 1.0.
- dim[†] provides the size of the requested dimension.

The fundamental operators available to jit.expr are the same as those in jit.op, with the exception of char only operations, and variants of pass[3]. The functions require one or two arguments in parentheses. When there is more than one argument, they must be separated by a comma, which is written \, unless the @expr is in quotes.

Function	Operation	Example
*	multiplication;	"in[0] * 3"
/	division;	"in[0] / 3"
+	addition;	"in[0] + 3"
-	subtraction;	"in[0] - 3"
%	modulo;	"in[0] % 3"
abs	absolute value;	"abs(in[0])"
min	minimum;	"min(in[0], 3)"
max	maximum;	"max(in[0], 3)"
avg	average;	"avg(in[0].p[0], in[0].p[1])"
absdiff	absolute value of difference;	"absdiff(in[0].p[0], in[0].p[1])"
fold	mirrored modulo;	"fold(in[0], 3)"
wrap	positive modulo;	"wrap(in[0], 3)"
sin	sine;	"sin(norm[0]* TWOPI)"
cos	cosine;	"cos(norm[0]* TWOPI)"
tan	tangent;	"tan(norm[0]* TWOPI)"
asin	arcsine;	"asin(norm[0])"
acos	arccosine;	"acos(norm[0])"
atan	arctangent;	"atan(norm[0])"

---

[2] Starting with 0, of course. This is in the order specified in the original matrix.
[3] Actually, you can write "pass(in[0])" but the point of that is unclear.

atan2	Arctangent of X and Y;	"atan2(snorm[0], snorm[1])"
sinh	hyperbolic sine;	"sinh(norm[0]* TWOPI)"
cosh	hyperbolic cosine;	"cosh(norm[0]* TWOPI)"
tanh	hyperbolic tangent;	"tanh(norm[0]* TWOPI)"
asinh	hyperbolic arcsine;	"asinh(norm[0])"
acosh	hyperbolic arccosine;	"cosh(norm[0])"
atanh	hyperbolic arctangent;	"atanh(norm[0])"
exp	e to the x;	"exp(in[0])"
exp2	2 to the x;	"exp2(in[0])"
ln	log base e;	"ln(in[0])"
log2	log base 2;	"log2(in[0])"
log10	log base 10;	"log10(in[0])"
hypot	hypotenuse(binary);	"hypot(snorm[0], snorm[1])"
pow	x to the y(binary);	"pow(in[0], 3)"
sqrt	square root;	"sqrt(in[0])"
ceil	integer ceiling;	"ceil(in[0])"
floor	integer floor;	"floor(in[0])"
round	round to nearest integer;	"round(in[0])"
trunc	truncate to integer	"trunc(in[0])"
		Logicals consider any non-0 true[4]
&&	logical and;	"in[0] && in[0].p[1]"
||	logical or;	"in[0] || in[0].p[1]"
!	logical not(unary);	"! in[0] "
>	greater than;	"in[0] > in[0].p[1]"
<	less than;	"in[0] < in[0].p[1]"
>=	greater than or equal to;	"in[0] >= in[0].p[1]"
<=	less than or equal to;	"in[0] <= in[0].p[1]"
==	equal;	"in[0] == in[0].p[1]"
!=	not equal	"in[0] != in[0].p[1]"

Many of the most common numbers can be called up by name. These are called constants, and using them is much more efficient than, for instance, sqrt(2).

Constant	Value	Notes
PI	3.141593	
TWOPI	6.283185	
HALFPI	1.570796	
INVPI	0.31831	
DEGTORAD	0.017453	Multiply by angle in degrees to get radians
RADTODEG	57.29578	Vice versa
E	2.718282	
LN2	0.693147	Natural logarithm of 2

---

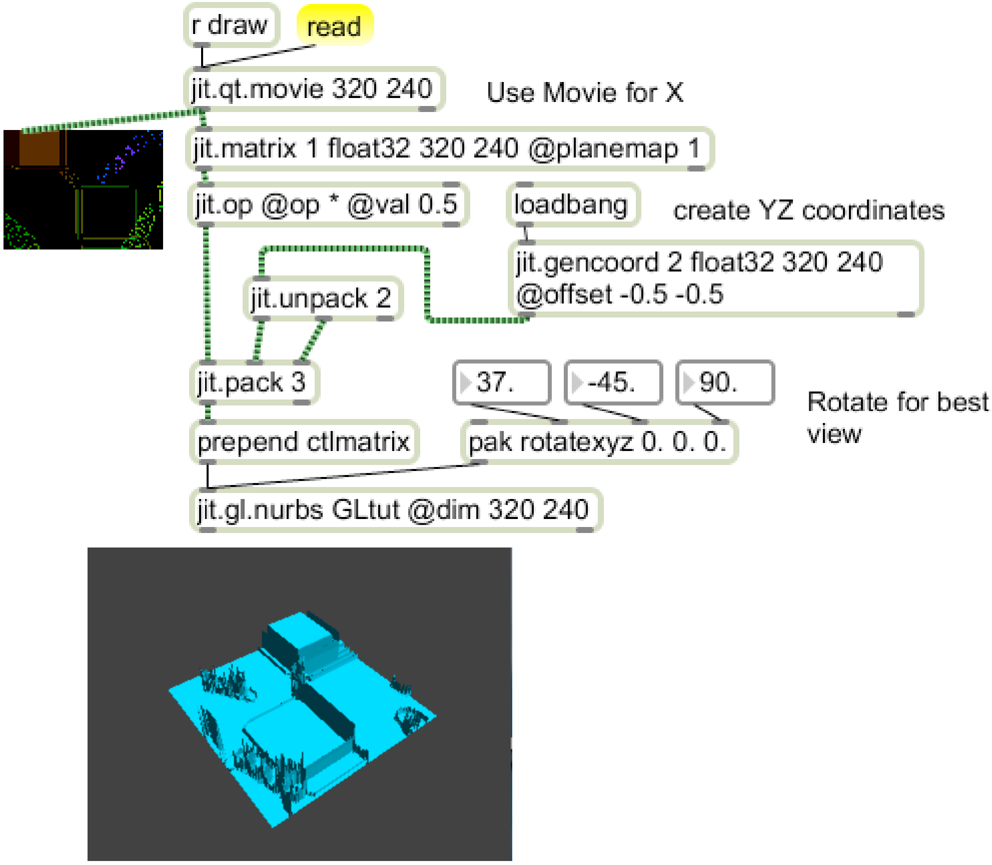
[4] Logical operations evealuate the truth of an expression and return 0 or 1, but there is no equivalent of an if statement.

LN10	2.302585	
LOG10E	0.434294	Logarithm base 10 of e
LOG2E	1.442695	Logarithm base 2 of e
SQRT2	1.414214	
SQRT1_2	0.707107	sqrt(0.5)

The advantage of jit.expr over jit.op is the possibility of complex operations in one object. There is some penalty in moving matrices around, so combining 3 jit.ops into 1 jit.expr will improve the frame rate. The advantage of jit.op over jit.expr is found in the various conditional pass operations.

Jit.gencoord

The @expr "norm[0]" "norm[1]" will fill a matrix with a set of coordinates that are useful to many jitter objects. For example, a jit.gl.nurbs surface is typically generated by specifying a regular gird in the Y and Z coordinates and applying some interesting information to the X plane. The need for coordinate generation is common enough that there is a dedicated object, jit.gencoord. This object produces a set of coordinates in X and Y with extra control offered by scale and offset attributes. Figure 2 shows a typical use.



```
r draw
read
jit.qt.movie 320 240
jit.matrix 1 float32 320 240 @planemap 1
jit.op @op * @val 0.5
loadbang
jit.gencoord 2 float32 320 240 @offset -0.5 -0.5
jit.unpack 2
jit.pack 3
37.
-45.
90.
prepend ctlmatrix
pak rotatexyz 0. 0. 0.
jit.gl.nurbs GLtut @dim 320 240
```

Figure 2. Jit.gencoord and jit.gl.nurbs

Note that the coordinates usually only need be generated once. Then they can be efficiently passed on for further calculations.

Jit.bfg

Jit.bfg is a basis function generator. Basis functions are formulas that produce curves and surfaces that are mathematically interesting. For instance, the sinc function is expressed in the formula:

$$\operatorname{sinc}(x) = \frac{\sin(\pi x)}{\pi x}.$$

When jit.bfg is asked to produce this, it takes the equivalent of norm[†] as the x value and outputs a matrix of the specified size. If a matrix is applied to jit.bfg, the matrix dimensions are used, but values in the matrix are ignored. The output of the sinc function is graphed in figure 3, first as a matrix visualization and on the right as interpreted by jit.graph.

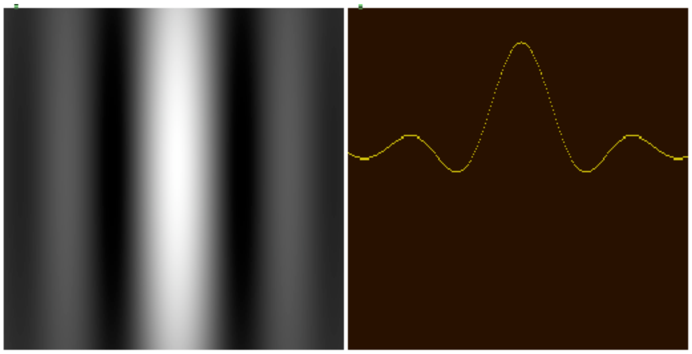


Figure 3.

Figure 3 has been manipulated by scale and offset values sent to jit.bfg[5]. These affect the coordinate values used for x. An offset of -12 and a scale of 15 would produce the input term (15*norm[†]-12). There are further manipulations possible by the origin (which is added to the coordinates before scaling), and rotation (which introduces a cos(norm[†]) operation). These are independently set for each dimension, which can produce a rich set of patterns:

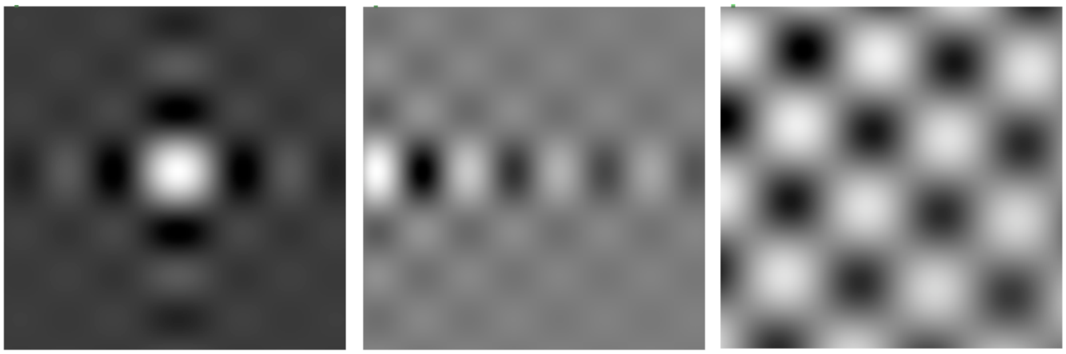


Figure 4.

---

[5] The peak of the sinc curve occurs at x = 0.

What are these good for? They can process images by multiplication, they can create NURBS based shapes, they can produce unique repos patterns-- the possibilities are endless.

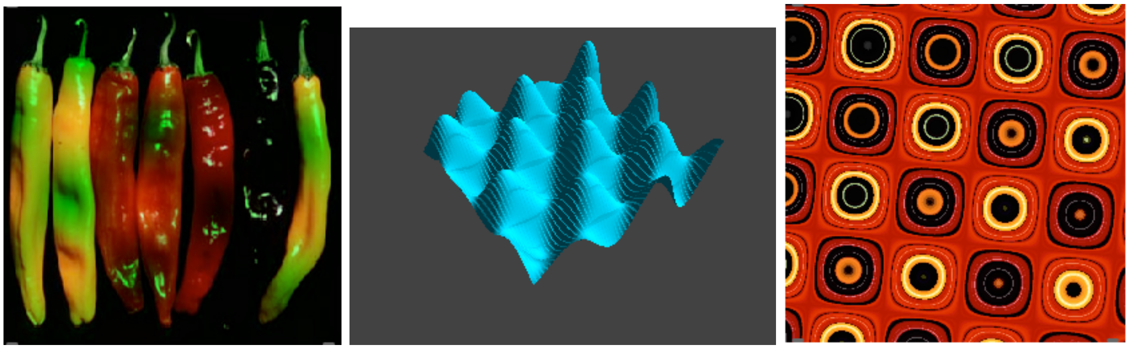


Figure 5.

Figures 6, 7 and 8 show how some of this is done.

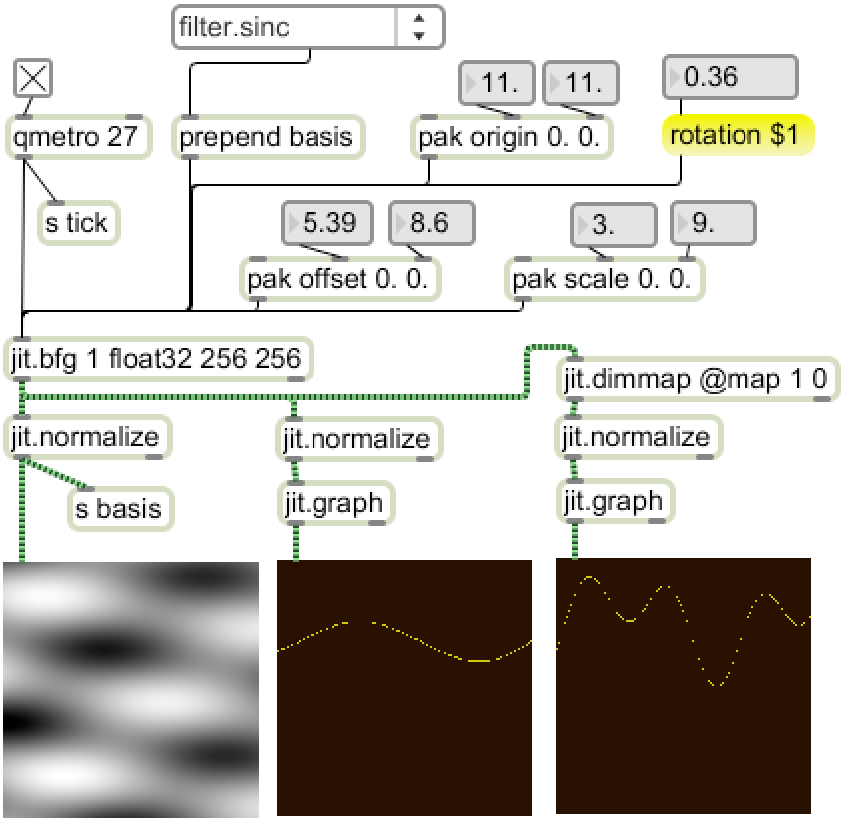


Figure 6.

Jit.bfg has a tutorial (#50) that demonstrates all of the functions[6], and the help file explores the fractal functions is some depth.

---

[6] At the time this is written, some functions are not working-- if a particular basis produces a black screen in the help file, just skip it for now and try another.

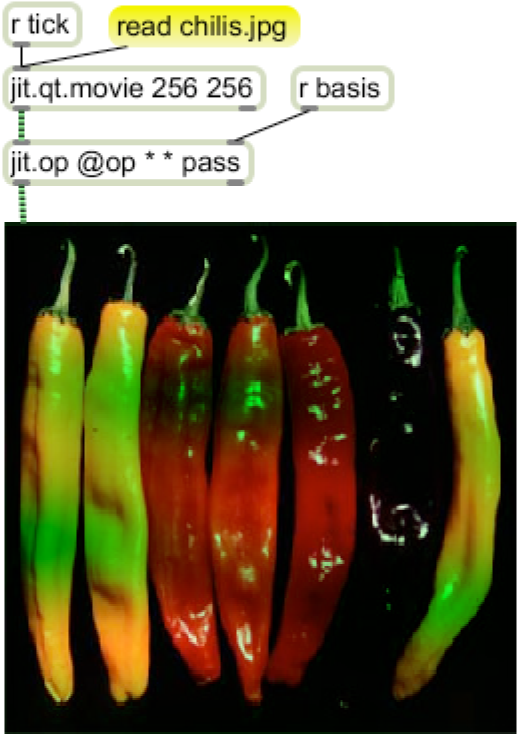


Figure 7.

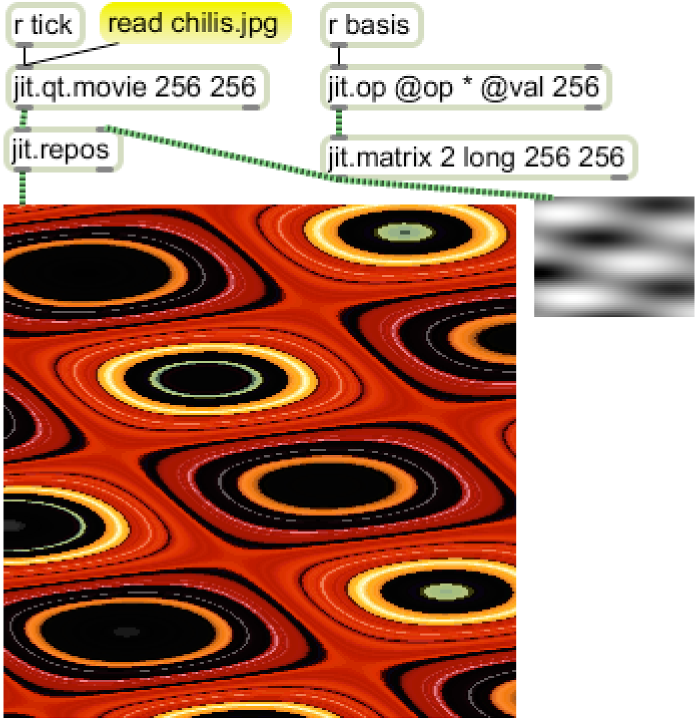


Figure 8.