Advanced Math in Max

Eventually we will need to do something more complicated than the ordinary math objects allow. This is not a tutorial on how to do that math, but a guide to where the math can be done.

Basic operators:

+	input plus argument
-	input minus argument
!-	argument minus input
*	input times argument
/	input divided by argument
!/	argument divided by input
%	input divided by argument and the remainder returned.

C library functions

Trig functions and the like used to require expr, but in Max 4 some are available as objects.

abs	gives absolute value of input.
pow	raises input to the power of the stored value. The result is always float. Negative powers are roots, of course.
sqrt	gives square roots.
sin, tan, cos	find sine, tangent or cosine of angles. The input is in radians. (You will remember that there are 2π radians in a full circle. To convert from degrees to radians, multiply by 0.01745.)
asin, tan, acos	give arcsine, arctangent or arccosine. The results are in radians.
atan2	Arctangent of x/y
sinh,tanh,cosh	Hyperbolic versions
asinh, atanh atanh2,acosh,	Hyperbolic versions
cartopol	Cartesian to polar coordinate conversion
poltocar	polar to Cartesian coordinate conversion

If you have trouble remembering exactly which ratio is the cosine anyway, just recite SOH, CAH, TOA until you feel better about it. The important thing to remember is that the sine is the one that starts on 0.

Math with signals

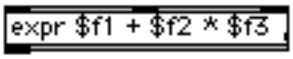
There are a set of similar objects in MSP. These indicated as

 (for instance) and process signals. The thing to remember is that signals are continuous, so connecting two signals to the same inlet is not the same as

connecting two floats. In the latter case you get the most recently triggered message. Two signals at the same inlet are added.

Expr

The expr object allows you to enter complex expressions. This can save a lot of space, and provides access to even more functions. To define an inlet, include $i1 or $f1 in the expression. The $i version is treated as an integer, the $f version is treated like a float. The numeral indicates which inlet, up to 9. Given these rules, the expression



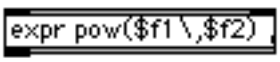
will return the input plus the product of stored values 2 and 3. There are rules of precedence when operations are mixed like this. From highest to lowest:

- ~ bitwise not
- ! logical not
- - negation
- *, /, % multiplication, division, remainder
- +, - addition and subtraction
- <<, >> left and right shift
- < > <= >= comparisons (which will be 1 or 0)
- & bitwise AND
- ^ bitwise XOR
- | bitwise OR
- && logical AND
- || logical OR

Precedence can be altered by parentheses, so [expr ($f1 + $f2) * $f3] multiplies the sum of input 1 and 2 by input 3. Be careful when you type parentheses -- if they don't balance out, you'll get an error message and all connections will be broken. Occasionally you'll get an "expression too complicated" error. If so, break the work into two exprs.

Functions in expr

C library functions are available in the expr object. Many C functions take two arguments, such as pow(a,b). Since Max is skittish about commas, you have to write this as



The functions implemented as objects (listed above) are all in there. The extra functions are:

exp($f1) gives e to the $f1
log($f1) and ln($f1) both give natural logarithm of $f1
log10($f1) gives the base 10 logarithm
fact($f1) gives factorial

Comparisons

All Max comparison operators return 1 if the comparison is true and 0 if the comparison is false. A string of false attempts will give a string of 0s. If you need a function that only gives output in the true cases, use select. If you only want to detect the change from true to false, use a change object on the output. Here's a list of the operators:

- < input less than stored value
- > greater than
- == equal to (there is no = object)
- != not equal
- <= less than or equal
- >= greater than or equal

Logic

The logic operators consider 0 to be false and any integer (including negatives) to be true. A float input is truncated, so it has to be 1.0 or more to be true. Note the difference between integer logic and bitwise logic, in the next section.

&& AND :

0 AND 0 = 0
1 AND 0 = 0
1 AND 1 = 1

|| OR :

0 OR 0 =0
0 OR 1 = 1
1 OR 1 = 1

! NOT :

NOT 0 = 1
NOT 1 = 0.

(This is an Lobject; Max does not have a not function except in expr.)

^ XOR :

0 XOR 0 = 0
1 XOR 0 = 1
1 XOR 1 = 0

There is no XOR operator, except in expr and the Lobject Llogic. The Max logic operators are just like all other math objects in that only left inlet triggers the output. The Llogic functions are gates, giving output for any change in input.

Bitwise Logic

Bitwise logic follows the rules given above, but works on the individual bits of the numbers. Thus 1 & 4 = 0 because binary one and binary 4 have no bits in common.

- & bitwise AND
- | bitwise OR
- << shift left. The stored value sets how many positions the bits are shifted.
- >> shift right.

Max does not have a bitwise NOT function except in expr ([expr ~$i1]). There is a Bitnot~ function for processing audio signals.

The most useful bit operation is the AND. [& 127] will clip all values to 7 bits, which is often needed for MIDI messages. This use of 7 is called masking. You can get a sense of how a number can be masked by looking at it in binary. 127 in binary is 01111111, so the lower 7 bits of another number would survive the AND operation. But 128 & 127 = 0.

Bitwise OR combines numbers in a strange way. The result has all bits set that were 1s in either input. 128 (10000000) | 127 (01111111) = 255 (11111111).

Shifting left is equivalent to multiplying by a power of 2. Shifting 1 bit is X2, shifting 2 bits is X4 and so on. Shifting right is like dividing by powers of two. You aren't going to get any float answers, though.

Hardware Logic

The logic chips used in circuit design follow the rules of bitwise logic, but feature multiple inputs, and a change at any input will change the output state. The Llogic object takes an argument to determine its function and another to set the number of inputs. Some operations just seem clearer with a hardware approach.

The Operations in jit.op

The tricky part of using the operators in jit.op is making sense of the pictures you get. Jit.op works on each plane of data independently, so it is easy to scramble the colors. Here is slightly more information that is found in the help file.

Arithmetic

pass: pass left input through with no change.
* : multiplication; this will dim the image.
/ : division; This will usually make the image nearly black.
+ : addition; Anything white in either image will be white.
- : subtraction; Makes dark images
+m : addition modulo; Modulus is 255. 200 +m 200 = 145. Rainbow effects
-m : subtraction modulo; Modulus is 255. 100 -m 200 = 155.
% : modulo; The right is subtracted from the left until the result is less than the right.
Min : minimum;

Max : maximum;
Abs : absolute value; This will only work in situations where negative numbers are in use.
Avg : average of left and right;
Absdiff : absolute value of difference between left and right;
!pass : pass right input, with no change;
!/ : right input/left input;
!- : right input-left input;
!% : right input/left input;
ignore: leave previous output value; this is a freeze function;
fold : mirrored modulo (float32/float64 only); left subtracted from right until the result is between 0 and right.
wrap : positive modulo (float32/float64 only); right subtracted from left until the result is between 0 and right.

Bitwise logic

&: bitwise and; darker
|: bitwise or; lighter
^: bitwise xor; The left goes negative where the right is white.
~: bitwise compliment(unary); This makes a negative of the left.
>>: right shift; Mostly black
<<: left shift; Sparkley

Logic

These result in 0 or 1 for the pixel. You get black, R G or B.
&&: logical and;
||: logical or;
!: logical not(unary);
>: greater than;
<: less than;
>=: greater than or equal to;
<=: less than or equal to;
==: equal;
!=: not equal

Logic switching

These perform the logic and pass the left value if it's true. These are still plane by plane though. You usually get R G or B pixels. These work best with a black and white right input.
>p: greater than(pass);
<p: less than(pass);
>=p: greater than or equal to(pass);
<=p: less than or equal to(pass);
==p: equal(pass);
!=p not equal(pass)

Trig Functions

These only work on floats, not on images.

sin: sine;
cos: cosine;
tan: tangent;
asin: arcsine;
acos: arccosine;
atan: arctangent;
atan2: arctangent(binary);
sinh: hyperbolic sine;
cosh: hyperbolic cosine;
tanh: hyperbolic tangent;
asinh: hyperbolic arcsine;
acosh: hyperbolic arccosine;
atanh: hyperbolic arctangent;

More Float Math

exp: e to the x;
exp2: 2 to the x;
ln: log base e;
log2: log base 2;
log10: log base 10;
hypot: hypotenuse(binary);
pow: x to the y(binary);
sqrt: square root;
ceil: integer ceiling;
floor: integer floor;
round: round to nearest integer;
trunc: truncate to integer;

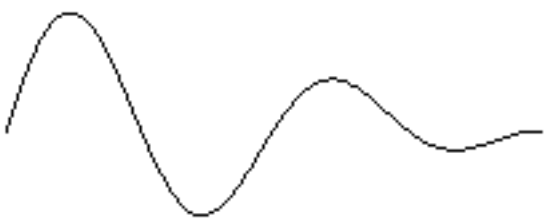
Complex Functions in Jitter

When we need to access a function repeatedly, is often best to calculate the function over the range of expected inputs and store the results in a matrix. Then the function can be computed as the patch is loaded and won't affect performance. Here is an example:

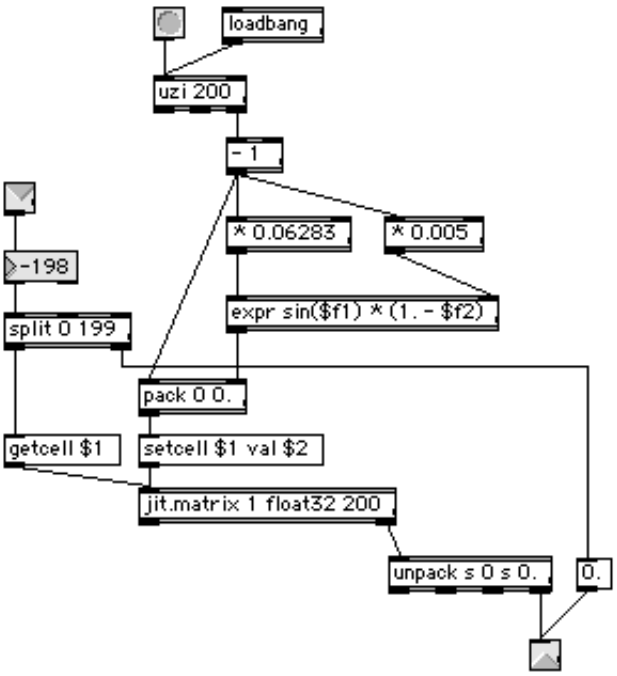
To compute:

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

Which looks like this:



Use a patch like this:



At load time, the uzi runs the numbers 0 to 199 through the math and the results are stuffed into the matrix. The values can be accessed later using the getcell message. This patch is in a subpatcher and appears as a single object in the main patch.

Jit.expr

Jit.expr is an expansion of expr that allows matrices as input. You have all of the jit.op operations (and a couple of useful additions, like the constant PI) but in addition to $f1 and $i1 you have tokens that pull values out off the matrix. Note that you have to explicitly say how many inputs jit.expr should have with the @inputs attribute. You also have to state the expression as an expr attribute with quotes around it.
The expression needs a matrix as input-- this sets the size of the output matrix. (The input matrix need not actually be used in the calculation.)

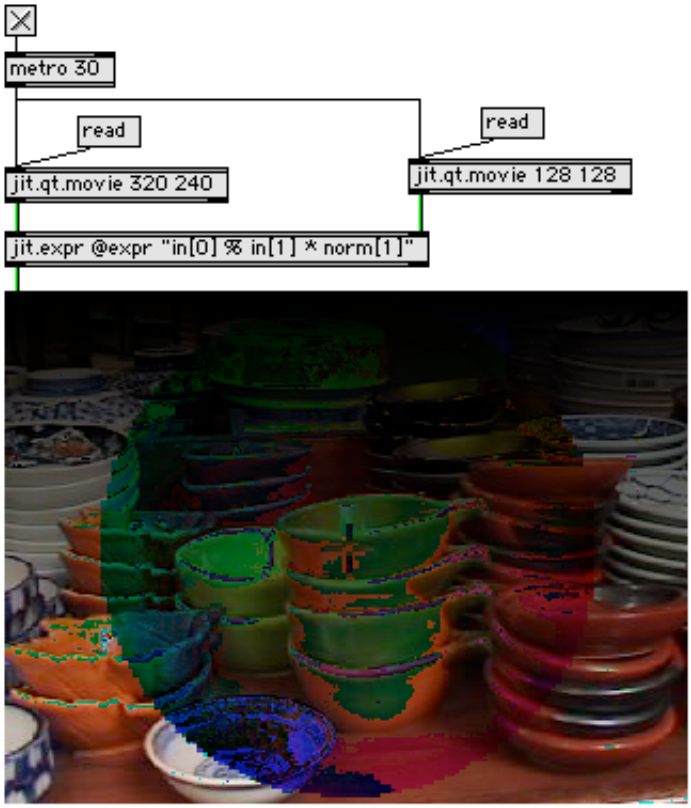
These are the tokens:

in[n]	this processes the whole matrix cell by cell. n of 0 means the left inlet.
in[n].p[m]	This refers to one plane of a matrix
cell[n]	This gives the integer address of the cell, where n is the dimension index. Cell[0] is the horizontal address, cell[1] is the vertical.
norm[n]	this gives a normalized address. norm[0] would have the value 0 at the left, 1.0 at the right.

snorm[n]	A normalized address between -1.0 and 1.0. This puts 0 in the middle.
Matrixname	The name of a matrix, which is similar to connecting it to a right inlet. This should match the input in size and type.
Matrixname.p[n]	A particular plane of a named matrix.

These are useful constants.
PI
TWOPI
HALFPI
INVPI 1/PI
DEGTORAD TWOPI/360
RADTODEG 360/TWOPI
E Euler's constant
LN2 natural log of 2
LN10 natural log of 10
LOG2E log base 2 of Euler's constant
LOG10E log base 10 of Euler's constant
SQRT2 square root of 2
SQRT1_2 square root of 1/2

Here's an example of jit.expr in action:



This is like jit.op %, but multiplying the result times norm[1] gives the top to bottom gradient.