Max and Math

We don't all have a strong background in math, so here's a review of the math operations available in Max. This isn't how to do math, just a note of where things are.

Basics

Number Types

Remember, math done with floats can give different answers than math done with integers. In most cases you put an object into float mode by entering a float as the argument. Also, remember that certain values will be displayed strangely when they are floats. 16.2 becomes 16.200001, for instance. When floats are converted to ints, the fractional part is truncated off.

Addition

The [+] function adds whatever has been received in the right to whatever comes in the left. In the following notes I'm going to call the right input (or written argument) the stored value, and the left input the input value.

Subtraction

The [-] function subtracts the stored value from the input. When you write an argument, be certain there is a space between the minus and the number. If you leave the space out, you won't get an error, you will get an integer object (exactly like[int -1]) and input will be passed through unchanged.

Complement

The [!-] function subtracts the input from the stored value. This is often called taking the "2s compliment" (if subtracting from 2). The 12s compliment of an interval gives the interval's inversion. !- is not a standard notation, only Max refers to it this way. The Lcomp object does the same on lists.

Multiplication

The [*] function multiplies the input by the stored value.

Division

The [/] function divides the input by the stored value. Dividing integers gives truncated results, with any fractional part discarded. If you divide by 0, an error message appears in the Max window. It's always faster to multiply than divide, so [* 0.5] is preferred to [/ 2], at least when the computer is working hard.

Inversion

The [!/] function divides the stored value by the input. Linvert does this with lists.

Modulo
The [%] function divides the input by the stored value and returns the remainder. This works only on integers. Lrem does this on lists.

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 anything else = 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.

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.

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.

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 cosine is the one that starts on 0.

Hyperbolic versions are available as sinh, asinh and so on.

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

[expr $f1 + $f2 * $f3]

will return the input plus the product of stored values 2 and 3. (The brackets represent the object box -- they aren't what you type!) 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. Many C functions take two arguments, such as pow(a,b). Since Max is skittish about commas, you have to write this as

[expr pow($f1\,$f2)]

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

Expr is supposed to be able to do some operations with tables, but frankly, it's easier to get what you want from the table and apply it to an expr inlet.