SECTION 10.6

# Working with Numbers

## Arithmetic Objects

Objects that operate on ordinary numbers to provide basic maths functions are summarised in figure 10.24. All have hot left and cold right inlets and all take

| Object | Function |
|---|---|
| + | Add two floating point numbers |
| - | Subtract number on right inlet from number on left inlet |
| / | Divide lefthand number by number on right inlet |
| * | Multiply two floating point numbers |
| div | Integer divide, how many times the number on the right inlet divides exactly into the number on the left inlet |
| mod | Modulo, the smallest remainder of dividing the left number into any integer multiple of the right number |

**Figure 10.24**
Table of message arithmetic operators.

| Object | Function |
|---|---|
| cos | The cosine of a number given in radians. Domain: $-\pi/2$ `to` $+\pi/2$. Range: $-1.0$ `to` $+1.0$. |
| sin | The sine of a number in radians, domain $-\pi/2$ `to` $+\pi/2$, range $-1.0$ `to` $+1.0$ |
| tan | Tangent of number given in radians. Range: 0.0 `to` $\infty$ `at` $\pm\pi/2$ |
| atan | Arctangent of any number in domain $\pm\infty$ Range: $\pm\pi/2$ |
| atan2 | Arctangent of the quotient of two numbers in Cartesian plane. Domain: any floats representing X, Y pair. Range: angle in radians $\pm\pi$ |
| exp | Exponential function $e^x$ for any number. Range 0.0 `to` $\infty$ |
| log | Natural log (base $e$) of any number. Domain: 0.0 `to` $\infty$. Range: $\pm\infty$ ($-\infty$ `is` $-1000.0$) |
| abs | Absolute value of any number. Domain $\pm\infty$. Range 0.0 `to` $\infty$ |
| sqrt | The square root of any positive number. Domain 0.0 `to` $\infty$ |
| pow | Exponentiate the left inlet to the power of the right inlet. Domain: positive left values only. |

**Figure 10.25**
Table of message trigonometric and higher math operators.

one argument that initialises the value otherwise received on the right inlet. Note the difference between arithmetic division with / and the div object. The modulo operator gives the remainder of dividing the left number by the right.

## Trigonometric Maths Objects

A summary of higher maths functions is given in figure 10.25.

## Random Numbers

A useful ability is to make random numbers. The random object gives integers over the range given by its argument including zero, so random 10 gives 10 possible values from 0 to 9.

## Arithmetic Example

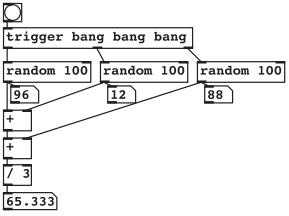


**Figure 10.26**
Mean of three random floats.

An example is given in figure 10.26 to show correct ordering in a patch to calculate the mean of three random numbers. We don't have to make every inlet hot, just ensure that everything arrives in the correct sequence by triggering the random objects properly. The first random (on the right) supplies the cold inlet of the lower +, the middle one to the cold inlet of the upper +. When the final (left) random is generated it passes to the hot inlet of the first +, which computes the sum and passes it to the second + hot inlet. Finally we divide by 3 to get the mean value.

## Comparative Objects

In figure 10.27 you can see a summary of comparative objects. Output is either 1 or 0 depending on whether the comparison is true or false. All have hot left inlets and cold right inlets and can take an argument to initialise the righthand value.

## Boolean Logical Objects

There are a whole bunch of logical objects in Pd including bitwise operations that work exactly like C code. Most of them aren't of much interest to us in this book, but we will mention the two important ones, || and &&. The output of ||, logical OR, is true if either of its inputs are true. The output of &&, logical AND, is true only when both its inputs are true. In Pd any non-zero number is "true," so the logical inverter or "not" function is unnecessary because there are many ways of achieving this using other objects. For example, you can make a logical inverter by using != with 1 as its argument.

| Object | Function |
|---|---|
| > | True if the number at the left inlet is greater than the right inlet. |
| < | True if the number at the left inlet is less than the right inlet. |
| >= | True if the number at the left inlet is greater than or equal to the right inlet. |
| <= | True if the number at the left inlet is less than or equal to the right inlet. |
| == | True if the number at the left inlet is equal to the right inlet. |
| != | True if the number at the left inlet is not equal to the right inlet. |

**Figure 10.27**
List of comparative operators.

SECTION 10.7

# Common Idioms

There are design patterns that crop up frequently in all types of programming. Later we will look at abstraction and how to encapsulate code into new objects so you don't find yourself writing the same thing again and again. Here I will introduce a few very common patterns.

## Constrained Counting

We have already seen how to make a counter by repeatedly incrementing the value stored in a float box. To turn an increasing or decreasing counter into a cycle for repeated sequences there is an easier way than resetting the counter when it matches an upper limit: we wrap the numbers using mod. By inserting mod into the feedback path before the increment we can ensure the counter stays bounded. Further mod units can be added to the number stream to generate polyrhythmic sequences. You will frequently see variations on the idiom shown in figure 10.28. This is the way we produce multirate timebases for musical sequencers, rolling objects, or machine sounds that have complex repetitive patterns.

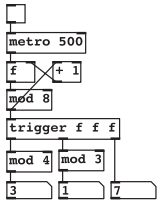


**Figure 10.28**
Constrained counter.

## Accumulator

A similar construct to a counter is the accumulator or integrator. This reverses the positions of f and + to create an integrator that stores the sum of all previous number messages sent to it. Such an arrangement is useful for turning


**Figure 10.29**
Accumulator.

"up and down" messages from an input controller into a position. Whether to use a counter or accumulator is a subtle choice. Although you can change the increment step of the counter by placing a new value on the right inlet of + it will not take effect until the previous value in f has been used. An accumulator, on the other hand, can be made to jump different intervals immediately by the value sent to it. Note this important difference: an accumulator takes floats as an input while a counter takes bang messages.

## Rounding

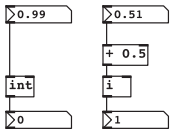


**Figure 10.30**
Rounding.

An integer function, int, also abbreviated i, gives the whole part of a floating point number. This is a *truncation*, which just throws away any decimal digits. For positive numbers it gives the *floor* function, written $\lfloor x \rfloor$, which is the integer *less than or equal to* the input value. But take note of what happens for *negative* values, applying int to $-3.4$ will give $-3.0$, an integer *greater than or equal to* the input. Truncation is shown on the left in figure 10.30. To get a regular rounding for positive numbers, to pick the *closest* integer, use the method shown on the right in figure 10.30. This will return 1 for an input of 0.5 or more and 0 for an input of 0.49999999 or less.

## Scaling

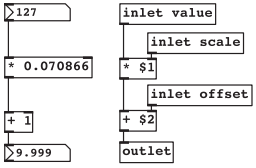


**Figure 10.31**
Scaling.

This is such a common idiom you will see it almost everywhere. Given some input values such as 0 to 127, we may wish to map them to another set of values, such as 1 to 10. This is the same as changing the slope and zero intersect of a line following $y = mx + c$. To work out the values you first obtain the bottom value or *offset*, in this case +1. Then a *multiplier* value is needed to scale for the upper value, which given an input of 127 would satisfy $10 = 1 + 127x$, so moving the offset we get $9 = 127x$, and dividing by 127 we get $x = 9/127$ or $x = 0.070866$. You can make a subpatch or an abstraction for this as shown in figure 13.1, but since only two objects are used it's more sensible to do scaling and offset as you need it.

## Looping with Until

Unfortunately, because it must be designed this way, until has the potential to cause a complete system lockup. Be very careful to understand what you are doing with this. A bang message on the left inlet of until will set it producing bang messages as fast as the system can handle! These do not stop *until* a bang message is received on the right inlet. Its purpose is to behave as a fast loop construct performing message domain computation quickly. This way you can fill an entire wavetable or calculate a complex formula in the time it takes to

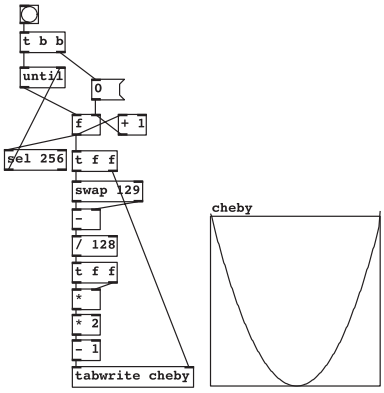


**Figure 10.32**
Using until.

process a single audio block. Always make sure the right inlet is connected to a valid terminating condition. In figure 10.32 you can see an example that computes the second Chebyshev polynomial according to $y = 2x^2 - 1$ for the range $-1.0$ to $+1.0$ and fills a 256-step table with the result. As soon as the bang button is pressed a counter is reset to zero, and then until begins sending out bangs. These cause the counter to rapidly increment until select matches 256, whereupon a bang is sent to the right inlet of until, stopping the process. All this will happen in a fraction of a millisecond. Meanwhile we use the counter output to calculate a Chebyshev curve and put it into the table.


**Figure 10.33**
For 256.

A safer way to use until is shown in figure 10.33. If you know in advance that you want to perform a fixed number of operations, then use it like a **for loop**. In this case you pass a non-zero float to the left inlet. There is no terminating condition; it stops when the specified number of bangs has been sent—256 bangs in the example shown.

## Message Complement and Inverse

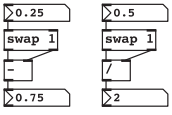


**Figure 10.34**
Message reciprocal and inverse.

Here is how we obtain the number that is $1 - x$ for any $x$. The *complement* of $x$ is useful when you want to balance two numbers so they add up to a constant value, such as in panning. The swap object exchanges its inlet values, or any left inlet value with its first argument. Therefore, what happens with the lefthand example of figure 10.34 is the - calculates $1 - x$, which for an input of 0.25 gives 0.75. Similarly, the inverse of a float message $1/x$ can be calculated by replacing the - with a /.

## Random Selection

To choose one of several events at random, a combination of random and select will generate a bang message on the select outlet corresponding to one of its arguments. With an initial argument of 4 random produces a *range* of 4 random integer numbers starting at 0, so we use select 0 1 2 3 to select amongst them. Each has an equal probability, so every outlet will be triggered 25% of the time on average.

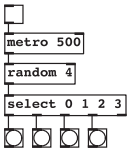


**Figure 10.35**
Random select.

## Weighted Random Selection

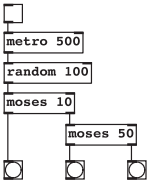


**Figure 10.36**
Weighted random select.

A simple way to get a bunch of events with a certain probability distribution is to generate uniformly distributed numbers and stream them with moses. For example, moses 10 sends integers greater than 9.0 to its right outlet. A cascade of moses objects will distribute them in a ratio over the combined outlets when the sum of all ratios equals the range of random numbers. The outlets of moses 10 distribute the numbers in the ratio 1 : 9. When the right outlet is further split by moses 50 as in figure 10.36, numbers in the range 0.0 to 100.0 are split in the ratio 10 : 40 : 50, and since the distribution of input numbers is uniform they are sent to one of three outlets with 10%, 40%, and 50% probability.

## Delay Cascade

Sometimes we want a quick succession of bangs in a certain fixed timing pattern. An easy way to do this is to cascade delay objects. Each delay 100 in figure 10.37 adds a delay of 100 milliseconds. Notice the abbreviated form of the object name is used.

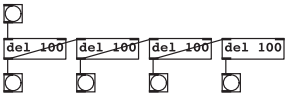


**Figure 10.37**
Delay cascade.

## Last Float and Averages

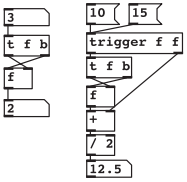


**Figure 10.38**
Last value and averaging.

If you have a stream of float values and want to keep the previous value to compare to the current one then the idiom shown on the left in figure 10.38 will do the job. Notice how a trigger is employed to first bang the *last* value stored in the float box and then update it with the current value via the right inlet. This can be turned into a simple "low pass" or averaging filter for float messages as shown on the right in figure 10.38. If you add the previous value to the current one and divide by two you obtain the average. In the example shown the values were 10 followed by 15, resulting in $(10 + 15)/2 = 12.5$.

## Running Maximum (or Minimum)

Giving max a very small argument and connecting whatever passes through it back to its right inlet gives us a way to keep track of the largest value. In figure 10.39 the greatest past value in the stream has been 35. Giving a very large argument to min provides the opposite behaviour for tracking a lowest value. If you need to reset the maximum or minimum tracker just send a very large or small float value to the cold inlet to start again.

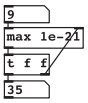


**Figure 10.39**
Biggest so far.

## Float Low Pass

Using only * and + as shown in figure 10.40 we can low pass filter a stream of float values. This is useful to smooth data from an external controller where values are occasionally anomalous. It follows the filter equation $y_n = Ax_n + Bx_{n-1}$. The strength of the filter is set by the ratio $A : B$. Both $A$ and $B$ should be between 0.0 and 1.0 and add up to 1.0. Note that this method will not converge on the exact input value, so you might like to follow it with int if you need numbers rounded to integer values.

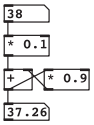


**Figure 10.40**
Low pass for floats.