Gen

Gen refers to a technology in Max representing a new approach to the relationship between patchers and code. The patcher is the traditional Max environment -- a graphical interface for linking bits of functionality together. With embedded scripting such as the `js` object text-based coding became an important part of working with Max as it was no longer confined to simply writing Max externals in C. Scripting however still didn't alter the logic of the Max patcher in any fundamental way because the boundary between patcher and code was still the object box. Gen represents a fundamental change in that relationship.

The Gen patcher is a new kind of Max patcher where Gen technology is accessed. Gen patchers are specialized for specific domains such as audio (MSP) and matrix and texture processing (Jitter). The MSP Gen object is called `gen~`. The Jitter Gen objects are `jit.gen`, `jit.pix` and `jit.gl.pix`. Each of these Gen objects contains within it a Gen patcher. While gen patchers share many of the same capabilities, each Gen object has functionality specific to its domain. For example, Gen patchers in `gen~` have delay lines while Gen patchers in `jit.gen` have vector operations.

Gen patchers describe the calculations a Gen object performs. When you're editing a Gen patcher, you're editing the internal calculations of the Gen object. In order do make use of the computations described in its Gen patcher, a Gen object compiles the patcher into a language called *GenExpr*. GenExpr bridges the patcher and code worlds with a common representation, which a Gen object turns into code necessary to perform its calculations. `gen~`, `jit.gen`, and `jit.pix` transparently generate and compile native CPU machine code on-the-fly, while `jit.gl.pix` does the same for GPU code (GLSL). When working with Gen objects, you're writing your own custom pre-compiled MSP and Jitter objects without having to leave Max.

Gen Patching

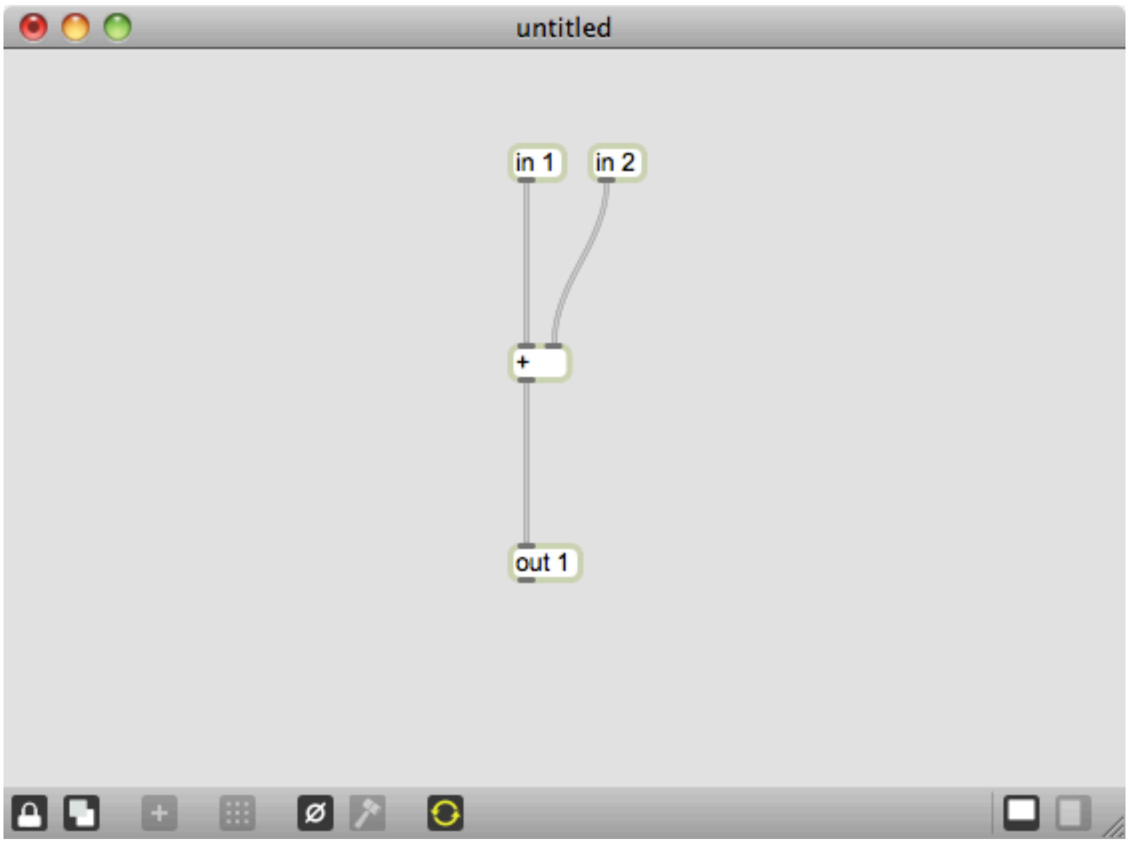
Gen patchers look similar to Max patchers, but there are a few important differences:

- The set of objects (or *"operators"*) available in a Gen patcher (and also the GenExpr language) are different; they are described later in this documentation.
- There are no messages. All operations are *synchronous*, rather like MSP patching. Because of this, there are no UI objects (sliders, buttons etc.). However *`param`* operators can be used to receive message-rate controls from the normal Max world. There is no need to differentiate hot and cold inlets, or the order in which outlets 'fire', since outlets always fire at the same time.
- There are no `send` and `receive` operators in Gen patcher. Gen patchers are connected to the outside world through the `in`, `out`, and `param` operators. In `gen~` there are some additional operators that are controllable with messages to `gen~`. See

the gen~ section for the details.

- The usual distinction between **int** and **float** numbers does not apply to Gen patchers. At the Gen patcher level, everything is a **float**.
- The codebox is a special operator for Gen patchers, in which more complex expressions can be written using the GenExpr language.

Gen patchers can be embedded within the gen~, jit.gen, etc. object, or can be loaded from external files (with .gendsp or .genjit file extensions respectively) using the @gen attribute.



Auto-Compile

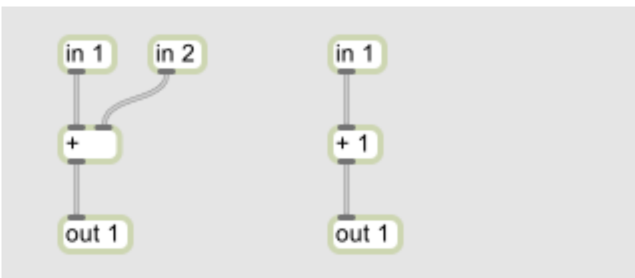
By default, compilation process occurs in the background while you are editing, so that you can see or hear the results immediately. This auto-compilation process can be disabled using the 'Auto-Compile' toggle in the Gen patcher toolbar. Compilation can also be triggered using the hammer icon in the Gen patcher toolbar or any codebox toolbar.

Gen Operators

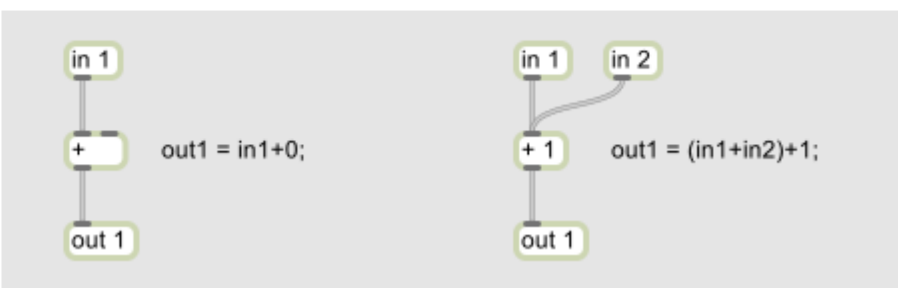
Gen operators represent the functionality involved in a Gen patcher. They can exist as object boxes in a patcher or as functions or variables in GenExpr code. They are the link between the patcher and code worlds.

Gen operators take arguments and attributes just like Max objects, but these are purely declarative. Since there is no messaging in Gen patchers, the attribute value set when the operator is created does not change. Attributes are most often used to specialize the implementation of the process the operator represents.

In many cases, the specification of an object's argument effectively replaces the corresponding inlet. This is possible in Gen because there is no messaging and all processing is synchronous. For example, the operator + takes two inputs, but if an argument is given only one input needs to be specified as an inlet:



An inlet with no connected patchcord uses a default value instead (often zero, but check the inlet assist strings for each operator). An inlet with multiple connections adds them all together, just as with MSP signal patchcords:



Standard Operators

Many standard objects behave like the corresponding Max or MSP object, such as all arithmetic operators (and the reverse operators like `!-`, `!/` etc.), trigonometric operators (`sin`, `cosh`, `atan2` etc.), standard math operators (`abs`, `floor`, `pow`, `log`, etc.), boolean operators (`>`, `==`, `&&` (also known as `and`) etc.) and other operators such as `min`, `max`, `clip` (also known as `clamp`), `scale`, `fold`, `wrap`, `cartopol`, `poltocar` etc. In addition there are some operators in common with GLSL (`fract, mix, smoothstep, degrees, radians` etc.) and some drawn from the jit.op operator list (`>p, ==p, absdiff` etc.).

There are several predefined constants available (`pi, twopi, halfpi, invpi, degtorad, radtodeg, e, ln2, ln10, log10e, log2e, sqrt2, sqrt1_2`, and the same in capitalized form as `PI, TWOPI` etc), which can be used in place of a numeric argument to any operator:

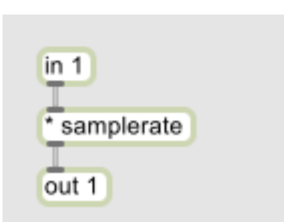
Gen~

The `gen~` object is specifically for operating on MSP audio signals. Unlike MSP patching however, each operation in a Gen patcher is applied per-sample. This makes possibly many more optimizations to make complex processes more efficient. It also allows you to design processes which must operate on a per-sample level, even with feedback loops. Because of this, many operators take duration arguments in terms of samples (where many equivalent MSP objects would use milliseconds).

`gen~` Operators

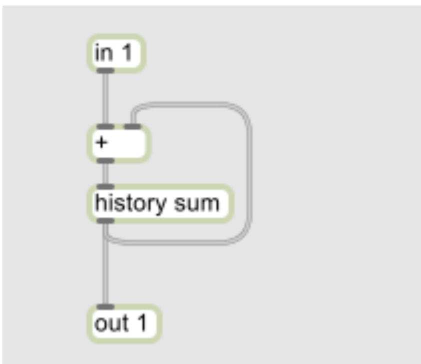
In addition to the standard Gen operators, which are often similar to the equivalent MSP objects (such as `clip, scale, minimum, maximum`, etc.), many of the operators specific to the `gen~` domain mirror existing MSP objects to make the transition to `gen~` easier. There are familiar converters (`dbtoa, atodb, mtof, ftom, mstosamps, sampstoms`), oscillators (`phasor, train, cycle, noise`), and modifiers (`delta, change, sah, triangle, phasewrap, pong`). In addition there are some lower-level operators to avoid invalid or inaudible outputs (`isnan, fixnan, isdenorm, fixdenorm, dcblock`).

A global value of `samplerate` is available both as an object, and as a valid value for an argument of any object.



History

In general, the Gen patcher will not allow a feedback loop (since it represents a synchronous process). To create a feedback loop in gen~, the `history` operator can be used. This represents a single-sample delay (a Z^{-1} operation). Thus the inlet to the `history` operator will set the outlet value for the next sample (put another way, the outlet value of the `history` operator is the inlet value from the previous sample). Multiple history operators can be chained to create Z^{-2} , Z^{-3} delays, but for longer and more flexible delay operators, use the `delay` operator.



A `history` operator can also be named, making it available for external control, just like a `param` parameter.

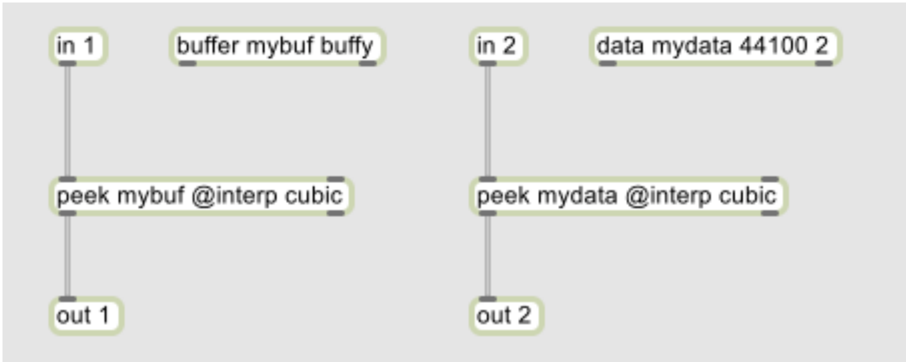
Delay

The `delay` operator delays a signal by a certain amount of time, specified in samples. The maximum delay time is specified as an argument to the delay object. You can also have a multi-tap delay by specifying the number of taps in the second argument. Each tap will have an inlet to set the delay time, and a corresponding outlet for the delayed signal.

An @interp attribute can choose between step (none), linear, cosine, cubic or spline interpolation. The `delay` operator can be used for feedback loops, like the `history` operator, if the @feedback attribute is set to 1.

Data and Buffer

For more complex persistent storage of audio (or any numeric) data, gen~ offers two objects: `data` and `buffer`, which are in some ways similar to MSP’s `buffer~` object. A `data` or `buffer` object has a local name, which is used by various operators in the Gen patcher to read and write the `data` or `buffer` contents, or get its properties.



Reading the contents of a `data` or `buffer` can be done using the `peek`, `lookup`, `wave`, `sample` or `nearest` operators, whose first argument is the local name of a `data` or `buffer`. They all support single- or multi-channel reading (the second argument specifies the number of channels, and the last inlet the channel offset, where zero is the default).

All of these operators are essentially the same, differing only in defaults of their attributes. The attributes are:

- `@index` specifies the meaning of the first inlet:
  - `samples`: (the first inlet is a sample index into the `data` or `buffer`
  - `phase`: maps the range 0..1 to the whole `data` or `buffer` contents
  - `lookup` or `signal`: maps the range -1..1 to the whole `data` or `buffer` contents, like MSP's `lookup~`
  - `wave`: adds extra inlets for start/end (in samples), driven by a phase signal between these boundaries (0..1, similar to MSP's `wave~`)
- `@boundmode` specifies what to do if the index is out of range:
  - `ignore`: indices out of bounds are ignored (return zero)
  - `wrap`: indices out of bounds repeat at opposite boundary
  - `fold` or `mirror`: indices wrap with palindrome behavior
  - `clip` or `clamp`: indices out of bounds use value at bound
- `@channelmode` specifies what to do if the channel is out of range; has the same options as `@boundmode`
- `@interp` specifies what kind of interpolation is used:
  - `none` or `step`: no interpolation
  - `linear`: linear interpolation
  - `cosine`: cosine interpolation
  - `cubic`: cubic interpolation
  - `spline`: Catmull-Rom spline interpolation

The `nearest` operator defaults to `@index phase @interp none @boundmode ignore @channelmode ignore`.

The `sample` operator defaults to `@index phase @interp linear @boundmode` `ignore @channelmode ignore.`

The `peek` operator defaults to `@index samples @interp none @boundmode ignore @channelmode ignore.`

The `lookup` operator defaults to `@index lookup @interp linear @boundmode clamp @channelmode clamp.`

The `wave` operator defaults to `@index wave @interp linear @boundmode wrap @channelmode clamp.`

Accessing the spatial properties of a `data` or `buffer` is done using the `dim` and `channels` operators (or the outlets of the `data` or `buffer` object itself), and writing is done using `poke` (non-interpolating replace) or `splat` (interpolating overdub).

Briefly, `data` should be thought of as a 64bit buffer internal to the `gen~` patcher, even though it can be copied to, and `buffer` should be thought of as an object which can read and write external `buffer~` data. The full differences between `data` and `buffer` are:

1. A `data` object is local to the Gen patcher, and cannot be read outside of it. On the other hand, a `buffer` object is a *shared reference* to an external MSP `buffer~` object. Modifying the contents in a Gen `buffer` is directly modifying the MSP `buffer~` it references.
2. The `data` object takes three arguments to set its local name, its length (in samples) and number of channels. The `buffer` object takes an argument to set its local name, and an optional argument to specify the name of a MSP `buffer~` to references (instead of using the local name).
3. The `data` object cannot be resized; but the `buffer` object always has the size of the `buffer~` it references (which may change).
4. Setting the `gen~` attribute corresponding to a named `data` object copies in values from the corresponding MSP `buffer~`, while for a named `buffer` object it changes the MSP `buffer~` referenced. As such, the `data` size/channels counts are unchanged (extra samples and channels are ignored), but the `buffer` data/size channels counts change according to the referenced `buffer~`.
5. The `data` object always uses 64-bit doubles, while the `buffer` object converts from the bit resolution of the MSP `buffer~` object (currently 32-bit floats) for all read and write operations, and may be less efficient.

Technical notes

All operations in `gen~` use 64-bit doubles, with the exception of reading/writing `buffer` contents.

The compilation process for `gen~` Gen patchers and GenExprs includes an optimization that takes into account the update rate of each operator, so that any calculations that do not need to occur at sample-rate (such as arithmetic on the outputs of `param` operators) instead process at a slower ‘control-rate’ for efficiency.

Jitter Gen Objects

There are three Gen objects in jitter: `jit.gen`, `jit.pix` and `jit.gl.pix`. `jit.gen` and `jit.pix` process Jitter matrices similar to `jit.expr`. `jit.gl.pix` processes textures and matrices just like `jit.gl.slab`. `jit.gen` is a generic matrix processing object that can handle matrices with any planecount, type and dimension. `jit.pix` and `jit.gl.pix` on the other hand are specifically designed for working with pixel data. They can handle data of any type, but it must be two dimensional or less and have at most four planes.

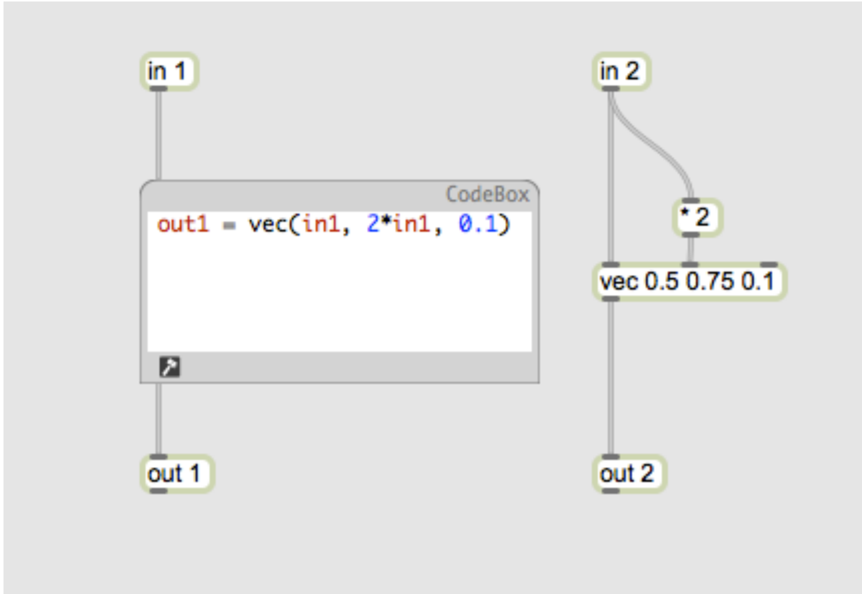
Jitter Operations

Coordinates

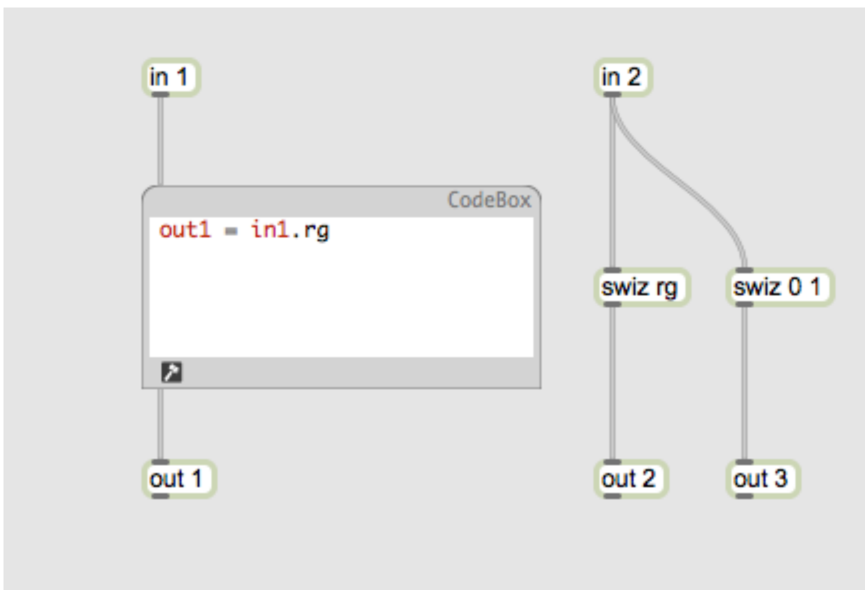
Jitter Gen patchers describe the processing kernel for each cell in a matrix or texture. As the kernel is processing the input matrices, a set of coordinates is generated describing the location of the current cell being processing. The objects are just like the operators in `jit.expr`. They are `norm`, `snorm`, and `cell` with the `dim` operator giving the dimensions of the input matrix. `norm` ranges from [0, 1] across all matrix dimensions and is defined as **`norm = cell/dim`**. `snorm` ranges from [-1, 1] across all matrix dimensions and is defined as **`snorm = cell/dim*2-1`**. `cell` gives the current cell index.

Vectors

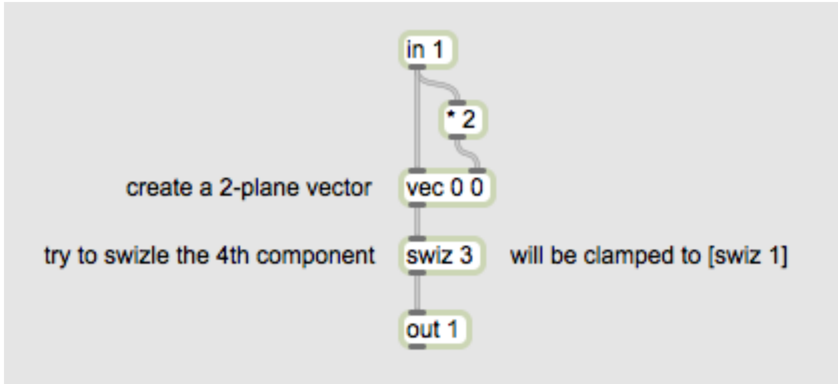
Since Jitter matrices represent arrays of vector (more than one plane) data, all Gen operators in Jitter can process vectors of any size, so Gen patchers once created work equally on any vector size. The basic binary operators `+`, `-`, `*`, `/`, and `%` can take vector arguments as in **[+ 0.5 0.25 0.15]**, which will create an addition operator adding a vector with the three components to its input.



The vec operator creates vector constants and packs values together in a vector. It takes default arguments for its components and casts all of its inputs to scalar values before packing them together.



The swiz operator applies a swizzle operation to vectors. In GLSL and similar shading languages, vector components can be accessed by indexing the vector with named planes. For example in GLSL you might see **red = color.r** or **redalpha = color.ra** or even **val = color.rbbg**. This type of operation is referred to as swizzling. The swiz operator can take named arguments using the letters **r**, **g**, **b**, **a**, as well as **x**, **y**, **z**, and **w** in addition to numeric indices starting at 0. The letters are convenient for vectors with four or less planes, but for larger vectors numeric indices must be used. The compilation process automatically checks any `swiz` operation so arguments indexing components larger than the vector being processed will be clamped to the size of the vector.

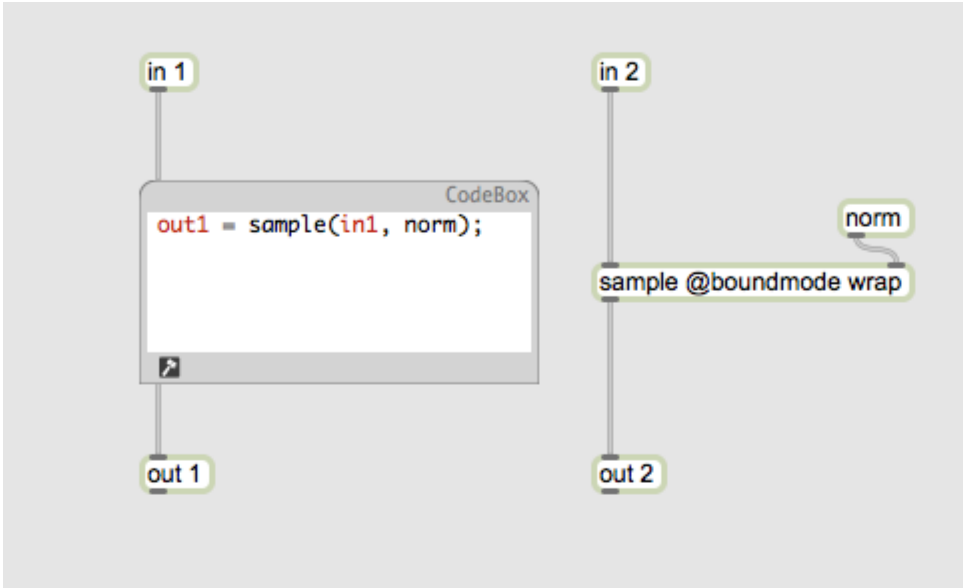


In addition, there are the basic vector operations for spatial calculations. These are `length`, `normalize`, `cross`, `dot`, and `reflect`.

Sampling

Sampling operators are one of the most powerful features of Jitter Gen patchers. Sampling operators take an input and a coordinate in the range [0, 1] as an argument, returning the data at the coordinate's position in the matrix or texture. The first argument always has to be a Gen patcher input while the second argument is an N-dimensional vector whose size is equal to the dimensionality of the input it is processing.

If the coordinate argument is outside of the range [0, 1], it will be converted to a value within the range [0, 1] according to its boundmode function. Possible boundmodes are **wrap**, **mirror**, and **clamp** where **wrap** is the default.



The two sampling operators in Jitter Gen patchers are `sample` and `nearest`. `sample` samples values form a matrix using N-dimensional linear interpolation. `nearest` will simply grab the value from the closest cell.

Geometry

Jitter Gen patchers include a suite of objects for generating surfaces. These include most of the shapes available in the `jit.gl.gridshape` object. Each surface function returns two values: the vertex position and the vertex normal. The geometry operators are `sphere`, `torus`, `circle`, `plane`, `cone`, and `cylinder`.

Color

There are two color operators in Jitter Gen patchers. They are `rgb2hsl` and `hsl2rgb`. They convert between the Red Green Blue color space and the Hue Saturation Luminance color space. If the input to these objects has an alpha component, the alpha will be passed through untouched.

jit.gen

`jit.gen` is a general purpose matrix processing object. It compiles Gen patchers into C++ code representing the kernel of an N-dimensional matrix processing routine. It follows the Jitter matrix planemapping conventions for pixel data with planes [0-4] as the ARGB channels. `jit.gen` can have any number of inlets and outlets, but the matrix format for the different inputs and outputs is always linked. `jit.gen` makes use of parallel processing just like other parallel aware objects in Jitter for maximum performance with large matrices.

How a matrix is processed by `jit.gen` is dependent on the input planecount, type, and dimension of the input matrices. In addition, there is a **precision** attribute that sets the type of the processing kernel. The default value for **precision** is **auto**. Auto precision automatically adapts the type of the kernel dependent upon the matrix input type. In auto mode, the following mapping between input matrix type and kernel processing type is used:

- **char** maps to **fixed**
- **long** maps to **float64**
- **float32** maps to **float32**
- **float64** maps to **float64**

Other possible values for the **precision** attribute are **fixed**, **float32**, and **float64**. **Fixed** precision is the only setting that doesn't correspond to a Jitter matrix type. **Fixed** precision specifies a kernel type that performs a type of floating point calculation with integers using a technique called *fixed-point arithmetic*. It's very fast and provides more precision than 8-bit char operations without incurring the cost of converting to a true floating-point type. However, fixed-point arithmetic calculations have more error that can sometimes be visible when using the sampling operators. If there are noticeable artifacts, simply increase the internal precision to **float32**.

jit.pix

`jit.pix` is a matrix processing object specifically for pixel data. When processing matrices representing video and images, `jit.pix` is the best object. Internally, data is in RGBA format always. If the input has less than four planes, `jit.pix` will convert it to RGBA format according to the following rules:

- 1-plane, Luminance format, L to LLL1 (Luminance for RGB and 1 for Alpha)
- 2-plane Lumalpha format, LA to LLLA (Luminance for RGB and Alpha for Alpha)
- 3-plane RGB format, RGB to RGB1 (RGB for RGB and 1 for Alpha)
- 4-plane, ARGB format, ARGB to RGBA (changes the order of the channels internall)

The output of `jit.pix` is always a 4-plane matrix in ARGB format, which is the standard Jitter pixel planemapping. Like `jit.gen`, `jit.pix` compiles Gen patchers into C++ and makes use of Jitter's parallel processing system. `jit.pix` also has a precision attribute that operates exactly the same was as it does in `jit.gen`.

jit.gl.pix

`jit.gl.pix` is a matrix and texture processing object specifically for pixel data that operates just like `jit.gl.slab`. The only difference between the two is that `jit.gl.pix` compiles its patcher into GLSL while `jit.gl.slab` reads it from a shader file. Like `jit.pix`, `jit.gl.pix` uses an internal RGBA pixel format.

Technical notes

jit.pix v. jit.gl.pix

For the most part `jit.pix` and `jit.gl.pix` will behave identically despite one being CPU-oriented and the other GPU-oriented. The differences have to do with differences in behavior between how matrix inputs are handled with `jit.pix` and how texture inputs are handled with `jit.gl.pix`. All of the inputs to `jit.pix` will adapt in size, type, and dimension to the left-most input. As a result, all input matrices within a `jit.pix` processing kernel will have the same values for the `cell` and `dim` operators. In `jit.gl.pix`, inputs can have different sizes. In `jit.gl.pix`, the values for the `cell` and `dim` operators are calculated from the properties of the left-most input (**`in1`**). A future version may include per-input `cell` and `dim` operators, but for now this is not the case.

Since the sampling operators take normalized coordinates in the range [0, 1], differently sized input textures will still be properly sampled using the `norm` operator since its value is independent of varying input sizes. However, in `jit.gl.pix` the sample and nearest operators behave differently than with `jit.pix`. How a texture is sampled is determined by the properties of the texture. As a consequence, sample and nearest behave the same in `jit.gl.pix`. To enable nearest sampling, set the `@filter` attribute to `nearest`. For `linear` interpolation, set `@filter` to `linear` (the default).

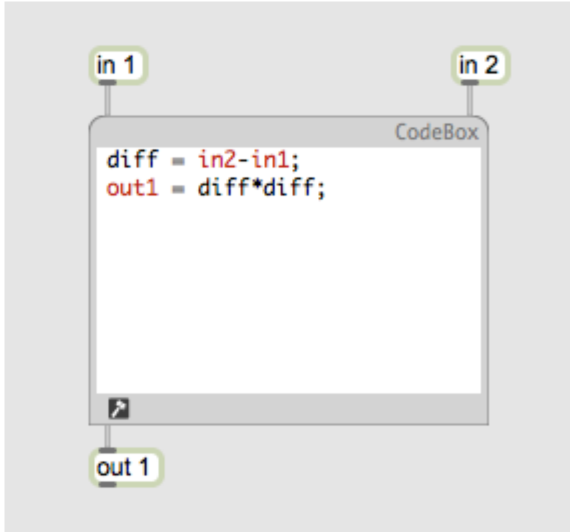
GenExpr

Background

GenExpr is the internal language used by gen patchers. It is used to describe computations in an implementation agnostic manner. To perform actual computations, it is translated into machine code for the CPU or GPU by the various gen objects (`gen~`, `jit.gen`, etc.).

The GenExpr language can be used directly in gen patchers with the `expr` and `codebox` objects. These objects analyze the expressions written in them and automatically construct the the appropriate number of inlets and outlets so that patchcords can be connected to the computations described within.

Note that there is absolutely no difference in terms of performance between describing computations with object boxes and the GenExpr language. When a gen patcher is compiled, it all gets merged into a single representation, so use the approach that is most convenient for the problem.

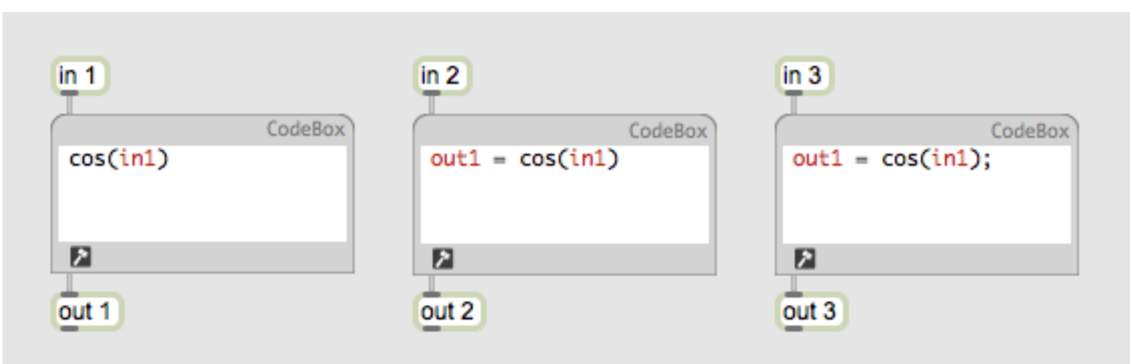


Gen Patcher with a `codebox` object

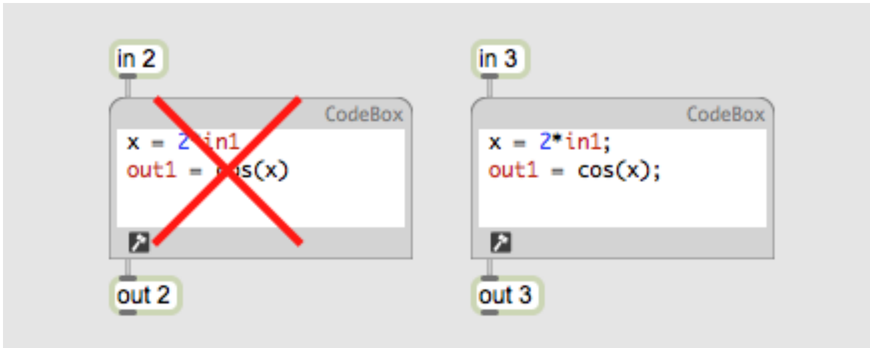
The GenExpr language is designed to complement the Max patching environment within Gen patchers. It provides a parallel textual mechanism for describing computations to be used in concert with the graphical patching paradigm of Max. As one example, the structural elements of user-defined GenExpr functions correspond closely to the structural elements of Max objects with their notions of inlets, outlets, arguments and attributes. Furthermore, the GenExpr language has keywords in, in1, in2, … and out, out1, out2, … that specifically refer to the inlets and outlets of the `expr` or `codebox` the GenExpr code is embedded in.

Language Basics

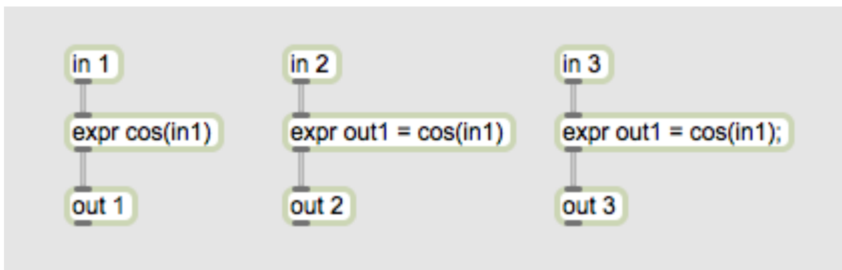
The GenExpr language's syntax resembles that of C and JavaScript for simple expression statements like those in the `codebox` above, however, semicolons are only necessary when there are multiple statements. The `codeboxes` below all contain valid expressions in GenExpr. When there is a single expression with no assignment like in the far left `codebox`, the assignment to **`out1`** is implied. Notice that it also doesn't have a semicolon at the end. When there is only one statement, the semicolon is also implied.



For multi-line statements however, semicolons are required. The `codebox` on the left doesn't have them and will generate errors. The `codebox` on the right is correct.



The `expr` operator is functionally the same as a `codebox` but lacks the text editor features such as syntax highlighting and multi-line text display and navigation.



`expr` is most useful for short, one-line expressions, saving the effort of patching together a sequence of objects together that operate as a unit.

An `expr` or `codebox` determines its number of inlets and outlets by detecting the inN and outN keywords where N is the inlet/outlet position. in1 and out1 are the left-most inlet and outlet respectively. For convenience, the keywords in and out are equivalent to in1 and out1 respectively.

Almost every object that can be created in a Gen patcher is also available from within GenExpr as either a function, a global variable, a declaration, or a constant. The number of inlets an object has corresponds to the number of arguments a function takes. For example, the object `atan2` has two inlets and takes two arguments as follows: **`out = atan2(in1, in2).`**

Comments

Comments in GenExpr follow the C style syntax for both single-line and multi-line comments. Single-line comments start with `//` and multi-line comments are defined by `/*` until the next `*/`.

Multiple Return Values

Just as object boxes can have multiple inlets and outlets, function in GenExpr can take multiple arguments and can return multiple values. The object `cartopol` has two inlets and two outlets. Similarly, in GenExpr the **`cartopol`** function takes two arguments and returns two values. In code, this looks like **`r, theta = cartopol(x, y).`** Functions that return mutiple values can assign to a list of variables. The syntax follows the pattern:

```
var1, var2, var3, … = <expression>
```

When a function returns multiple values but assigns to only one value, the unused return values are simply ignored. When a return value is ignored, the GenExpr compiler eliminates any unnecessary calculations. The function `cartopol` could be expanded out to

```
r, theta = sqrt(x*x, y*y), atan2(y, x
```

If we remove theta and have instead

```
r = sqrt(x*x, y*y), atan2(y, x)
```

the compiler simplifies it to

```
r = sqrt(x*x, y*y)
```

Even for more complex examples where the outputs share intermediate calculations, the compiler eliminates unnecessary operations, so there is no performance penalty for not using all of a function's return values.[1]

Just as the left-hand side list of variable names being assigned to are separated by commas, the right-hand side list of expressions can also be separated by commas:

```
sum, diff = in1+in2, in1-in2
out1, out2 = diff, sum
```

If there are more values on the left-hand side than on the right-hand side, the extra variable names are given a value of zero.

For example,

```
out1, out2 = in1
```

becomes

```
out1, out2 = in1, 0
```

---

[1]The geometry generators such as `cylinder` and `torus` in the Jitter Gen objects are even more complex than `cartopol` since the outputs are interdependent. Still, the GenExpr compiler will eliminate any unused operations specific to unused return values even in these situations.

If any of the expressions in the right-hand side return more than one value, these additional values will be ignored unless the expression is the last item in the right-hand side list. This is complex to describe, but should be clear from these examples:

Unused Return Values

The second return value gets discarded and `cartopol` is optimized:

`r = cartopol(x, y)`

Extra Assignment Values

Zeros are assigned to extra assignment values:

`x, y = in1` becomes **`x, y = in1, 0`**

Multiple Return Values in an Expression List

Only the last expression can return multiple values. `cartopol`'s second return value discarded, as it is not the last expression in the right-hand side:

`r, out1 = cartopol(x, y), in1`

Here `cartopol` returns both values, since it is in the last position:

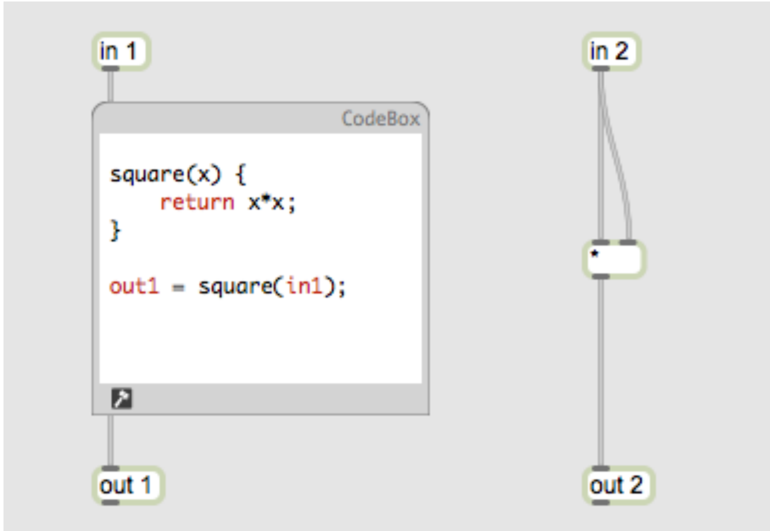
`out1, r, theta = in1, cartopol(x, y)`

The same principle applies when expressions are used as arguments to a function call. In this example, the two output values of `poltocar` connect to the two input values of `min`:

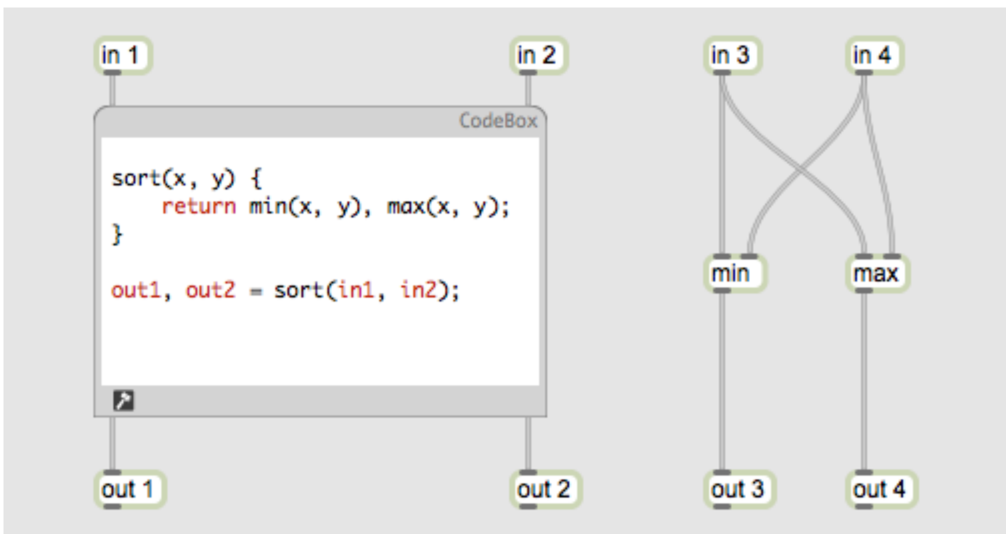
`out = min(poltocar(in1, in2))`

Defining GenExpr Functions

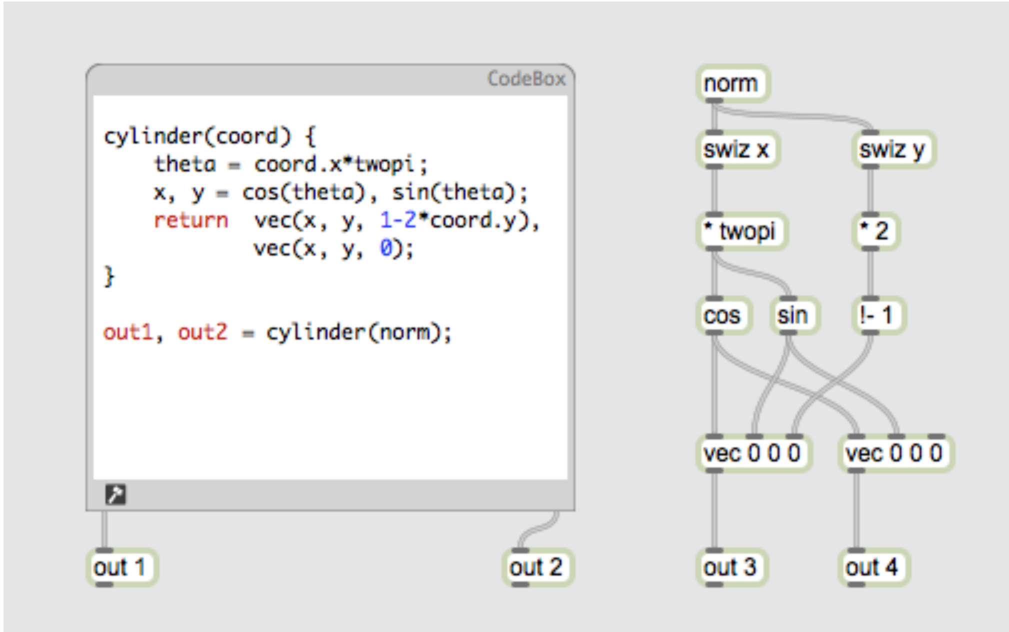
Defining new functions in GenExpr happens in much the same way as other familiar programming languages. Since there are no types in GenExpr function arguments are specified simply with a name. A basic function definition with an equivalent patcher representation looks like:



A function returning multiple values looks like:



The `cylinder` operator in Jitter Gen objects is defined as:



While simple functions in GenExpr can be easily patched together, more involved functions like the above `cylinder` definition start to become unwieldy, especially if the function is used several times within the GenExpr code. This is the advantage of textual representations.

Technical Notes

GenExpr is a type-less language. Variables are given types automatically by the compiler depending on the Gen domain and the Gen object's inputs. Gen variables are also local-to-scope by default so they don't have to be declared with a keyword like **`var`** as in JavaScript. Note that GenExpr has no array notation **`[index]`** as there is currently no notion of an array structure.

Common Operators

Comparison

- `<`, `lt` Returns 1 if in1 is lesser (in the positive direction) than in2, else returns zero.
- `>`, `gt` Returns 1 if in1 is greater (in the positive direction) than in2, else returns zero.
- `>=p`, `gtep` Returns in1 if in1 is equal to or greater (in the positive direction) than in2, else returns zero.
- `==`, `eq` Returns 1 if in1 equals in2, else returns zero.
- `==p`, `eqp` Returns in1 if it equals in2, else returns zero.
- `>=`, `gte` Returns 1 if in1 is equal to or greater (in the positive direction) than in2, else returns zero.
- `<=`, `lte` Returns 1 if in1 is equal to or lesser (in the positive direction) than in2, else returns zero.

- <=p, ltep Returns in1 if in1 is equal to or lesser (in the positive direction) than in2, else returns zero.
- <p, ltp Returns in1 if in1 is lesser (in the positive direction) than in2, else returns zero.
- max, maximum The maximum of the inputs
- min, minimum The minimum of the inputs
- !=, neq Returns 1 if in1 does not equal in2, else returns zero.
- !=p, neqp Returns in1 if it does not equal in2, else returns zero.
- step Reverse less-than operator, akin to the GLSL step operator.

Constant

- degtorad, DEGTORAD the constant pi/180
- e, E the constant e
- constant, f, float, i, int A constant value
- halfpi, HALFPI the constant pi/2
- invpi, INVPI the constant 1/pi
- ln10, LN10 the constant ln10
- ln2, LN2 the constant ln2
- log10e, LOG10E the constant log10e
- log2e, LOG2E the constant log2e
- pi, PI the constant pi
- radtodeg, RADTODEG the constant 180/pi
- sqrt1_2, SQRT1_2 the constant 1/sqrt(2)
- sqrt2, SQRT2 the constant sqrt(2)
- twopi, TWOPI the constant 2*pi

Declare

- param, Param Named parameters can be modified from the host object of the gen patcher. The first argument specifies the name of the parameter, the second argument specifies the initial value.

Expression

- codebox Evaluates GenExpr code and provides an in-patcher code editor.
- expr Evaluates GenExpr code.

Input-Output

- in Receive input into a gen patcher
- out Send output from a gen patcher

Logic

- `!`, `not` Zero input returns 1, any other value returns zero.
- `&&, and` Returns 1 if both in1 and in2 are nonzero.
- `bool` Any nonzero value becomes 1, zero passes through.
- `or`, `||` Returns 1 if either in1 and in2 are nonzero.
- `^^`, `xor` Returns 1 if one of in1 and in2 are nonzero, but not both.

Math

- `%`, `mod` Modulo inputs (remainder of in1 / in2)
- `+`, `add` Add inputs
- `/`, `div` Divide inputs
- `absdiff` Compute the absolute difference between two inputs
- `cartopol` Convert Cartesian values to polar format. Angles are in radians.
- `*`, `mul` Multiply inputs
- `neg` Negate input
- `poltocar` Convert polar values to Cartesian format. Angles are in radians.
- `!/`, `rdiv` Reverse division: divide in2 by in1
- `!%`, `rmod` Reverse modulo: in2 % in1
- `!-`, `rsub` Reverse subtraction: subtract in1 from in2
- `-`, `sub` Subtract inputs

Numeric

- `abs` Negative values will be converted to positive counterparts.
- `ceil` Round the value up to the next higher integer
- `floor`, `trunc` Round the value down to the next lower integer
- `fract` Return only the fractional component
- `sign` Positive input returns 1, negative input returns -1, zero returns itself.

Powers

- `exp` Raise the mathemtatical value e to a power
- `exp2` Raise 2 to a power
- `ln`, `log` The natural logarithm
- `log10` The logarithm base 10 of the input
- `log2` The logarithm base 2 of the input
- `pow` Raise in1 to the power of in2
- `sqrt` The square root of the input

Range

- `clamp`, `clip` Clamps the input value between specified min and max. Ranges are inclusive (both min and max values may be output). If two arguments are given, they correspond to the min and max values respectively. If one argument is given, it is assumed to correspond to the maximum value, while the minimum is set by the second inlet. If no arguments are given, min and max are specified by the second and third inlets (defaulting to 0 and 1 respectively).
- `fold` Low and high values can be specified by arguments or by inlets. The default range is 0..1.
- `scale` If four arguments are given, they correspond to the input low, high and output low high values respectively (an optional fifth argument specifies the exponential curve (default 1); otherwise these values are determined by the second through sixth inlets. The high and low values can be reversed for inverted mapping.
- `wrap` Low and high values can be specified by arguments or by inlets. The default range is 0..1.

Route

- `?`, `switch` Selects between the second and third inputs according to the Boolean value of the first: returns in2 if in1 is nonzero, else returns in3. If one argument is given, it specifies the 'true' value (and the right inlet is the 'false' value). If two arguments are given, they are the true and false values.
- `mix` Mixes (interpolates) between inputs a and b according to the value of the third input t, using linear interpolation. The factor (t) should vary between 0 (for a) and 1 (for b). If one argument is given, it specifies the mix (interpolation) factor.
- `smoothstep` Smoothstep is a scalar interpolation function commonly used in computer graphics. The function interpolates smoothly between two input values based on a third one that should be between the first two. The returned value is clamped between 0 and 1. The slope (i.e. derivative) of the smoothstep function starts at 0 and ends at 0.

Trigonometry

- `acos` The arc cosine of the input (returns radians)
- `acosh` The inverse hyperbolic cosine of the input
- `asin` The arc sine of the input (returns radians)
- `asinh` The inverse hyperbolic sine of the input
- `atan` The arc tangent of the input (returns radians)
- `atan2` Returns the angle to the coordinate (in2, in1) in radians.
- `atanh` The inverse hyperbolic tangent of the input
- `cos` The cosine of the input (in radians)
- `cosh` The hyperbolic cosine of the input
- `degrees` convert radians to degrees

- `hypot` Returns the length of the vector to (in1, in2).
- `radians` convert degrees to radians
- `sin` The sine of the input (in radians)
- `sinh` The hyperbolic sine of the input
- `tan` The tangent of the input (in radians)
- `tanh` The hyperbolic tangent of the input

MSP Operators

Audio

- `atodb` Convert deciBel value to linear amplitude
- `dbtoa` Convert linear amplitude to deciBel value
- `mstosamps` Convert period in milliseconds to samples
- `samplerate` The current samplerate
- `sampstoms` Convert period in samples to milliseconds.

Buffer

- `buffer`, `Buffer` References an external named buffer~ object. The first argument specifies a name by which to refer to this data in other objects in the gen patcher (such as peek and poke); the second optional argument specifies the name of the external buffer~ object to reference (if ommitted, the first argument name is used). The first outlet sends the length of the buffer in samples; the second outlet sends the number of channels.
- `channels` The length (in samples) of a data/buffer object. The first argument should be a name of a data or buffer object in the gen patcher.
- `cycle` An interpolating oscillator that reads repeatedly through one cycle of a waveform. If the buffer/data waveform is not specified, an internal sine table is used. By default it is driven by a frequency input, but if the @index attribute is set to 'phase', it can be driven by a phase input instead.
- `data`, `Data` Stores an array of sample data (64-bit floats) usable for sampling, wavetable synthesis and other purposes. The first argument specifies a name by which to refer to this data in other objects in the gen patcher (such as peek and poke); the second optional argument specifies the length of the array (default 512 samples); and the third optional argument specifies the number of channels (default 1, maximum 16). The first outlet sends the length of the buffer in samples; the second outlet sends the number of channels.
- `dim` The length (in samples) of a data/buffer object. The first argument should be a name

of a data or buffer object in the gen patcher.

- `poke` Write values into a data/buffer object. The first argument should be a name of a data or buffer object in the gen patcher. The second argument (or third inlet if omitted) specifies which channel to use. The first inlet specifies a value to write, while the second inlet specifies the sample index within the data/buffer. If the index is out of range, no value is written.
- `sample` Linear interpolated multi-channel lookup of a data/buffer object. The first argument should be a name of a data or buffer object in the gen patcher. The second argument specifies the number of output channels.
- `splat` Mix values into a data/buffer object, with linear interpolated overdubbing. The first argument should be a name of a data or buffer object in the gen patcher. The second argument (or third inlet if omitted) specifies which channel to use. The first inlet specifies a value to write, while the fractional component of the second inlet specifies a phase (0..1) within the data/buffer (indices out of range will wrap). Splat writes with linear interpolation between samples, and mixes new values with the existing data (overdubbing).

DSP

- `fixdenorm` Replace denormal values with zero.
- `fixnan` Replace NaN (Not a Number) values with zero.
- `isdenorm` Return 1 if the input is denormal, else return zero.
- `isnan` Return 1 if the input is NaN (Not a Number), else return zero.

Feedback

- `history`, `History` The history operator allows feedback in the gen patcher through the insertion of a single-sample delay. The first argument is an optional name for the history operator, which allows it to also be set externally as a parameter. The second argument specifies an initial value of stored history (defaults to zero). Denormal protection is automatically applied to the history input.

Filter

- `+=`, `accum`, `Accum`, `plusequals`, `PlusEquals` The object adds to, and then outputs, an internal sum. This occurs at sample-rate, so the sum can grow very large, very fast. The first optional argument specifies an initial value for the sum (default 0). The value to be added is specified by either the first inlet, or the second optional argument. The internal sum can be reset by sending a nonzero value to the right-most inlet.
- `change`, Change Returns the sign of the difference between the current and previous input: 1 if the input is increasing, -1 if decreasing, and 0 if unchanging.
- `dcblock`, DCBlock A simple high-pass filter to remove DC components.
- `delta`, `Delta` Returns the difference between the current and previous input.

- `*=`, `mulequals`, `MulEquals` The object multiplies by, and then outputs, an internal value. This occurs at sample-rate, so the stored value can grow very large or very small, very fast. The first optional argument specifies an initial value for the stored value (default 0). The value to be multiplied with is specified by either the first inlet, or the second optional argument. The stored value can be reset by sending a nonzero value to the right-most inlet.
- `phasewrap` Wrap input to the range -pi to +pi
- `pong` The first argument specifies the mode (0=fold, 1=wrap). Low and high values can be specified by additional arguments or by inlets. The default range is 0..1.
- `sah`, `Sah` The first inlet is the 'input' and the second inlet is the 'control'. When the control makes a transition from being at or below the trigger value to being above the trigger threshold, the input is sampled. The sampled value is output until another control transition occurs, at which point the input is sampled again. The default threshold value is 0, but can be specified as the last inlet/argument. The @init attribute sets the initial previous value to compare to (default 0).

MIDI

- `ftom` Frequency given in Hertz is converted to MIDI note number (0-127). Fractional note numbers are supported. An additional optional argument sets the tuning base (default 440).
- `mtof` MIDI note number (0-127) is converted to frequency in Hertz. Fractional note numbers are supported. An additional optional argument sets the tuning base (default 440).

Routing

- `gate` Similar to the MSP gate~ object. It takes an argument for number of outputs (one is the default) and lets you choose which the incoming signal (at the right inlet) is sent to according to the (integer) value in the left inlet. A value of zero or less to the left inlet will choose no output; a value greater than the number of outlets will select the last outlet. Like gate~, un-selected outlets will send zero.
- `selector` Similar to the MSP selector~ object. It takes an argument for number of inputs (one is the default) and lets you choose which incoming signal is sent to the output according to the (integer) value in the left inlet. A value of zero or less to this inlet will result in a zero signal at the output; a value greater than the number of inlets will select the last inlet.

Waveform

- `noise`, `Noise` A random number generator
- `phasor`, `Phasor` A non-bandlimited sawtooth-waveform signal generator which can be used as an audio signal or a sample-accurate timing/control signal.

- `train`, `Train` train~ generates a pulse signal whose period is specifiable in terms of samples.
- `triangle` A triangle/ramp wavetable with input to change phase offset of the peak value.

Jitter Operators

Color

- `hsl2rgb` Convert HSL to RGB
- `rgb2hsl` Convert RGB to HSL

Coordinate

- `cell` Cell coordinates of input matrix [0, dim-1]
- `dim` Dimensions of input matrix
- `norm` Normalized coordinates of input matrix [0, 1]
- `snorm` Signed normalized coordinates of input matrix [-1, 1]

Sampling

- `nearest` Nearest neighbor sample a matrix at a given coordinate (normalized)
- `sample` Sample a matrix at a given coordinate (normalized) with linear interpolation

Surface

- `circle` Equation of a circle taking input coordinates ranging from [0, 1]
- `cone` Equation of a cone taking input coordinates ranging from [0, 1]
- `cylinder` Equation of a cylinder taking input coordinates ranging from [0, 1]
- `plane` Equation of a plane taking input coordinates ranging from [0, 1]
- `sphere` Equation of a sphere taking input coordinates ranging from [0, 1]
- `torus` Equation of a torus taking input coordinates ranging from [0, 1]

Vector

- `cross` Take the cross product of two vectors
- `dot` Take the dot product of two vectors
- `length` Get the length of a vector
- `normalize` Normalize of a vector to unit length
- `reflect` Reflect a vector off a surface defined by a normal
- `swiz` Swizzle and mask vector components
- `vec` Pack scalar values into a vector