11 Expression Language

The built-in Expression Language allows you to animate values in many fields.

11 EXPRESSION LANGUAGE

Values in many fields can be animated using math functions and expressions, providing very powerful capabilities. Valid math expressions include:

- Numbers
- Strings
- Vectors
- Matrices
- Variables
- Arithmetic Operators
- Math Functions
- Interpolation Functions

11.1 NUMBERS

Three types of numbers can be used:

Integers	Integers are those numbers which have no decimal or fractional part, e.g.: 15, -3
Floating point numbers	Floating point numbers contain all numbers including Integers, and numbers with decimals, e.g.: 15.25
Exponential Notation	Exponential notation is a concise way of representing large numbers with a minimal amount of digits, e.g.: 3.2e-5 (equivalent to: 0.000032)

11.2 STRINGS

There are two methods for punctuating a command string:

"string"	Enclosing your command string with double-quotes will expand the variables within the string.
'string'	Enclosing your command string with single close-quotes won't expand the variables within the string

1.3 BUILT-IN VARIABLES

Two built-in Variables are re-computed automatically every frame. These Variables are *$F* (frame number) and *$T* (time in seconds).

Following, is a list of Houdini's built-in Variables and their function:

GLOBAL VARIABLES

PI	The value of the mathematical constant π (3.14159...). Use it to calculate the lengths of arcs. *2*$PI*r* (radius of circle) equals the circumference. Also, π = 180° expressed in radians.
E	The value of the mathematical constant e (2.71828...).
NFRAMES	Number of frames in the animation. The length is set with the Playbar Controls.
FPS	Number of frames per second. Set with the Playbar Controls.
FSTART	Start frame. Set with the Playbar Controls.
FEND	End frame. Set with the Playbar Controls.
F	The current frame, set with the Playbar Controls. Very important and commonly used Variable, especially for rendered picture filename numbering.
FF	Floating point frame number.
T	Current time in seconds. Equals *($F-1)/$FPS*
TLENGTH	Total length of animation in seconds.
TSTART	Start time of animation in seconds.
TEND	End time of animation in seconds.
HIP	Job directory. This defaults to the directory where you started Houdini. You can set it through the Textport.
HIPNAME	The name of the current *.hip* file.

CHANNEL VARIABLES

OS	Operator String. Contains the current OP's name.
CH	Current channel name.
IV	In value (value at start of segment).
OV	Out value.

IM	In slope
OM	Out slope
IA	In acceleration
OA	Out acceleration
LT	Local time - not including stretch or offset
IT	Start time of segment
OT	End time of segment
LIT	Local start time of segment
LOT	Local end time of segment
PREV_IT	Previous segment start time
NEXT_OT	Next segment end time

COP SPECIFIC VARIABLES

CSTART	Start frame of the current COP.
CEND	End frame of the current COP.
CFRAMES	Number of frames for the current COP.
CFRAMES_IN	Number of frames available from the first input COP.
CINC	Gets the global frame increment value.
W	Current image width.
H	Current image height

OP SPECIFIC VARIABLES & CHANNELS

Consult the specific OP section for their local variables. You will also find the applicable parameter channel names listed beside the Parameter titles.

OUTPUT DRIVER SPECIFIC VARIABLES

N	Current frame being rendered.
NRENDER	Number of frames being rendered.

1.4 ARITHMETIC OPERATORS

Following is a list of the arithmetic operations, in order of precedence:

( )	Operations in parentheses
-	Negation (e.g. 3 + -3 evaluates to 0)
* / ^ %	Multiplication, division, exponent and modulus
+ -	Addition and subtraction
< > == != || && !	Logical operators

All operators with the same priority are evaluated from left to right.

EXAMPLES

This:	**Evaluates to:**
3 + 4*5	23
2 * 3^2 + 4 * 6 / 2	48 (that is: 36+12)
((3 + 4) * 5 - 5) / 6	5
13 % 5 + 13 / 5	5.6 (that is: 3+2.6)
3.1 + 1	4.1

Note: `%`, the modulus operator, which is useful for cycling values, works on real numbers, not just integers. Therefore, the values returned might not be the values you expect. For example:

3.1415 % 1	Evaluates to .1415
34.999 % 5	Evaluates to 4.999

$F % 120 Gives a ramp from 0 to 119 which repeats starting at frame 120, 240 etc.

2 MATH & STRING EXPRESSION FUNCTIONS

The following math functions can be used:
(All angles in degrees unless otherwise noted)

abs(val) — This is a mathematical function used to express the absolute value of the number.
e.g. *abs(-2.6) = 2.6*

acos(val) — Trigonometric arccosine of val.
e.g. *acos(0) = 90*

arg(string line, float argNum) — This function extracts an argument from a line. The example below will extract the time out of the date string returned by the system function. Indexing begins at 0.
e.g. *arg(system(date), 3) = 20:17:46*

argc(string line) — Returns the number of arguments in the line. Counts the number of arguments in the line. Standard parsing is done, no variable expansion is done on the line.
e.g. *argc("This has four arguments") = 4*

asin(val) — Trigonometric arcsine of val.
e.g. *asin(.866025) = 60*

atan(val) — Trigonometric arctangent of val.
e.g. *atan(1.73205) = 60*

atan2(float y, float x) — Compute the arctangent of y/x. This is more stable than *atan()* since it can use the signs of y and x to determine the quadrant the angle is in. It also handles the case where *x* is zero correctly, returning 90 or -90.
e.g. *atan2(1, 0) = 90*
atan2(0, 1) = 0
atan2(0, -1) = 180

atof(string source) — Will convert a string into a floating point value.

ceil(float) — Takes the smallest integer greater than the float passed in. e.g. *ceil(2.718135) = 3*

chgroup(string) — Returns a string of all channels belonging to a channel group. This expression is helpful for performing operations on each channel in a channel group.

clamp(val, min, max) — Clamps the value between *min* and *max*. Used to prevent *val* from going outside the specified range.

cos(val) — Trigonometric cosine of val (in degrees).
e.g. *cos(60) = 0.5*

cosh(val) — Hyperbolic cosine of *val*.

deg(val) — Convert val to degrees (*val* is in radians).
e.g. *deg($PI) = 180*

distance(floatx1, floaty1, floatz1, float x2, float y2, float z2)
Returns *sqrt((x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2)* - the distance between the given points.

execute(string command) — Performs the Houdini command and returns the results. Trailing new-lines will be stripped from the returned result, and all embedded new-lines and tabs in the result will be converted to spaces.

exp(val) — Logarithmic exponentiation function of *val*.
e.g. *exp(2) = 7.38906*

eval(string expression) — Evaluates the string passed in as an expression. The result of this function is a floating point number.
e.g.
```
% set foo = 1+1
% set bar = system(date)
% echo `eval($foo)`
2
```

evals(string expression) — Evaluates the string passed in as an expression. The result of this function is a string value.
e.g.
```
% echo `evals($bar)`
Tue Dec  21 13:58:37 EST 1996
```

findfile(string filename) — Finds a file within the Houdini search path. The full path name of the file is returned. This function is useful when writing scripts.

% opparm read1 file (`findfile("geos/defgeo.bgeo")`)

Sets the file parameter of the Read SOP to the full path name of the first file which is found in the Houdini search path.

fit(num, oldmin, oldmax, newmin, newmax)
Returns a number between newmin and newmax that is relative to num in the range between oldmin and oldmax. e.g. *fit(3,1,4,5,20)=15*

fit01(num, newmin, newmax)
Return a number between *newmin* and *newmax* that is relative to *num* in the range between 0 and 1. If the value is outside the 0 to 1 it will be clamped to the new range.
e.g. *fit01(.3,5,20)=9.5*
See also: *fit fit11 fit10* .

fit10(num, newmin, newmax)

Returns a number between *newmin* and *newmax* that is relative to *num* in the range between 1 and 0. If the value is outside the 1 and 0 it will be clamped to the new range. e.g. *fit(.3,5,20)=15.5*
See also: *fit fit01 fit11* .

fit11(num, newmin, newmax)

Returns a number between *newmin* and *newmax* that is relative to *num* in the range between -1 and 1. If the value is outside the -1 to 1 it will be clamped to the new range. e.g. *fit(.3,5,20)=14.75*
See also: fit fit01 fit10 .

floor(val) — Returns the largest integer smaller than *val*.
e.g. *floor(2.78135) = 2*

ftoa(float number) — This expression will convert a number to a string. Type conversion is usually done automatically; however, you may wish to use this to force the conversion.

hsv(float R, float G, float B, string component)

Given the RGB color expressed by R, G, B, this function will return the HSV component specified. The component should be one of *H*, *S*, or *V*

if(exp, true_val, false_val) — Returns *true_val* if *exp* is true (i.e. *exp* > 0), and *false_val* if *exp* is false (i.e. *exp* = 0).
Example: if($F>12, $F, 12) will send back the value 12 unless *$F* is greater than 12, in which case it will return the current frame number.

index(string source, string pattern)

Finds the first occurrence of pattern in source and returns the number of characters before the pattern occurs. If the pattern is not found -1 is returned. See also: *rindex*. e.g.
> echo `index("Testing index", "sting")`
2
> echo `index("Testing index", "i")`
4

int(val) — Integer value of *val* by truncating.
e.g. *int(1.3) = 1, int(1.8) = 1,*
int(0.3) = 0, int(-0.3) = 0, int(-0.8) = 0

length(floatx, floaty, floatz) — Returns *sqrt(x*x + y*y + z*z)* - the length of the vector.

lock(float, float) — The *lock* function returns the value of its argument. It is special because when a parameter has an expression surrounded by a lock function, the value of the expression will not be changed until the lock function is removed.

log(val) — Natural logarithm of *val*.
e.g. *log(2.718281828)* = 1

log10(val) — Logarithm base 10 of *val*.
e.g. *log10(10)* = 1

max(val1,val2) — Returns the larger of value1 or value2.
e.g. *max (5,3)=5*

min(val1,val2) — Returns the smaller of value1 or value2.
e.g. *min (5,3)=3*

modblend (float val1, float val2, float length, float weight)
Blends the two modular values. This function can be used to correctly belnd two angles or other cyclic values. For example: modblend(355, 5, 360, 0.5) evaluates to 0. Simple linear blending of two values would result in an incorrect value of 180.

noise(x,y,z) — Generates random value given a position in space. Nearby positions will have similar values. The range of returned values is from 1.0 to -1.0.

padzero(float number, float value)
Returns a string containing value preceded by enough zeros to make up 'number' digits.
e.g. *padzero(5, 126)* = 00126
padzerp(5, 23) = 00023
padzero(1, 23) = 23

pow(float base, float exponent)
This computes the base to the power given.
e.g. *pow(2, 3)* = 8

pulse(val, min, max) — Creates an on/off pulse. If the *val* <*min* or >*max* then pulse returns 0, otherwise it returns 1.

rad(val) — Convert val to radians (*val* is in degrees).
e.g. *rad(180) = 3.1415926*

rand(val) — Produces a pseudo-random number between 0 and 1 depending on the value. If you use the same value on each use of this function, you will get the same random value. If you specify a different fractional value, *rand* produces a different result. e.g. *rand (12.1)* yields a different number from *rand (12.2)*.

Note: A very subtle problem exists because the math libraries on UNIX and NT are different. Therefore they occassionally yield different results when using the *rand()* function. This means that IFD's or RIB files generated on NT may not always be the same as IFD's or RIB files generated on SGI's.

rgb(float H, float S, float V, string component)

Given the HSV color expressed by H, S, V, this function will return the RGB component specified. The component should be one of *R*, *G*, or *B*.

rindex(string source, string pattern)

Finds the last occurrence of *pattern* in source and returns the number of characters before the pattern occurs. If the pattern is not found -1 is returned. See also: *index*. e.g.
> echo `rindex("Testing rindex", "sting")`
2
> echo `rindex("Testing rindex", "i")`
8

rint(float)

Rounds a number.
e.g. *rint(1.3) = 1, rint(1.8) = 2,*
rint(0.3) = 0, rint(-0.3) = 0, rint(-0.8) = -1

round(float)

Performs the same function as *rint*.

run(string)

Performs the Houdini command and returns the result. Trailing new-lines will be stripped from the result of the *run()*, and all embedded new-lines and tabs in the result will be converted to spaces.

sign(val)

Returns the sign of the value. For example, it returns 1 if the value is any positive number, -1 if the value is a negative number and 0 if the value is 0.

sin(val)

Trigonometric sine of *val*.
e.g. *sin(60)* =.866025

sinh(val)

Hyperbolic sine of *val*.

smooth(val, min, max)

Returns a smooth interpolation between 0 and 1.
If *val <min* it returns 0, if *val >max* it returns 1.
e.g. *smooth($F, 12, 55)* generates an ease-type curve between values 0 and 1 starting at frame 12, and ending at frame 55.

sqrt(val)

Square root of *val*.
e.g. *sqrt(144)* = 12

strcat(string, string)

This will concatenate two strings.
e.g. *strcat("Current motion file is; ", $HIPNAME)*
Current motion file is *job1.hip*

strcasematch (string pattern, string[s])

Performs a pattern matching comparison for a string ignoring case. If string matches pattern, return code =1; if string doesn't match, the return code =0. Specify multiple patterns using a comma-separated list.

Examples:
strmatch("FOO", "foobar")* returns 1
strmatch("?baR", "fred") returns 0
strmatch("FoO,bAr*, "bar")* returns 1
See also: *strcmp*, *strcasecmp*, *strcasematch*

strcasecmp (string s1, string s2)

Preforms a string comparison which ignores the case of the string. Return codes are:
negative if s1 < s2
positive if s1 > s2
zero if s1 == s2

strmatch (string pattern, string[s])

Performs a pattern matching comparison for a specified string. If the string matches the pattern, the return code is one; if the string doesn't match, the return code is zero. Multiple patterns may be specified using a comma-separated list. Examples:
strmatch("foo", "foobar")* returns 1
strmatch("?bar", "fred") returns 0
strmatch("foo,bar*", "bar")* returns 1
See also: *strcmp*, *strcasecmp*, *strcasematch*

snoise(X, Y, Z)

Applies noise based on sparse convolution. The X, Y, and Z are floating-point values. The theoretical bounds for the noise are about -2 to 2. However, typically, the bounds tend to be closer to -1 to 1.
e.g. *noise($TX, $TY, $TZ)*

strcmp(string1, string2)

Returns a negative number if *string1* is lexicographically less than *string2*. Returns a positive number if *string1* is lexicographically greater than *string2*. Returns a zero if *string1* is equal to *string1*.
e.g. *strcmp("abc", "xyz")* = -23
strcmp("xyz, "abc") = 23
strcmp("abc", "abc") = 0

strlen(string)

Returns the number of characters in the string.
e.g. *strlen("abcde")* = 5

sturb(X, Y, Z, depth)

Generates spatially coherent noise based on sparse convolution. The depth passed in is the amount of "fractalisation" which is done to the noise. The X, Y, Z, and depth parameters are floating point values.
e.g. *sturb($TX, $TY, $TZ, 5)*

substr(string, float, float)

This extracts a sub-string of the first argument.
e.g. *> echo `substr("abcdefghijklm", 3, 4)`*
defg
Note: The first character is specified by a start of 0

system(string)	Returns the output of a UNIX command. e.g. *system("finger")`* Directory: /usr/staff/john Shell: /bin/csh
systemES(string)	Returns the exit status of a UNIX command. e.g. *> echo `systemES("test -r $HOME")'* 0
tan(val)	Trigonometric tangent of *val*. e.g. *tan(60) = 1.73205*
tanh(val)	Hyperbolic tangent of *val*.
tolower (string[s])	Converts all the characters in the string to lower case.
toupper (string[s])	Converts all the characters in the string to upper case.
trunc(val)	Truncates all digits to the right of the decimal e.g. *trunc(4.5678) = 4*
turb(x,y,z, depth)	Like the *noise()* function, but allows for fractalisation of the noise. Thus, at a depth of three, the noise will be fractalised three times.
wrap(val, min, max)	Similar to *clamp()* in that the resulting value always falls between *min* and *max*. The difference is that it creates a sawtooth wave for continuously increasing and decreasing values of the value.

3 VECTOR / MATRIX EXPRESSIONS

3.1 VECTOR TYPES

Vectors are arbitrary-length arrays of floats. The operations listed below have been defined for vectors in Houdini.

OPERATIONS

vector[idx]	Extract the “idx” component from the vector
-vector	Negation
+vector	Positive
vector * vector	Cross product
vector / float	Divide a vector by a float
vector * float	Scale a vector by a float (associative)
vector + vector	Addition
vector - vector	Subtraction
vector == vector	Equality
vector != vector	Inequality

All of these operations result in vectors.

3.2 VECTOR TYPE FUNCTIONS

dot(vector v0, vector v1)
Computes the dot product between two vectors (returns float).

normalize(vector v)
Returns the normalized vector (returns vector).

vangle(vector v0, vector v1)
Returns the angle between the two vectors specified (returns float).

vector(string pattern)
The pattern passed in will be converted to a vector. The pattern should consist of a leading square bracket followed by a comma separated list of values and a closing square bracket (returns vector).
Example: *vector(“[1,2,3,4,5]”)*

vector3(float x, float y, float z)
Creates a 3 vector with the x, y, and z components specified (returns vector).

vector4(float x, float y, float z, float w)
Creates a 4 vector with the x, y, z, and w components specified (returns vector).

vlength(vector v)
Computes the length of the vector specified. This is equivalent to: sqrt(dot(v, v)) (returns float).

vlength2(vector v)
Computes the square of the length of the vector specified. This is equivalent to: dot(v, v) (returns float).

vorigin (string obj1, string obj2)
Returns all the values of the origin function at once. They are returned as a vector. The vector will contain "[TX, TY, TZ, RX, RY, RZ]". In many cases this will provide faster performance since the origin function is quite expensive to compute.

vrorigin (string obj1, string obj2)
Returns a vector specifying the rotations required to transform obj1 into the space of obj2. The vector takes the form "[RX, RY, RZ]".

vtorigin (string obj1, string obj2)
Returns a vector specifying the translation required to transform obj1 to the space of obj2. The vector takes the form "[TX, TY, TZ]".

vscale(vector v, float scale)
Multiplies the vector by the scale. This is equivalent to: vec*scale (returns vector).

vset(float size, float value)
Creates a vector of the size specified. Each component of the vector will be set to the value given (returns vector).

vsize(vector v)
Returns the number of elements in the vector (returns vector).
Examples:
vsize("[1,2]") = 2
vsize("[3,4,5,6]") = 4

3.3 MATRIX TYPE

The Matrix type allows for matrices of arbitrary rows and arbitrary columns to be used in expressions. Some functions will only work under certain conditions (i.e. square matrices or a specific number of rows/columns).

LIMITATIONS

3×3 or 4×4	This operation/function only works on 3×3 or 4×4 matrices. If the matrix specified is larger than 4x4, it will be converted to a 4×4 matrix for the operation. If the matrix size is less than 3×3, the matrix will be 'enlarged' to a 3×3 matrix for this operation. The

results of 'enlargement' of a matrix are not well defined (meaning the result may not be what is expected).

OPERATIONS

-matrix	Negation
+matrix	Positive
matrix * matrix	Matrix multiplication. Limitation: 3×3 or 4×4
vector * matrix	Multiply a vector by a matrix. Limitation: 3×3 or 4×4. For this operation, the vector will be truncated to the "best fit". The result of this operation is a vector.
matrix * float	Multiply every element of the matrix by the float value (associative).
matrix / float	Divide every element of the matrix by the float value.
matrix + matrix	Addition
matrix - matrix	Subtraction
matrix == matrix	Equality
matrix != matrix	Inequality

3.4 MATRIX TYPE FUNCTIONS

determinant(matrix mat)
Computes the determinant of the matrix. Limitation 3×3 or 4×4 (returns float).

dihedral(v0 v1)
Computes the dihedral matrix between v0 and v1. This is a rotation matrix which will rotate vector v0 to vector v1.

explodematrix(mat trs xyz component)
This explodes a 3×3 or 4×4 matrix into the euler rotations required to rebuild it. These components can be stuffed directly into Houdini rotation, scale, and translate channels.

mat is the matrix to transform. trs and xyz give the order of the expansion. In *trs*, a "t" represents translation, "r" rotation, and "s" scale. The xyz refers to the order of the rotations. The component is a string describing which channel to extract. It is "[trs][xyz]", where the *trs* chooses the channel between translate, rotate, and scale, and the *xyz* chooses the dimmension.

For example:

```
explodematrix(mlookat(vector("[1,0,0]"),vector("[0,1,0]")), "RST", "XYZ", "RZ")
explodematrix(identity(3)*2, "RST", "XYZ", "SZ")
```

identity(float size)
Creates an identity matrix of the size specified. That is, the resulting matrix will have *size* rows and size columns (returns matrix).

invert(matrix mat)
Inverts the matrix. Limitation: 3×3 or 4×4 (returns matrix).

matrix(string pattern)
Converts a string pattern to a matrix. The pattern should start with a square bracket, followed by a series of rows (specified as vector patterns - see the *vector()* function), followed by a trailing square bracket (returns matrix).
Example: *matrix("[[1,2,3][2,3,5],[-3,2,-3]]")*

mcols(matrix m)
Returns the number of columns in a given matrix.

mlookat(from to)
Computes a transform matrix specifying a lookat from the from point to the to point. The from and to vectors are converted to 3 vectors for this computation. The resulting matrix will be a 3x3 matrix.

morient(zaxis yaxis)
Computes the transform matrix to rotate the x,y,z axes such that the specified zaxis is the new zaxis and yaxis the new yaxis. The resulting matrix is a 3x3 matrix.

mrows(matrix m)
Returns the number of rows in a given matrix.

mzero(matrix mat)
Sets all values of the matrix to 0.

rotate(angle axis)
Computes a 4x4 rotation matrix of a rotation specified by the angle (in degrees) around an axis. The axis should be a string which is one of 'x', 'y', or 'z'.
See also: rotaxis, scale, translate

rotaxis(angle axis)
Computes a 4x4 rotation matrix of a rotation specified by the angle around the axis specified by the vector. The vector is converted to a 3 vector for the purposes of this computation. See also: rotate, scale, translate

scale(sx sy sz)
Computes a scale matrix given by the three scale values.
See also: rotate, rotaxis, translate

translate(tx ty tz)
Computes a translation matrix given the three translate values.
See also: rotate, rotaxis, scale

transpose(mat)
Computes the transpose of the matrix specified.

stripmatrix(mat)
This function will strip out all non-essential characters from the string representation of a matrix or vector. This allows users to pass matrix or vector expression results to VEX or RenderMan. A string containing the floating point numbers (and only the numbers) which make up the matrix will be returned.
Example: *stripmatrix(identity(3)) = "1 0 0 0 1 0 0 0 1"*
Example: *stripmatrix(vector3(1,2,3) = "1 2 3"*

transpose(matrix mat)
Computes the transpose of the matrix specified (returns matrix).

3.5 TRANSFORMATION FUNCTIONS

dihedral(vector v0, vector v1)
Computes the 3×3 dihedral matrix between v0 and v1. This is a rotation matrix which will rotate vector v0 to vector v1 (returns matrix).

mlookat(vector from, vector to)
Computes a 3×3 rotation matrix specifying a lookat from the *from* point to the *to* point. The from and to vectors are converted to three vectors for this computation (returns matrix).

optransform(string object_name)
Returns a matrix containing the transform for the given object.

rotate(float angle, string axis)
Computes a 4×4 rotation matrix of a rotation specified by the angle (in degrees) around an axis. The axis should be a string which is one of 'x', 'y', or 'z' (returns matrix).

rotaxis(float angle, vector axis)
Computes a 4×4 rotation matrix of a rotation specified by the angle around the axis specified by the vector. The vector is converted to a 3 vector for the purposes of this computation (returns matrix).

scale(float sx, float sy, float sz)
Computes a 4×4 scale matrix given by the three scale values (returns matrix).

translate(float tx, float ty, float tz)
Computes a 4×4 translation matrix given the three translate values (returns matrix).

3.6 EXAMPLES

Expression	*vector("[1,2,3,4,5]")*
Result	*[1,2,3,4,5]*
Expression	*vector3(1, 2, 3)*2*
Result	*[2,4,6]*

Expression *vector3(1, 2, 3) + vector4(5, 4, 3, 1)*
Result *[6,6,6,1]*

Expression *normalize("[1,2,3]")*
Result *[0.267261,0.534522,0.801784]*

Expression *vlength("[1,2,3]")*
Result *3.74166*

Expression *dot("[1,2,3]", "[-3,2,2]")*
Result *7*

Expression *matrix("[[1,2,3][3,2,1][-1,-1,0]]")*
Result *[[1,2,3][3,2,1][-1,-1,0]]*

Expression *rotate(45, "y")*
Result *[[0.707107,0,-0.707107,0]*
[0,1,0,0]
[0.707107,0,0.707107,0]
[0,0,0,1]]

Expression *rotate(45, "y")*rotate(45, "z")*
Result *[[0.5,0.5,-0.707107,0]*
[-0.707107,0.707107,0,0]
[0.5,0.5,0.707107,0]
[0,0,0,1]]

Expression *rotate(45, "y")*translate(1, 0, 0)*
Result *[[0.707107,0,-0.707107,0]*
[0,1,0,0]
[0.707107,0,0.707107,0]
[1,0,0,1]]

Expression *translate(1, 0, 0)*rotate(45, "y")*
Result *[[0.707107,0,-0.707107,0]*
[0,1,0,0]
[0.707107,0,0.707107,0]
[0.707107,0,-0.707107,1]]

Expression *vector("[1, 0, 0]")*rotate(45, "y")*
Result *[0.707107,0,-0.707107]*

Expression *vector("[0,1,2,3,4,5]")[3]*
Result *3*

Expression *(vector("[1,0,0]")*rotate(45, "y"))[0]*
Result *0.707107*

4 CHANNEL INTERPOLATION FUNCTIONS

4.1 CHANNEL INTERPOLATION FUNCTIONS

The following interpolation functions generate smooth values between 1 and 0:

bezier()	Interpolates the in and out point by fitting a Bézier curve using the slopes and accelerations.
cubic()	Interpolates the in and out point using the slopes. This results in a cubic polynomial.
constant()	Value is the same for the entire segment.
ease()	Values ease in smoothly starting at 1 and easeout smoothly at 0.
easein()	Ease in smoothly at the start only.
easeout()	Ease out at the end only.
easep(val)	An ease-in ease-out curve which is weighted by the power. This weighting has the effect of shifting the inflection point.
easeinp(val)	Same as *easep()* but it's an ease in curve.
easeoutp(val)	Same as *easep()* but it's an ease out curve.
linear()	Straight line interpolation from 1 to 0.
match()	A channel expression which matches the incoming and outgoing slope. The curve produced moves smoothly from the incoming value to the outgoing value.
matchin()	A channel expression which matches the incoming slope and extends the previous segment in a straight line from where it completes.
matchout()	A channel expression which calculates a straight line with the same slope as that of the next segment. The line is computed such that there is a smooth transition to the next segment.
quintic()	Interpolates the in and out point using slopes and accelerations. This results in a quintic (i.e. degree 5) polynomial.
raw()	The *raw()* interpolation function takes an array, or list of numbers and will use them as values for the channel.

repeat(frame1, frame2)	The *repeat()* function takes a frame range and repeats the animation curve in the specified range. For example, *repeat(1, 30)* repeats the animation curve between frames 1 and 30. You must make the repeated range outside of the current segment to avoid an impossible situation.
repeatt(time1, time2)	The *repeatt()* function is the "time" counterpart of the *repeat()* function. The *repeatt()* function takes a time range as opposed to frame range. *repeatt(0, 1)* repeats the animation curve between 0 and 1 seconds.
spline()	The *spline()* function takes each span of the neighbouring spline segments, and creates one smooth section.
vmatch()	A channel expression which matches the incoming and outgoing slope. However, this function allows you to override the values at the in and out points of the segment.
vmatchin()	A channel expression which matches the out going slope of the previous segment. A straight line will be generated starting at the value specified at the beginning of the segment.
vmatchout()	A channel expression which matches the in coming slope of the next segment. A straight line will be generated which will terminate at the value specified at the end of the segment.

REVERSING THE FUNCTIONS

You can reverse most of these functions by multiplying the function by -1. For example:

*linear() * -1* *ease() * -1* *easein() * -1*

It also possible to use these with other functions. For example:

*ch("spare1")*ease()*

5 CHANNEL / OPERATOR EXPRESSION FUNCTIONS

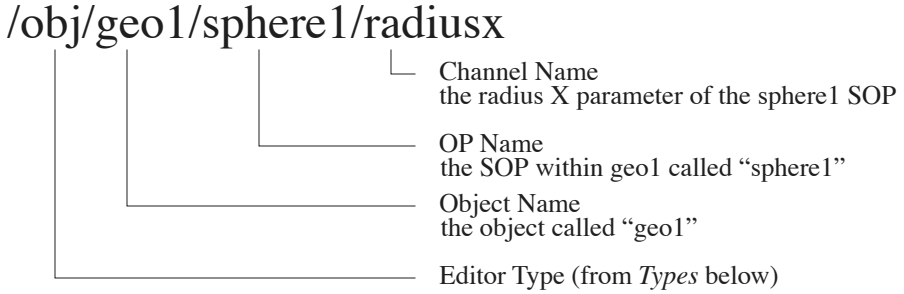
5.1 FOLDER PATH CONVENTIONS

When referencing any parameter or any object in houdini using the scripting language or expression functions, you should first be aware of the naming conventions used within Houdini to reference objects.

FOLDER PATHS

The nomenclature of Houdini folder paths is very similar to that of UNIX. The world is represented by / and then comes the editor's name, the OP name, and finally the channel name.

You can specify a pathname to any object, OP, or channel within Houdini. An example of a complete pathname to a channel would look like this:



You do not necessarily have to specify a complete pathname. You can use a portion thereof. For example, if you were referencing another SOP from an existing SOP, your pathname might only include: *../sphere2/radiusy* . In this case, the .. specifies that you are referencing one level up from the SOP, and then the */sphere2* specifies that you are referencing the SOP */sphere2*.

LIST TYPES

When specifying a complete pathname within Houdini, you need to specify one of the following list types:

List Name	Alias(es)	References to
obj	o	Objects
ch	ch	CHOP Networks
comp	c	Composite Networks
mat	m	Materials
out	dr, driver	Output Drivers
part	p	Particles

EXAMPLES

```
/comp/COPnet1/bright1/brightness
```

This means that you are referring to a composite list (/comp), you would like to examine */COPnet1* out of many possible Composite networks. Within */COPnet1* you want to select the */bright1* COP, and access the */brightness* channel of that COP.

```
/mat/blinn1/ortho1/wrapu
```

This means that you will be querying a material list, specifically the material */blinn1*, and you want to access the */wrapu* parameter of the *ortho1* TOP within the */blinn1* material.

PATH AND POINT EXPLANATION

The syntax for the *point()* function is:

point("../box2")	This object, the SOP named “box2”
ch("../box2/spare1")	This object, SOP “box2”, channel spare1
point("../../geo2/box2")	Object geo2, SOP box2 (note that here it’s probably easier to use the full path)

Note that in the *point()* function, you’re trying to find the OP, while in the channel function, you’re trying to find the channel.

example

Say that you’re in object geo1, SOP point1. Therefore, the full path of this SOP is:

```
/obj/geo1/point1
```

Let’s say that you want to reference the SOP box2 of this object. Therefore, you want to get to:

```
/obj/geo1/box2
```

So, the relative path would be "../box2". The .. takes you up to your current object, the box2 tells you to get the box2 OP from whatever object you belong to.

It’s very similar to the *ch()* function, but in the channel function, you’re specifying a channel of the OP that you're interested in. For example, if we wanted to get at the "tx" channel of the box2 SOP in the previous example, the relative path would be: "../box2/tx". When in doubt, use the full path name since you can’t easily go wrong.

SHORTHAND NOTATION WITHIN A NODE

Note: If you want to access another OP of the same type, for the *path* you can use:

../opname	if you are sourcing information from another OP.
opname	if you are sourcing information from within the OP.

5.2 CHANNEL EXPRESSION FUNCTIONS

ch("path/channel")

This extremely useful and frequently used function allows you to copy another OP's channel value at the current frame into the current channel.

path/channel	enter a Houdini folder path and channel here

Example: Entering the expression: *ch("/obj/geo1/sphere1/tx")* into some object's translate parameter will make that translate parameter the same as the */tx* (Translate X) parameter of */geo1/sphere1* .^

chexist (channel_name)

This function returns 1 if the specified channel exists, 0 if it doesn't.
Example: *echo `chexist("/obj/geo1/tx")`*

chf("path/channel", frame)

Copy another OP's channel at a specified frame into the current channel.

path/channel	enter a Houdini folder path and channel here
frame	The frame at which to evaluate (accepts float)

Tip: If you want the current frame + an offset rather a specific frame, use *$F+n* where *n* is the offset. For example *$F+12* evaluates at the current frame + 12.

chgroup(string groupName)

Returns a string containing all of the channels contained in the group specified. It is useful in the command language for traversing all channels in a group. For example:

```
foreach channel ( 'chgroup("group_name")' )
   echo $channel is in group_name
end
```

chs("path/channel")

Copy another OP's channel value into the current channel as a string. This is similar to *ch()*. For example:

```
echo `chs("/objects/geo1/lookat")`
```

echos the *lookat* parameter of object *geo1*.

cht("path/channel", time)

Copy another OP's channel at a specified time into the current channel.

path/channel	enter a Houdini folder path and channel here
time	The time at which to evaluate

5.3 GENERIC OPERATOR EXPRESSION FUNCTIONS

isvariable (variable_name)

Is there a system variable named *variable_name* , returns 1 if the variable is defined, and 0 if it isn't. See also: *ishvariable* .

ishvariable (variable_name)
Is there an houdini variable named *variable_name* , returns 1 if the variable is defined, and 0 if it isn't. See also: *isvariable* .

opnchildren (name)
Returns the number of nodes contained in the specified node. This returns the number of nodes in a sub-network or the number of SOPs in an Object. It is non-recursive in that only the direct contents of the node are counted, not all of their nodes as well.

opdigits(string name)
Return the numeric value of the last set of consecutive digits in an OP's name. For example, it gets *23* from an OP named *leg23right* . It is used when building several similar networks with small changes in each. For example:

```
opdigits("/obj/geo1") = 1
opdigits("..") = 1 (at the sop level of geo1)
```

opexist (op_name)
This function returns 1 if the specified node exists, 0 if it doesn't.
Example: *echo `opexist("../box1")`*

opflag (string network, string flag)
Returns a space-separated string of all nodes in the network which have the specified flag set. Recognized flags are:

`d`	Display Flag
`r`	Render Flag
`t`	Template Flag
`l`	Locked Flag
`s`	Selected Flag
`c`	Current Flag
`b`	Bypass Flag

Example:

```
hscript -> foreach obj ( `opflag("/obj", "d")` )
```

Loops through all the objects which are currently displayed.

```
hscript -> opgop group_name add `opflag("/obj", "d")`
```

Adds all the display objects to the group.

opfullpath (op)
This function returns the full path to the operator specified.
See also: *opname*, *opsubpath*.

opinput (string name, float index)
Displays the name of the operator that is connected to the input of the given index.

opsubpath (op)
This function will return the path of the specified operator including any containing subnets. It is similar to opfullpath, except instead of returning the full path to the op, it returns the name of the op preceded by any containing sub-networks.

For example, *opsubpath("/obj/sub1/geo1")* returns: *sub1/geo1* .
See also: *opfullpath()*, *opname()*.

opname ()
Returns the name of a SOP/COP or Object. It is used primarily to determine the name of an Object operator containing a SOP. For example, working from a Font SOP: *text:= 'opname("..")'* returns the name of the object that the Font SOP is in.

opselect (string network)
Returns a space-separated string of all the nodes which are selected in the specified network. This is very useful for scripting. For example:

```
hscript -> foreach obj ( `opselect("/obj")` )
```

loops through all the selected objects.

optype(string opname)
Returns the type of operator that the specified operator is (returns string).
For example:

```
hscript-> echo `optype("/obj/geo1")`
geo
hscript-> echo `optype("/obj/geo1/font1")`
font
```

5.4 SOP-SPECIFIC EXPRESSION FUNCTIONS

Many of the variables required in the following expression functions are listed in *Attributes* p. 233 in the *Geometry Types* section.

bbox("path", D_XMIN | D_XMAX | D_YMIN...)
Returns the component of the bounding box of the specified SOP.

path	enter a Houdini folder path and channel here
bbox spec	*D_XMIN, D_XMAX,* *D_YMIN, D_YMAX,* *D_ZMIN, D_ZMAX* *D_XSIZE, D_YSIZE, or D_ZSIZE.*

centroid (string sop, float type)
This function will return centroid information for a SOP. The *type* can be one of D_X, D_Y, D_Z for the corresponding components of the centroid (returns float).

curvature("path", prim_num, u, v)
Evaluates the curvature of the surface at the parametric (u,v) location. *u* and *v* are unit values, defined in the [0,1] interval. Note that if the primitive is a mesh, *u* and *v* are defined in terms of the number of its rows and columns.

path	Enter the SOP path and name here.
prim_num	primitive number to evaluate
u, v	u and v location on surface

degree("path", prim_num)
Returns the degree of the polynomial defining the face or hull whose primitive

number is specified. Polygons and meshes are expressed as linear functions, so their degree is 1. Spline types – NURBS and Bézier curves and surfaces – have degrees ranging from 1 to 10.

path	Enter the SOP path and name here
prim_num	primitive number to evaluate.

path	Enter the SOP path and name here for the point.
point_num	Point number to evaluate.
path	Enter the SOP path and name here for the primitive.
prim_num	Primitive number to evaluate.

return_type	0 - returns the distance 1 - returns the u parametric value closest to the point 2 - returns the v parametric value closest to the point

haspoint("group_name", "path/channel", point_number)

If target OP has the specified point, the function returns 1, if not, it returns 0.

group_name	Name of group (if specified); if not, the entire graphics detail is checked.
path/channel	Enter a Houdini folder path and channel here
point_number	point number to search for.

hasprim("group_name", "path/channel", prim_number)

If target OP has the specified primitives, the function returns 1, if not, it returns 0.

group_name	name of group if specified; if not, the entire graphics detail is checked.
path/channel	enter a Houdini folder path and channel here.
prim_number	primitive number to search for.

iscollided("soppath", point_number)

This function can be used to determine whether a particle is “collided”. For this function to work properly, the SOP specified should contain a particle system. The function will return 1 if the particle collided during last cook, 0 otherwise.

soppath	Houdini folder path of the SOP specified
point_number	point number to search for.

isspline("path", prim_num)

Returns 1 if the primitive is a spline, i.e. a NURBS or Bézier curve or surface. Otherwise, the value returned is 0.

path	enter the SOP path and name here
prim_num	primitive number to evaluate

isstopped("soppath", point_number)

This function can be used to determine whether a particle is “stopped”. For this function to work properly, the SOP specified should contain a particle system. The function will return 1 if the particle is stuck, 0 if the particle is not stopped (or if there is no particle matching the point passed in).

soppath	Houdini folder path of the SOP specified
point_number	point number to search for

isstuck("soppath", point_number)

This function can be used to determine whether a particle is “stuck”. For this function to work properly, the SOP specified should contain a particle system. The

function will return 1 if the particle is stuck, 0 if the particle is not stuck (or if there is no particle matching the point passed in).

soppath	Houdini folder path of the SOP specified
point_number	point number to search for

mindist("path", point_num, "path", prim_num, return_type)

This expression is an alias for the *pointdist()* function. Given a point and a primitive, this function will find the distance between the point and the closest spot on the primitive.

normal("path", prim_num, u, v, index)

Evaluates the X, Y, or Z component of the surface normal at the parametric (u,v) location. u and v are unit values, defined in the [0,1] interval. Note that, if the primitive is a mesh, u and v are defined in terms of the number of its rows and columns.

path	enter the SOP path and name here
prim_num	primitive number to evaluate
u, v	u and v location
index	x, y, or z component to evaluate

npoints("path/channel")

Returns number of points in a specific OP.

path/channel	enter a Houdini folder path and channel here

nprims("path/channel")

Returns number of primitives in a specific OP.

path/channel	enter a Houdini folder path and channel here

origin("path/object1", "path/object2", "parameter_type")

This function will return one of TX, TY, TZ, RX, RY, RZ value necessary to transform object1 to object2, depending on the type argument ("TX", "TY", "TZ", "RX", "RY" or "RZ"). This can also be thought of as the position of object2 relative to object1. It will compute the position of object1 relative to object2 and returns one of TX, TY, TZ, RX, RY, RZ based on the type argument.

Note: The rotate values returned by the *origin/vorigin/vrorigin* functions are only valid if there is no difference in the scale channels between the two objects.

originoffset (obj1 pos1 obj2 pos2 constant_type)

This function will return one of TX, TY, TZ, RX, RY, RZ value necessary to transform the point *pos1* in the space of object *obj1* to point *pos2* in the space of object *obj2*, depending on the type argument ("TX", "TY", "TZ", "RX", "RY" or "RZ"). This can also be thought of as the position of *pos2* in *obj2* relative to *pos1* in *obj1*. See also: *origin(), vorigin(), vtorigin(), vrorigin()*.

opoutput(name, index)

This allows you to find out which nodes use the given node as an input (i.e. which nodes are outputs of the given node). For example, in objects, you can find out who your children are by querying the outputs. (You can find out the parent of an OP by using *opinput()*).

opninputs(name)

Returns the maximum number of inputs that the node receives input from. In

some cases, it is possible to have blank inputs. For example, the Particle SOP takes three inputs (the source, the collision geometry and the metaball force geometry). If the SOP has the source and force inputs connected, *opninputs()* will return 3 (indicating that there are a maximum of three inputs). However, *opinput()* will return a blank for the second input (i.e. *opinput(particle1, 1)* == "").

opnoutputs(name)

Returns the number of nodes which reference this node as their inputs. Unlike *opninputs()*, there will be no blank entries in the list. The order of the outputs is arbitrary.

param(string param Name, float value)

This function is only applicable in conjunction with the Copy SOP. It retrieves a value into a field from another SOP based on the value of *param Name*. See example in *Ref. Volume I > Creating Stamped Geometry* p. 523.

point("path/channel", point#,"attrib_type", subtype)

Extract attribute information from a point in a SOP's geometry. You can find a list of attribute names for the *point()*, and *prim()* functions by looking at the list of attributes in the SOP's info pop-up. The attributes have a number in square brackets after them – this indicates how many parameters exist in the attribute. For example, uv[3] means there is a UV Attribute with three parts: u, v, and w, for which you use the ordinal numbers 0, 1, and 2 respectively.

```
point("../point1", 55, "uv", 0)    -> 0 is for u
point("../point1", 55, "uv", 1)    -> 1 is for v
```

path/channel	enter a Houdini folder path and channel here	
point#	point number to acquire information from	
attrib_type	Type of attribute. This is the same as the attribute name in a SOP's info pop-up.	
	Cd	Color
	P	Position
	N	Normals
	v	Velocity
subtype	The address of the actual variable held within the type.	
e.g.:	*"P", 0*	X position
	"P", 1	Y position
	"P", 2	Z position
	This allows for the inclusion of any number of actual variables held within a variable type.	
e.g.:	point("../xform1", 0, "P", 0)	in X field
	point("../xform1", 0, "P", 1)	in Y field
	point("../xform1", 0, "P", 2)	in Z field

pointavg("path", "attribute")

Returns the average value for the specified attribute.

Example: *pointavg("/obj/geo1/sop1", "P", 0)*

This returns the average X position of the points in the SOP.

pointdist("path", point_num, "path", prim_num, return_type)

Given a point and a primitive, this function will find the distance between the point and the closest spot on the primitive.
return_type 0 yields the minimum distance.
return_type 1 yields the U knot value at the point of minimum distance.
return_type 2 yields the V knot value at the point of minimum distance.
return_type 3 yields the primitive number that was closest.
For example: *pointdist("/obj/geo1/add1", 0, "/obj/geo1/grid1", 0, 0)*
Returns the distance between point 0 of add1 and the closest spot from the surface of grid1 primitive number 0. If the *return_type* was 1 then the U knot value that is closest to the point would be returned.
See also: *primdist(), nearpoint(), xyzdist()* .

Note: You need to scale the knot vector using the Basis SOP if you want to use *pointdist()* in a UV situation like a Carve SOP.

pointlist(string sopname, string group_name)

Returns a space-separated list of points contained within the group.

points (SOP, point_number, attribute)

This function returns the value of a string attribute for a given point of a SOP.
For example: *points("/obj/geo1/facet1", 3, "instance")* returns the string associated with the string attribute 'instance' for point 3 in the SOP *facet1* located within *geo1* .

prim("path/channel", prim_number, "var_type", subtype)

Extract information from a primitive in a SOP.

path/channel	enter a Houdini folder path and channel here	
prim_number	primitive number to acquire information from	
var type	type of variable. This is the same as the attribute name in the geo info button.	
	Cd	Color
	P	Position
	N	Normals
	v	Velocity
subtype	The address of the actual variable held within the type.	
e.g.:	*"P", 0*	x position
	"P", 1	y position
	"P", 2	z position
	This allows for the inclusion of any number of actual variables held within a variable type.	
e.g.:	prim("../transform1", 0, "P", 0)	in x field
	prim("../transform1", 0, "P", 1)	in y field
	prim("../transform1", 0, "P", 2)	in z field

primdist(SOP prim1_num SOP prim2_num return_type)

This expression finds the minimum distance between two primitives.
return_type 0 yields the minimum distance.
return_type 1 yields prim1's u value at the point of minimum distance.
return_type 2 yields prim1's v value at the point of minimum distance.

return_type 3 yields prim2's u value at the point of minimum distance.
return_type 4 yields prim2's v value at the point of minimum distance.

Currently, *primdist()* will return 0 unless given face types (polygons and/or curves) or spline surfaces. For example:

```
primdist("/obj/geo1/sphere1", 0, "/obj/geo1/grid1", 0, 0)
```

Will return the distance between the first primitives in both sphere1 and grid1.
See also: *pointdist()*.

primduv("path", prim_num, attrib_name, attrib_index, u, v, du, dv)

Evaluates the (partial) derivatives of a face or hull primitive attribute at a parametric (u,v) position. u and v are unit values, defined in the [0,1] interval. When given the "P" or "Pw" attribute, the positional derivative of (u,v)'s image on the primitive will be returned. If the primitive is a face type, v and dv are ignored. If both du and dv are 0, primduv becomes equivalent to primuv. Note that if the primitive is a polygon or a mesh, u and v are defined in terms of the number of vertices, or rows or columns respectively.

path	enter the SOP path and name here
prim_num	primitive number to evaluate
attrib_name	name of attribute to evaluate; similar to *var_type* above
attrib_index	index of the attribute; similar to *subtype* above
u, v	u and v location
du, dv	partial derivatives of u and v

Example:

```
primduv("/obj/geo1/tube1", 12, "P", 2, 0.4, 0.5, 1, 0)
```

Evaluates the Z component of the first-order partial derivative of primitive 12 with respect to u, at the parametric location (0.4, 0.5).

primlist(string sopname, string group_name)

Returns a space-separated list of primitives in the specified group.

prims (SOP, primitive_number, attribute)

This function returns the value of a string attribute for a given primitive in a SOP. For example: *prims("/obj/geo1/facet1", 3, "texturemap")* returns the string associated with the string attribute 'texturemap' for primitive 3 in the *facet1* SOP in *geo1* .

primuv("path", prim_num, attrib_name, attrib_index, u, v)

Evaluates the specified attribute of a face or hull primitive at a parametric (u,v) position. *u* and *v* are unit values, defined in the [0,1] interval. When given the "P" or "Pw" attribute, the x, y, or z image of the (u,v) domain point will be returned. If the primitive is a face type, v is ignored. If the primitive is a polygon or a mesh, u and v are defined in terms of the number of vertices, or rows or columns respectively.

path	enter the SOP path and name here
prim_num	primitive number to evaluate
attrib_name	name of attribute to evaluate; similar to *var_type* above
attrib_index	index of the attribute; similar to *subtype* above
u, v	U and V location

Example1: Finding the Color on a Surface

```
primuv("/obj/geo1/tube1", 0, "Cd", 1, 0.7, 0.3)
```

Evaluates the Green component of the diffuse color attribute at a location on primitive 0 given by the parametric coordinates uv=(0.7, 0.3).

Example 2: Finding the Radius of a Sphere

To compute the radius of a sphere for which you know the point number, you can use an expression like the one below to find the radius. Why not just use the *ch()* function? Because the sphere may be deformed and resized subsequent to definition in the Sphere SOP.

```
length(
primuv(sop, num, “P”, 0, 0, .5) - point(sop, num, “P”, 0),
primuv(sop, num, “P”, 1, 0, .5) - point(sop, num, “P”, 1),
primuv(sop, num, “P”, 2, 0, .5) - point(sop, num, “P”, 2)
)
```

This gives the X radius (i.e. evaluates the sphere at uv=(0, .5)). To get the Y radius, you would evaluate at uv=(0, 1). For the Z radius, evaluate for uv=(0.25, 0.5).

realuv(SOP, prim_num, uv_unit, D_U|D_V)

Returns the real u or v parametric value given the unit value of the same parameter. The unit value is defined in the [0,1] interval. The real value is defined in the valid interval of the primitive's domain if the primitive is a spline type. If the primitive is a polygon or a mesh, the size of its domain is given by the number or vertices, or rows or columns respectively. If the primitive is a polygon or a curve, D_U and D_V are irrelevant. Note that the result is undefined if the primitive is neither a face nor a hull.

path	enter the SOP path and name here
prim_num	primitive number to evaluate
uv_unit	a value between 0 and 1
D_U|D_V	defines operation in u or v parametric direction

spknot("path", prim_num, knot_index, D_U|D_V)

This spline-specific function returns the floating-point knot value given the knot index in the U or V knot sequence. The first valid knot index is 0. If the primitive is a Bézier curve or surface, the values returned are those of its breakpoints. If the primitive is a curve, D_U and D_V are irrelevant.

path	enter the SOP path and name here
prim_num	primitive number to evaluate
knot_index	the number (>=0) of a knot in the knot sequence
D_U|D_V	defines operation in u or v parametric direction

unituv("path", prim_num, uv_real, D_U|D_V)

Returns the unit u or v parametric value given the real value of the same parameter. The unit value is defined in the [0,1] interval. The real value is defined in the valid interval of the primitive's domain if the primitive is a spline type. If the primitive is a polygon or a mesh, the size of its domain is given by the number or vertices, or rows or columns respectively. If the primitive is a polygon or a curve, D_U and D_V are irrelevant. Note that the result is undefined if the primitive is neither a face nor a hull.

path	enter the SOP path and name here

prim_num	primitive number to evaluate
uv_real	real value of U and V
D_U\D_V	defines operation in U or V parametric direction

uvdist (SOP prim1_num u1 v1 SOP prim2_num u2 v2)

This expression finds the distance between two primitives at two parametric locations. Valid U and V values are between 0 and 1. Any primitive type is allowed. For example:

```
uvdist("/obj/geo1/sphere1", 0, 0.1, 0.8, "/obj/geo1/grid1", 2, 1,
0.5)
```

Returns the distance between point (0.1, 0.8) on the first primitive in sphere1 and point (1, 0.5) on the third primitive in grid1. See also: *distance()*, *primdist()*, *pointdist()*, *unituv()*

vertex("path/channel", prim#, vertex#, "attrib_type", subtype)

The *vertex()* function is similar to the *point()* function. It extracts attribute information from a vertex in a SOP's geometry. It is necessary to specify the primitive and the vertex numbers in order for this function to work.

path/channel	enter a Houdini folder path and channel here
prim#	primitive number to acquire information from
vertex#	vertex number to acquire information from
attrib_type	Type of attribute. This is the same as the attribute name in a SOP's info pop-up.

Cd	Color
Alpha	Transparency
uv	Texture
P	Position

subtype	The ordinal index of the actual variable held within the attribute.

Two special attributes exist, *P* and *Pw* , which represent the position of the point in space. *Pw* allows you to access the *W* component of the position. For example:

```
vertex("/obj/geo1/facet1", 2, 3, "P", 0)
```

Returns the X component of vertex 3 of primitive2 in the SOP facet1.

```
vertex("/obj/geo1/facet1", 2, 3, "Cd", 2)
```

Returns the Z component of the colour attribute of vertex 3 of primitive 2 in the facet1 SOP of the object geo1.

You can find a list of attribute names by looking at the list of attributes in a SOP's info pop-up. The attributes have a number in square brackets after them – this indicates how many parameters exist in the attribute. For example, uv[3] means there is a UV Attribute with three parts: u, v, and w, for which you use the ordinal numbers 0, 1, and 2 respectively.

Note: This function will interpolate between values if the vertex number is fractional, such as 3.35.

vertexs (SOP, primitive_number, vertex_number, attribute)
This function returns the value of a string attribute for a given vertex (of a given primitive) in a SOP. For example: *vertexs("/obj/geo1/facet1", 1, 3, "instance")* returns the string associated with the string attribute 'instance' for vertex 3 of primitive 0 in the *facet1* SOP in *geo1* .

xyzdist(x, y, z, SOP, prim_num, return_type)
Finds the distance between the point (x, y, z) and the specified SOP's primitive. If the *prim_num* is -1, it finds the closest distance to any primitive in the mentioned SOP.

return_type 0 yields the minimum distance.
return_type 1 yields the u parametric value at the point of minimum distance.
return_type 2 yields the v parametric value at the point of minimum distance.
return_type 3 yields the primitive number that was closest.

For example: *xyzdist(1, 2, 3, "/obj/geo1/grid1", 0, 0)*
returns the distance between (1, 2, 3) and the closest spot from the surface of grid1 primitive number 0. If the return_type were 1, the u parametric value that is closest to the point would be returned. See also: *primdist, nearpoint, pointdist* .

5.5 POP-SPECIFIC EXPRESSION FUNCTIONS

popevent(string, event_name)
Returns whether a POP event is occurring or not. This function should be used from within a POP.

popeventtime(string, event_name)
Returns the time at which the named event occurred. It returns -1 if the event has no yet occurred. This function should be used from within a POP.

poppoint(point_number, attribute_type, index)
Returns the value of an attribute for a specific particle within a pop function. Use this only from within a POP.

point_number	Point number to acquire information from.	
attribute_type	Type of attribute. This is the same as the attribute name in a POP's info pop-up:	
	v	Velocity
	accel	Acceleration
	life	Life of particle
	id	Unique particle ID
index	The address of the actual variable held within the type.	

5.6 COP-SPECIFIC EXPRESSION FUNCTIONS

These functions become available when the first COP is created:

pic(copname, U, V, color_type)
Look up a pixel colour. The pixel value returned will be between 0 and 1, and the U and V parameters must be between 0 and 1.

picni(copname, U, V, color_type)
Look up pixel colour (without interpolation). The pixel value returned will be between 0 and 1, and the U and V parameters must be between 0 and 1.

res(copname, res_type)
Look up the "natural" resolution of a COP where *res_type* is one of:

D_XRES	Horizontal resolution (width).
D_YRES	Vertical resolution (height).

e.g. `res("/comp/ice1/color1", D_XRES)`

5.7 CHOP-SPECIFIC EXPRESSION FUNCTIONS

CAN BE USED ANYWHERE IN HOUDINI

chop("choppath/channelname")
This evaluates *choppath/channelname* at the current time/frame. An example is `"/ch/ch1/wave1/chan1"`.

chopf("choppath/channelname", frame)
This evaluates the channel at the specified `frame`.

chopt("choppath/channelname", time)
This evaluates the channel at the specified `time`, expressed in seconds.

chopi("choppath/channelname", index)
This evaluates the channel at the specified *index* .

chopcf("choppath", channelnum, frame)
Like *chopf()* this evaluates the channel at the specified *frame*, but the channel is specified with two fields: the CHOP name and the channel number, as an index that starts at 0. Example: *chopcf("/ch/ch1/wave1", 0, 61)* .

chopct("choppath", channelnum, time)
Like *chopt()* this evaluates the channel at the specified *time*, but the channel is specified with two fields: the CHOP name and the channel number.

chopci("choppath", channelnum, index)
Like *chopi()* this evaluates the channel at the specified *index*, but the channel is specified with two fields: the CHOP name and the channel number.

chopstr("choppath/channelname")
Like *chop()*, this evaluates a CHOP output channel at the current time/frame. It returns not a float (a raw number) like the other functions here, but a text string in ASCII containing the same number. (Both output out the same, so they appear similar.)

chopn("choppath")
This returns the number of channels in the CHOP.

chops("choppath")
This returns the start index of CHOP, expressed as samples. To express the start of the CHOP in seconds, divide this by *chopr()* .

chope("choppath")
This returns the end index of CHOP.

chopl("choppath")
This returns the length of the CHOP in samples.

chopr("choppath")
This returns sample rate of the CHOP.

example

To get the value of a channel from a Wave CHOP at the previous frame:

As it is in UNIX, / is the root of all data, and in Houdini, */ch* is where the CHOP Networks are located. So if the CHOP Network is *ch1*, and the Wave CHOP is called *wave1*, you need: */ch/ch1/wave* .

But this identifies the CHOP, not its channels. So you need to append the channel you want to this, resulting in: */ch/ch1/wave/chan1* . So the channel function is:

```
chopf("/ch/ch1/wave1/chan1", $F-1)
```

CAN ONLY BE USED LOCALLY WITHIN CHOPS

If you are working within the context of CHOPs – say by putting math expressions in the Expression CHOP and fetching channels from Expression's input – you should use the faster functions:

beat(keyname parameter)
Get the value attached to one of the keyboard keys. The first argument is the key name, including the 0 to 9 keys, named *KEY0* to *KEY9*, the A to Z keys named *KEYA* to *KEYZ*, and the keypad keys named *KEYPAD0* to *KEYPAD9*. The second argument is the parameter to retrieve: VAL, SPEED, BINARY, ANALOG, TICK, COUNT, PERIOD, CYCLE.

icn(input)
The returns the number of input channels at *input*, where 0 is the first input to the CHOP.

ics(input)
This returns the 'start index' of the *input*.

ice(input)
This returns the 'end index' of the *input*.

icl(input)
This returns the length of the *input*, expressed in samples.

icr(input)
This returns the sample rate of the *input*.

icmax(input, channelindex)
This returns the maximum value in the channel *channelindex* of *input*.

icmin(input, channelindex)
This returns the minimum value in the channel *channelindex* of *input*.

ic(input, channelindex, sampleindex)
This returns the value of the channel number *channelindex,* of input number *input*, at sample number *sampleindex*. They all start at 0.

oc(channelindex, sampleindex)
This gets values from the output channel as the CHOP's output is being calculated. While, for example the Expression CHOP is computing its output at index `$I`, it can access the output values at the previous index, `$I-1`. This is useful when stepping forward frame-by-frame. The *oc()* function is only valid for *sampleindex* < *$I*.

Note: *ics, ice, icl, chops, chope, chopl* return values in terms of 'sample index' and represent the entire CHOP.

6 CUSTOM EXPRESSION FUNCTIONS

6.1 INTRODUCTION

Custom Expression Functions are a way of extending the built-in expression language using a simple 'C' like scripting language. These functions can then be used anywhere you use a built-in function.

BASIC FORM

The basic form of a custom expression is:

```
# Function to double a number.

double(number)
{
  number = number * 2;
  return number;
}
```

Once you have entered this custom expession function, you can use it within any Houdini edit field as if it were a regular function. For example, you could then type:

```
sin( double($F) )
```

into the Centre-X field of a Sphere SOP, and it would provide values to the *sin()* function that are equal to double that of $F at a given frame.

EDITING

The source for expression functions is stored within a .hip file. Therefore any changes made to the source must be made from within Houdini. This can be done using *exedit* (see *exedit* p. 110 of the *Scripting* Section).

You can also edit Expression Functions via the *Dialogs > Aliases/Variables...* dialog. Once the dialog appears, select the *Expressions* tab to view a list of currently existing functions. Select one, and click the *Edit* button to edit the function in a text editor (default = *vi* . Set this with the EDITOR environment variable – see Reference > Interface > *Edit Fields* p. 213 for how to change this). Once you have finished editing and saved, the function will be available throughout Houdini.

For more information on creating your own expression functions, also see the *Expressions* section of the *User Guide*.

6.2 EXAMPLES

FACTORIAL()

```
# All variables in expression functions are floating point,
# therefore, no declaration block is needed.
# This function computes the factorial of a number.

factorial(number)
{
     result = 1;
     for (i = 2; i < number; i++)
          result *= number;
     return result
}
```

SIGN_INC()

```
# Add the sign of a number to itself.

sign_inc(number)
{
      if (number < 0)
      {
          number--;
      }
      else if (number > 0)
      {
          number++;
      }
      return number;
 }
```

WAVENOISE()

Comuputes noise for waves (as used in the *UG > Rendering > Water* example).

```
wavenoise(x, y, z, amp, rough, exponent)
{
    n = 0;
    l = 1;
    for (i = 0; i < 3; i++)
    {
        n += snoise(x, y, z) * l;
        x *= 2;
        y *= 2;
        z *= 2;
        l *= rough;
    }
    if (n < 0)
        n = -pow(-n, exponent);
    else n = pow( n, exponent);
    return n * amp;
}
```

7 PATTERN MATCHING

7.1 STRING MATCHING

String matching characters are used to incorporate or specify larger numbers of referenced items in an expression using special characters and rules.

*	Wild card – matches any character or group of characters
?	Matches any single character
[string]	Matches only a character in the string. Does not support the hyphen syntax of [a-z].
^	Only significant at the beginning of a pattern. Specifies removal from a previous match.
@	Specifies an object or channel group (depending on the context). Allows you to use a group wherever you might have listed specific OP names before.

EXAMPLES

geo*	Matches everything beginning with “geo”.
[gG]eo*	Matches everything beginning with “geo” or “Geo”.
?eo*	Matches everything that has any character followed by “eo” and then any number of characters.
* ^geo1	Matches everything except the string “geo1”.

7.2 NUMERIC PATTERN EXPANSION – USED FOR GROUP PARSING

There are three places where this happens:

- The Group SOP
- During Group specification in SOPs

The Group SOP is slightly different from specification in SOPs.

THE GROUP SOP

*	Matches all points/primitives.
number	Matches a single number.
start-end	Matches all numbers between start to end (inclusive).
start-end:step	Matches numbers between start and end skipping every "step" numbers.
start-end:keep,step	Matches numbers between start and end skipping every "step" numbers. However, in each step, "keep" number are selected from the beginning.
!pattern	Specifies everything except the specified pattern. The pattern can be any of the above.
^pattern	Specifies removal of the pattern from a previous match.

Note: Multiple patterns *must* be specified as a space separated list. Commas are not allowed as separator characters.

examples

10-20	Choose numbers 10 through 20 (including 10 and 20)
0-30:2	Choose every other number between 0 and 30 (i.e. 0, 2, 4, 6, ... 30)
0-30:2,3	Choose every 2 of 3 numbers between 0 and 30 (i.e. 0, 1, 3, 4, 6, 7, ... 30)
!3-5	Choose everything except 3-5
0-100:2 ^10-20	Choose every other number between 0 and 100 except for numbers between 10 and 20.

GROUP SPECIFICATION & MODEL COMMANDS

The group specification and model commands are slightly different than the Group SOP. The patterns here include not only the numeric patterns, but also group name pattern matching. This combines the string matching with the numeric matching.

*, ?, [ ]	String matching wild cards. These apply to group name specifications. It is important to note that the asterix (*) has a different meaning in these patterns than in the Group SOP. In these patterns, it matches all group names instead of all numbers.
number_pattern	All number patterns are the same as the group SOP.

Note: As with the Group SOP, multiple patterns must be specified as a space separated list. Commas are not allowed.

examples

g*	All groups starting with "g"
g* ^group1	All groups starting with "g" except for group1
* ^10-20	All groups but not numbers 10 to 20
g* 10-20:2,3	All groups starting with "g" as well as every two of three numbers between 10 and 20.

Primitive Numbers in Patterns

A paste hierarchy created with the Paste SOP displays it's the paste hierarchy's primitive number in brackets – e.g. (5) – when primitive numbers are enabled in the Viewport options. However, the brackets should not be used when specifying the primitive number. It should be treated just like any other primitive number.

8 EXPRESSIONS IN FILE NAMES

Tip: Don't use spaces in filenames. Although both Unix and NT recognise their usage, a space in a filename will sometimes be interpreted as seperating the two parts of the filename into two seperate filenames, and cause endless trouble with getting your paths recognised correctly. Instead of a space, use a dash (-).

USING $F IN FILENAMES

Use *$F* in a file name to key on the frame numbers for a sequence of images
For example:

mine$F.pic produces filenames: *mine1.pic, mine2.pic, mine3.pic...*

FILENAMES WITH LEADING ZEROS

To key image names with leading zeros, put a single digit non-zero number (N) after *$F*, and it will generate N digit frame numbers. For example:

mine$F3.pic produces filenames: *mine001.pic, mine002.pic, mine003.pic...*

FILENAMES BASED ON RESOLUTION

To store images in directories based on image resolution, use a path like this:

Pics${W}x${H}/$F.pic

Then output at different resolutions gets put into different directories.

FILENAMES WITH THE NAME OF THE CURRENT OPERATOR

To store the name of the current operator (say in the name of a Z-depth map, you want the name of the light within the filename), you can use $OS.

$OS-$F.pic

Gives filenames like: *light1-1.pic*, *light1-2.pic*, *light1-3.pic*, etc. Changing the name of the operator (say from "light1" to "light1wHalo") automatically updates the output filenames as well.

FILENAMES WHICH OFFSET FROM THE CURRENT FRAME

If you're loading a sequence of images, say into the COP Editor, or as a Texture map, and you want the filename to increment with the frame number, but be offset from the current frame by a certain amount (say you want to read in the *current frame + 12*), you could use something like:

MyImage`$F+12`.pic

The backquotes cause Houdini to evaluate the expression within them, so when you're at frames 1, 2, and 3, it will read in the images: *MyImage13.pic* , *MyImage14.pic* , and *MyImage15.pic* .

FILENAMES WHEN RENDERING FIELDS

If you're rendering fields, the frame number may not be the best solution, as integer values for frame 5.0 and 5.5 would both evaluate to 5 – causing frame 5.5 to overwrite frame 5.0. In this case $N or $FF might be a better choice.

Following, is a table which compares filenames, rendering frame 30 to 33 by frame increment of 0.5 .

$N	$FF	Filename $F	Filename $FF	Filename $N
1	30	30.pic	30.pic	1.pic
2	30.5	30.pic	30.5.pic	2.pic
3	31	31.pic	31.pic	3.pic
4	31.5	32.pic	31.5.pic	4.pic
5	31.9999	32.pic	31.9999.pic	5.pic
6	32.5	32.pic	32.5.pic	6.pic
7	33	33.pic	33.pic	7.pic

advantages / disadvantages

$F.pic — Since $F is an integer, it rounds at the half frames. This causes the previous output files to be overwritten.

$FF.pic — Aside from having floating point frame numbers, you may end up with binary-to-decimal arithmetic errors (like 31.9999), which produces bad filenames.

$N.pic — $N doesn't pick up the starting frame, it only counts the total frames rendered. So, if you render a sequence of 1-10, then render 11-20, they would all have the same filename.

Pick your poison.

2 Expression Cookbook

I INTRODUCTION

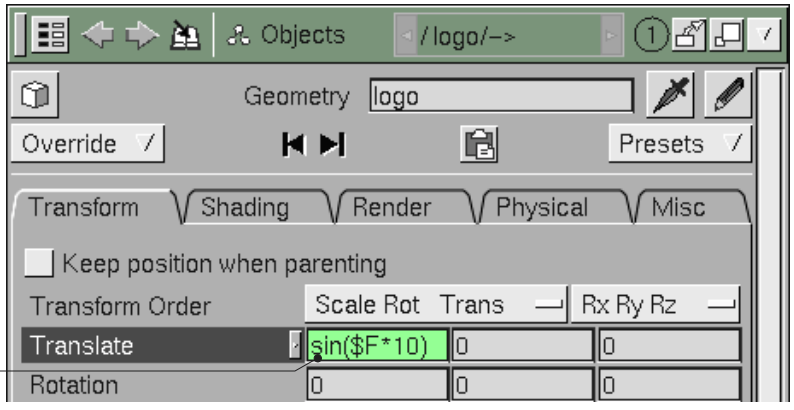
You know that you can type a value into a parameter field, and the number you type in is then a fixed value assigned to that parameter. For example, if you type '5' into the Translate-X field of an object, it will move over 5 units in the Viewport, and it will stay there until you type in another number.

You also know that if you want to animate the position of the object, you can Key-frame the positions using the either Select state or the Channel Editor.

Two other ways also exist to control the values being plugged into the various parameters – you can use CHOPs to feed in the values, or you can use a mathematical expression to describe changes in value over time.

In Houdini, time is refered to through the use of global variables. The number of frames through which you have played is accessed by using the variable *$F*. Because the value of $F changes from frame to frame – it allows us to introduce the element of time into our expressions. For example, if we wanted to move an object back and forth using a sine wave, we could type an expression like: *sin($F*10)* into an object's Translate-X parameter. Try this:

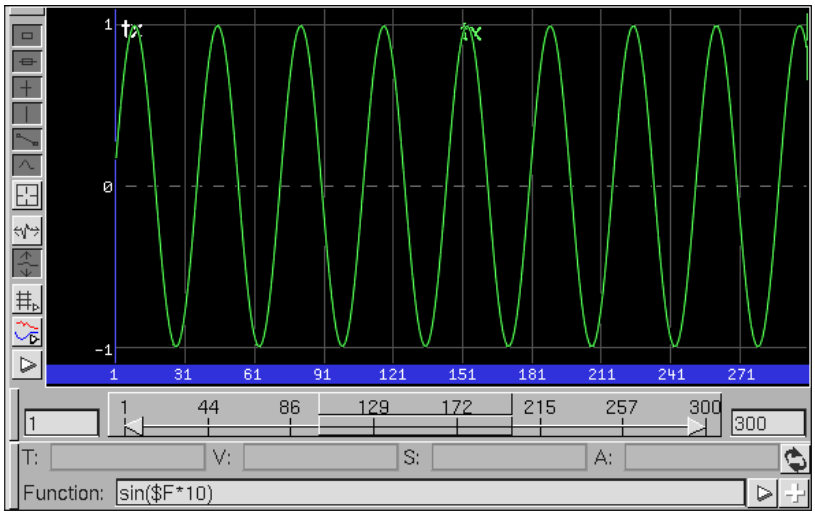
1. Start-up Houdini, place a Geometry object and in the parameters, type the expression: *sin($F*10)* into the Translate-X field, type Enter, and click the *Play* button.



You should see the object move back and forth in the Viewport. This is because the value of $F constantly changes the result of the *sin()* function. If we simply typed $F into the Translate-X field without putting it inside a *sin()* function, you would

quickly see the logo move offscreen from 0 to 300 units to the right (assuming the default animation length of 300 frames).

2. If you want to see a graphical representation of the oscillating values generated by the *sin()* function, simply click on the *Translate* parameter, and in the menu that appears, select *Scope Channels*. You should then see a graph of the values being fed into the Translate-X field by our *sin($F*10)* expression. You can see that as the values go up and down, the logo moves back and forth.



3. Now say we wanted to make the logo move back and forth twice as fast. If we were keyframing, this would be a rather tedious process. With expressions, we simply double the frequency used in the *sin()* expression. Change the expression to: *sin($F*20)* . You should see *logo* move back and forth twice as fast – you have just saved yourself a lot of keyframe work.

Note: From here on in, any time you enter an expression, you should automatically type Enter after typing it in.

4. Now say we want the logo to oscillate up and down along the Y axis instead of left and right along the X axis. To do this: Stop the playback, select the expression and copy it with Ctrl C. Then on the *Translate* parameter and select *Delete Channels* from the menu that appears. Set the X value back to 0. Now, in the Translate-Y field, paste the expression with Ctrl V, and type Enter to accept the entry. Click *Play*. You now see *logo* moving up and down along the Y axis.

5. To increase the amplitude of the oscillations, we multiply the expression by some number. Multiplying it by 1.0 makes it the same height; multiplying by a number less than 1 (i.e. 0.0 - 0.999) shrinks it; and a number greater than 1.0 makes it bigger. To make it double the size, we multiply by 2 . So, change the expression to: *sin($F*20) * 2* . You should now see the *logo* object moving twice as high.

SUMMARY

Using mathematical expressions to control the values of parameters gives us a greater amount of control than if we had manually keyframed the bouncing motion.

Changes in the frequency and amplitude are easily adjustable in a way that would be quite difficult with keyframing.

2 USING ABS() TO MAKE IT BOUNCE

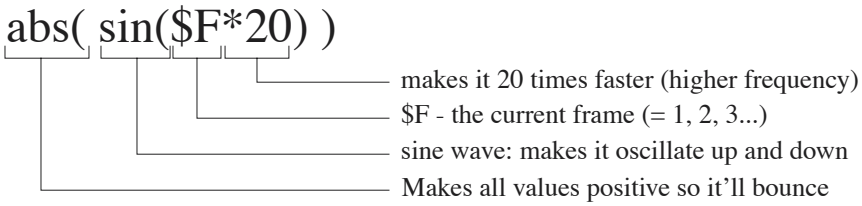
2.1 INTRODUCTION

We have seen how the *sin()* function allows us to make something oscillate back and forth. Is there a way we could change that behaviour into a bounce? We can do it by enclosing the *sin()* expression within an *abs()* function.

2.2 EXERCISE

1. Change the SOPs of the *logo* object. Click on the *logo* object with and select *Edit SOPs*. Then in the SOPs, delete the *file1* and *xform1* SOPs, and place a Sphere SOP. You now have a sphere in the Viewport instead of a Houdini logo.

2. Go back up to the Object Editor, and select *logo* to edit its parameters. It should still have the *sin($F*20)* expression in the Translate-Y field. We want to change it by enclosing it within an *abs()* function. We do this quite literally – in the Translate-Y parameter, change the expression to read:

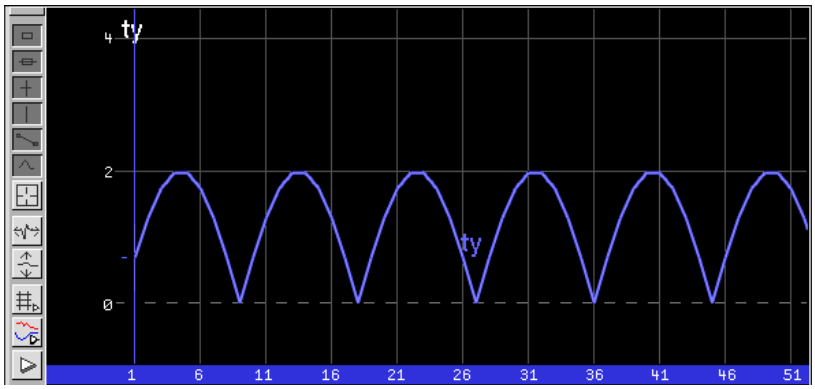


Note: For each open bracket *(* we *always* need a corresponding close bracket *)* .

3. Click *Play*, and observe how the sphere bounces. Home your view if necessary.

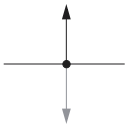
2.3 VIEWING A GRAPH OF THE EXPRESSION

Just as we did before, we can see a graph of the *abs(sin())* expression in the Channel Editor by doing a click on the *Translate* parameter, and selecting *Scope Channels* from the pop-up menu. You should see the bounce values for */ty* channel like so:

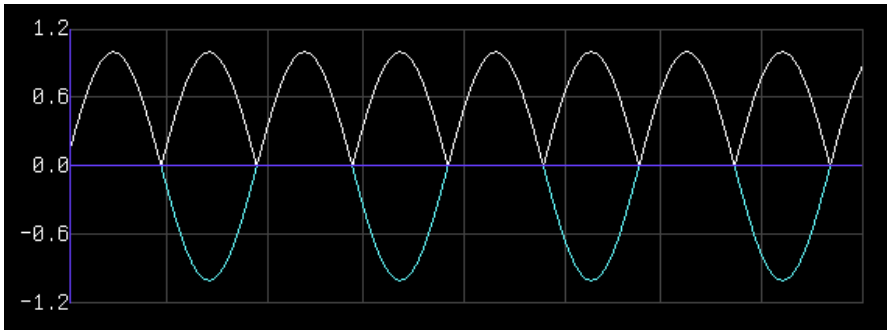


2.4 EXPLANATION

Because a sine function oscillates between 1.0 and -1.0, when we had only the sine expression: *sin($F*20)* in the */ty* (Translate-Y) field, we saw the sphere oscillate up and down between +1.0 and -1.0 units. The 'absolute' function *abs()* converts all negative values (whenever it dips below 0) to positive values – thus we get the bouncing motion.



abs() converts all negative values to positive ones



3 CH() FUNCTION – PHOTOCOPY A CHANNEL

3.1 THE CH() FUNCTION

A useful trick is to key one parameter's values into another parameter with the *ch()* function. This allows you to dynamically copy the values of one channel into any other parameter. This is extremely useful when you want to key one behaviour off of another existing behaviour.

For example, say you wanted to have a Tube OP change it's height based on the translation of of the *logo* object. To do this, you would use the *ch()* function to retrieve that value into the Tube OP. Assuming that you haven't changed the expression in the *logo* object from the previous exercise, you would enter:

```
ch("/obj/logo/ty")
```

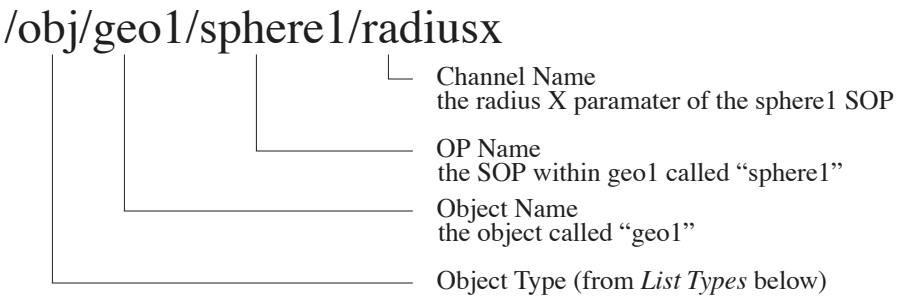
into the *Height* parameter of the Tube OP. This way, the height in the Tube OP will change in direct proportion to the translation of the *logo* object based on the value it retrieves from the Translate-Y field of the *logo* object.

Tip: You can have Houdini do this for you by clicking on the name of a parameter, and selecting *Copy Parameter*, and then going to the parameter where you want the channel reference, and clicking and selecting *Paste Copied References*.

3.2 PATHNAMES

To get at any parameter in Houdini with the *ch()* function, you first need to know it's pathname. The nomenclature for a Houdini pathname is very similiar to that of the UNIX directory structure. The world is represented by / and then comes the list type, the OP type, the OP name, and finally the paramater name.

You can specify a pathname to any object, OP, or channel within Houdini. An example of a complete pathname to a channel would look like this:



You do not necessarily have to specify a complete pathname. You can use a portion thereof. For example, if you were referencing another SOP from an existing SOP – say */sphere1*, your pathname might only include: *../sphere2/radiusy* . In this case, the .. specifies that you are referencing one level up from the SOP, and then the / *sphere2* specifies that you are referencing the SOP */sphere2* .

You can find out the channel names by viewing them in a *Channel List* pane. The channel names for individual parameters are also listed beside the parameter names in the SOP/COP/POP sections of the Reference manual.

LIST TYPES

When specifying a complete pathname within Houdini, you need to specify one of the following list types at the beginning of the path:

List Type	References to
/obj	Objects
/comp	Image Composites
/out	Output Renderers
/part	Particle Systems
/ch	Channels (Animation & Audio)
/shop	Shaders
/vex	Vector Operations

3.3 EXAMPLES

- The Translate X, Y, and Z values of sphere1 inside geo1:

```
ch("/obj/geo1/sphere1/tx")
ch("/obj/geo1/sphere1/ty")
ch("/obj/geo1/sphere1/tz")
```

- The amount of Blending in a Sequence blend SOP (sblend2) in geo8:

```
ch("/obj/geo8/sblend2/blend")
```

- The Color in RGBA values, of a Color COP (color3) in a COP Network (comp1):

```
ch("/comp/comp1/color3/colorr")
ch("/comp/comp1/color3/colorg")
ch("/comp/comp1/color3/colorb")
ch("/comp/comp1/color3/colora")
```

- The Translate-Y value of font1 located inside geo3, when referenced from another SOP also contained within geo3:

```
ch("../font1/ty")
```

- The Scale-X amount of a Transform SOP that you want to put in the Scale-Y field of the very same SOP so that Scale-Y will always equal Scale-X within that SOP:

```
ch("./sx")
```

- To find the X and Y resolution of cam1 in a script (at the current frame):

```
set xres = `ch("/obj/cam1/resx")`
set yres = `ch("/obj/cam1/resy")`
```

Tip: You can refer to the *object name* in the expression with the variable *$OS* which refers to the current object – this is useful if you change the name of your objects; because then you don't need to change each object reference.

3.4 THE CHF() FUNCTION

The *chf()* function is the same as the *ch()* function, except that instead of using the value at the current channel, you can specify a specific frame. For example:

```
chf("/obj/geo1/sphere1/tx, 13)
```

Will read the value of *geo1/sphere1's* Translate X channel at frame 13. This will be a constant value, as for every frame, it still references back to frame 13. If instead you wanted a *frame offset* of 13 frames, you would add $F to the value:

```
chf("/obj/geo1/sphere1/tx, $F+13)
```

3.5 THE CHS() FUNCTION

For parameters that have a check button, you can use the *chs()* function, which evaluates a parameter as a string. You can type the following into a Textport:

```
> echo `ch("/out/mantra1/tscript")`   # Eval toggle as float
  0
> echo `chs("/out/mantra1/tscript")`  # Eval toggle as string
  off
> echo `chs("/out/mantra1/command")`  # Eval string parameter
  mantra3 -v 0.01
> echo `chs("/obj/light1/lookat")`    # Eval menu choice
  geo1
```

3.6 USING CHANNEL VALUES IN AN ARITHMETIC EXPRESSION

You can also use the channel value within an arithmetic expression, for example:

```
ch("/obj/$OS/spare1") + ch("/obj/$OS/tx") + 2
```

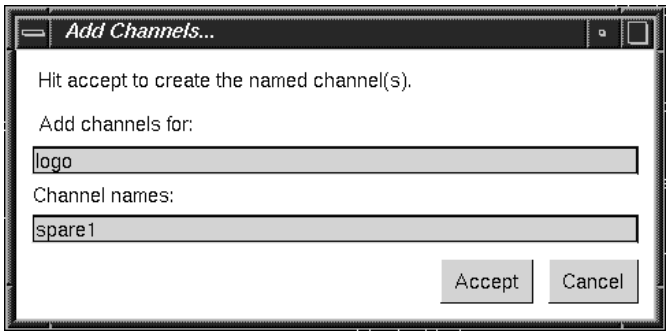
takes the value of the channel *$OS/spare1* and adds the X value of that point plus two.

4 ADDING A SPARE CHANNEL

4.1 INTRODUCTION

From our previous example, we have a bouncing sphere, and it bounces at a constant amplitude for the entire length of the animation. If we want the height of the bouncing to decay over time, we can resort to keyframing and edit the values manually (which destroys the handiness and of changing it via exprressions), or we can be smart about it, and add a spare channel to control the decay by multiply the expression that causes the bouncing by the values held in the spare channel.
Do the following:

1. Select a geometry Object, then Ctrl click in the Layout Area, and select the *Add Spare Channels...* In the dialog that appears, specify the name of a new "spare" channel. By default, it will add the channel to the selected object (e.g. 'logo').



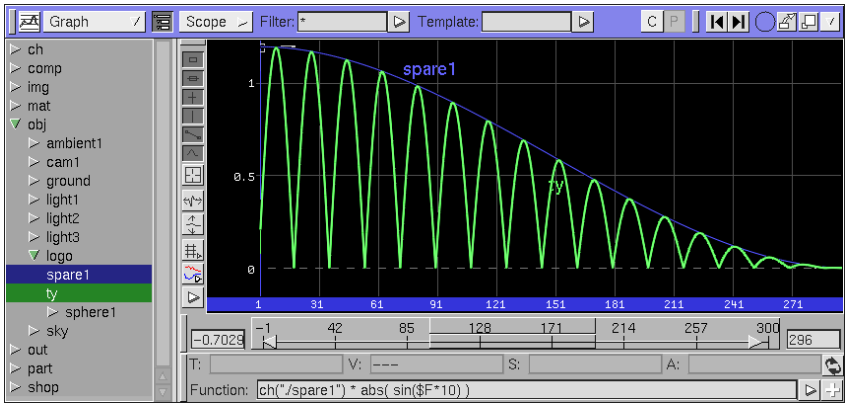
2. Click *Accept* to add the spare channel, and dismiss the dialog. If you have a Channel Editor pane open, you should see *spare1* appear under the object.

3. In the Channel List, go to the object, and click on *ty* channel. The function defining *ty* is displayed. Change the function to read: *ch("./spare1") * abs(sin($F*10))*

4. Hold down the Shift key, and click on *spare1* in the Channel list. You should now see both *spare1* and *ty* selected in the list. However, *spare1* initially contains nothing but zero values, so both graphs are flat.

5. Click and drag the small value handle for *spare1* at time 0 upwards. Stop dragging when you get to a value of about 1.2. Be careful not to drag too far.

6. You should see the amplitude of *ty*'s bounces slowly scaled up within an envelope defined by *spare1*. The graph should look something like the one below.



7. Experiment by dragging the value at 0 for *spare1* up and down to control the decay of the bounce over time.

Using a spare channel gives us very precise and realistic control over the bouncing behaviour using only a single parameter - this provides a much more flexible solution than what we would have got with the tedium of keyframing.

4.2 EXPLANATION

The *ty* channel controls the value of the Y translation of the sphere over time. By defining the value of *ty* as a sine function based on $F (the current frame number), we tell Houdini that the value of Yshould follow a sine wave (oscillating up and down). We modify this sinusidal motion by bracketing the *sin()* function with an *abs()* function to get the bouncing motion. To scale the bouncing over time, we multiply this with a spare channel using a *ch()* function:

```
ch("./spare1") * something
```

which yields control of the bounces via *spare1* the way we want. You can multiply any expression function in this way.

Another example of a use of spare channels would be to animate the number of copies in a Copy SOP by placing an expression like:

```
ch("./spare1")
```

into the *Number of Copies* parameter. We don't need to use a full pathname in this instance, because the */spare1* channel resides inside the same object that is calling it.

Tip: As you get better at editing expressions, you may want more space to edit what you're typing. Typing Alt E in any edit field will pop-up a text editor so you can edit the text using *vi*. To change the editor, see *Edit Fields* p. 213 in the *Interface* section of the Reference manual (which explains how to set the $EDITOR variable). For information on *vi*, see the *spy* section of the Reference manual.

4.3 DELETING A SPARE CHANNEL

There is no neat way to remove a spare channel through the GUI, you have to resort to using a Textport command to do this. You can delete any channel using the *chrm* command. For example:

```
chrm /obj/geo1/spare1
```

RENAMING CHANNELS

It is difficult to rename channels after they are created, so take care to get the desired name right the first time.

5 THE POINT OP

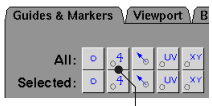
5.1 INTRODUCTION

In geometry, every OP has a list of points numbered from 0 up. Each point has an XYZ location, colour, alpha, texture UV, weight (W), and a normal. Every polygon, NURBS, or primitive also has a list of vertices, which references this list of points by their ordinal number (i.e. their position within the list of points – 1st, 2nd, 3rd, etc.). These points are shared between multiple polygons/NURBS/primitives (you can get more specific info on this in: *Geometry Types > Geometry Detail* p. 223).

In the case of polygons, if two adjacent polygons which share an edge use the same two points at the end of a shared edge, then it will be smooth-shaded across the shared edge. This shading is controlled by the normals of the shared points.

The Point OP can reference each of these points (and their normals) and modify them. For example, it can add or double the distance of a point from the centre of the object's bounding box (using the local variables: $BBX, $BBY and $BBZ), or change the color of a point ($CG, $CY, $CZ), or even change the normal of a point (for example, by doubling it: $NX*2, $NY*2, $NZ*2).

This is useful for a wide variety of things. We can use this to deform the shape of an object (point positions), create intresting colour effects (point colours), and alter the initial trajectories of particles by altering the point normals. This is why the Point OP so powerful – it allows access to the raw data in your geometry.



Point Numbers

Tip: To see the point numbers, click the ✣ button at the bottom-right of a Viewport to see the Viewport Display options. Then enable the Point Numbers icon.

For more information on points, vertices, and primitive types, see the *Geometry Types* section of the Reference manual.

5.2 OPS

The Point and Primitive OPs are particularly versatile for using expressions to create particular forms or animations. Some OPs have local variables that can only be used in that particular OP, such as *$PT* for the Point OP. Use the *Help* button on an OP to see which local variables are available, or the *Reference Manual*. For a complete description of the expression language see the *Expression Language* section of the Reference manual. You can also use the *$T* and *$F* time variables in any OP expression to add motion to the expression. The examples in the following section *Expression Cookbook* p. 56 were created by using a Point OP.

5.3 SEGMENTS

You can set the segment function of any channel to an expression instead of the default *ease()* or *constant()*. This can be done in the Channel Editor pane.

5.4 POINT() & POINTAVG() FUNCTIONS

POINT()

The *point()* function is used to extract information about a particular point from a SOP. The basic form for *point()* is:

```
point (SOP, point_number, attribute, index)
```

This function extracts information from a point in an OP. The attribute parameter is the name of the attribute (e.g. "Cd" for diffuse colour). Two special attributes exist, "P" and "Pw" which represent the position of the point in space ("Pw" allows you to access the W component of the position). For example:

```
point ("/obj/geo1/facet1", 3, "P", 0)
```

Returns the X component of point 3 of the *facet1* OP in *geo1*.

```
point ("obj/geo1/facet1", 3, "N", 2)
```

Returns the Z component of the normal attribute of point 3 in the *facet1* OP of the object *geo1*. For a list of valid attributes, see the *Geometry Types* section.

Note: This function will interpolate between point values if the point number is fractional, such as 3.35 .

POINTAVG()

```
pointavg (SOP, attribute, index)
```

This function works much like the *point()* function, except that it returns the average value of the attribute for all points in the specified OP.

5.5 FLIPPING NORMALS

You can flip the point normals of incoming geometry by using a Point OP, setting it to *Add Normal*, and entering:

```
-$NX  -$NY  -NZ
```

in the fields. This works, because it takes the existing normals ($NX, $NY, $NZ), and inverts them with the preceding - sign .

5.6 WHERE TO FIND MORE FUNCTIONS

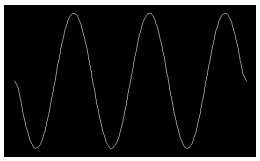
Look in the *Expression Language* section of the Reference manual for a complete listing of all Houdini functions.

6 EXPRESSION COOKBOOK

6.1 INTRODUCTION

The following expression examples were created by using a Grid OP set to 1 row by 180 columns, and *Connectivity* set to *Rows*. To this, a Point OP was appended. The Position X, Y, and Z (Pos X, Pos Y, and Pos Z) were modified with the expressions shown to produce the display in the Viewport. To use as a segment expression, use the same expression in a channel, and replace the position variable *($PT, $TX or $BBZ)* with *$T*.

6.2 WAVES



The sine (and cosine) function are extremely versatile for the creation of all kinds of shapes. The basic sine function can be used to transform a line or surface on the XZ plane into an oscillating wave as follows (see also *Anatomy of a Sine Wave Expression* p. 63):

The example shown is set to:

```
Pos Y = sin ( $TX * 1080 ) * 0.3 + 0.4
```

The general form for a geometric sine wave is:

```
Pos Y = sin ( $TX * frequency ) * amplitude + offset
```

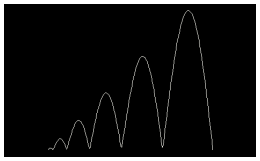
For wave motion, you could use this segment expression:

```
geo1/ty = sin ( $T * frequency ) * amplitude + offset
```

where:

$TX	The basis for the wave: to animate the geometry of the wave, this could be set instead to (*$TX* + *$F*).
frequency	Affects the number of waves. This value could be set to the bounding box position, *$BBX*, to make the waves more frequent towards the end of the line.
amplitude	Affects the height of waves. This value could be set to the bounding box position, *$BBX*, to flatten the waves toward the end of the line; or the current frame, *$F*, to make the waves larger as the animation progresses.
offset	Translates the waves back and forth.

6.3 BOUNCES AND PEAKS

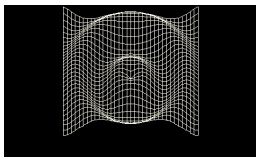


Use the absolute value of the sine function to get bounces and peaks:

```
pos y = abs ( sin ($BBX^0.5 * 1080) * $BBX)
```

Use the negative of the absolute value to invert the wave.

6.4 RIPPLES



To get ripples radiating from a centre, base the sine function on the distance of the point to the centre of the surface:

```
pos y= sin(sqrt(($BBX-0.5)^2+($BBZ-0.5)^2)*1080)
```

To animate this, add a time-based variable to the expression:

```
sin(sqrt(($BBX-0.5)^2+($BBZ-0.5)^2)*720+$F*4)
```

6.5 ARCS

Arcs, Circles, Ellipses and Spirals can be made by modifying a point's position in two axes using a sine and a cosine function:

```
pos x = cos ($PT * arcangle) * radiusX + offsetX
pos y = sin ($PT * arcangle) * radiusY + offsetY
```

or

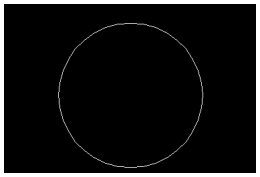
```
geo1/tx = cos ($T * arcangle) * radiusX + offsetX
geo1/ty = sin ($T * arcangle) * radiusY + offsetY
```

where:

The angle subtended by this expression equals *arcangle* * total number of points.

radiusX and *radiusY* are the radius of the circle. If differing, then an ellipse is generated. If set to a variable such as *$PT* then a spiral may be generated. *offsetX* and *offsetY* translate the arc.

6.6 CIRCLE



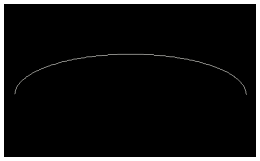
```
pos x = cos ($PT*2)
pos y = sin ($PT*2)
```

This expression is useful as a segment expression when you want an object to go in the path of a circle and maintain the same orientation of the object (if you use a rotation channel, the orientation will rotate as well). i.e.

```
tx = cos ($T * 180)
ty = sin ($T * 180)
```

For example, using a rotation channel, an arrow would travel a circular path with the arrow-head rotating as well; whereas using this expression in the *tx* and *ty* channels, the arrowhead would always point in the same direction.

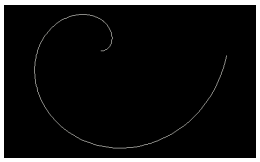
6.7 ELLIPTICAL ARC



Change values of the multiplier (2 and 0.7) to change the aspect ratio of the arc.

```
pos x = cos ($PT) * 2
pos y = sin ($PT) * 0.7
```

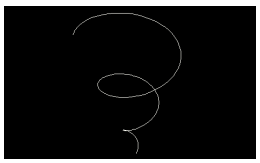
6.8 LOGARITHMIC SPIRAL



This creates a logarithmic spiral, reminiscent of nautilus shells.

```
pos x = cos ($PT * 2) * $PT / 200 - 0.2
pos y = sin ($PT * 2) * $PT / 200 + 0.2
```

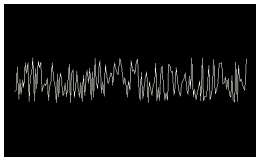
6.9 3D SPIRAL



The height of the spiral versus the spiral size can be controlled by the Position in Z.

```
pos x = cos ($PT * 5) * $PT / 500
pos y = sin ($PT * 5) * $PT / 500
pos z = $PT / 200
```

6.10 RANDOMIZE



```
pos y = $TY + rand ($PT) * 0.2
```

Often it is desirable to modify point positions by a small amount; this is done with the random function which generates a random value between 0 and 1 based on an input number. This value can then be added onto the current position to modify it.

6.11 CLAMPING

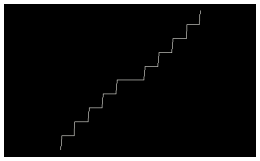
```
pos y = clamp ( (sin($TX * 1080) *0.3), -0.1, 0.28)
```

To ensure that a value stays in a certain range, use clamping. Clamping will use a specified minimum or maximum if the number generated is below or above the clamp value. For example:

```
clamp ($TX, minvalue, maxvalue)
```

Clamps the x position to be between the *minvalue* and the *maxvalue*.

6.12 STEP

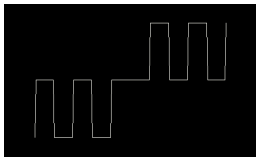


```
pos y = int ($TX * 10) * 0.1
```

General form:

```
pos y = int ($TX * frequency) * amplitude
```

6.13 SQUARE WAVE

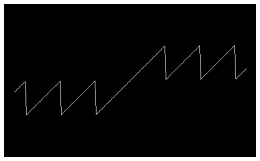


```
pos y = int ($TX * 10) % 2 * 0.3
```

General form:

```
pos y = int ($TX * frequency) % 2 * amplitude
```

6.14 SAWTOOTH WAVE

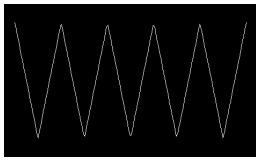


```
pos y = $TX  % 0.15
```

General form:

```
pos y = $TX % frequency * amplitude
```

6.15 TRIANGULAR WAVE

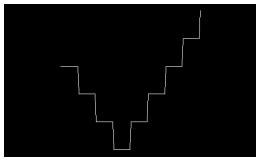


```
pos y = abs ( $BBX * 5 % 1 - 0.5 )
```

General form:

```
pos y = abs ( $BBX * frequency % amplitude - .5 * amplitude )
```

6.16 STEPPING TRIANGULAR WAVE



```
pos y = abs(int($BBX*8) % 12 - 3 )* 0.2
```

General form:

```
pos y = abs(int($BBX * frequency) % step-0.5*step)*amplitude
```

7 DEFORMATION EXPRESSIONS

7.1 INTRODUCTION

A Point OP referring to spare channels can be used to deform geometry for such effects as shear, taper, squash and bend.

These effects can also be achieved with the Twist OP.

SHEAR

Shears geometry in any direction, X, Y, or Z. To shear about the bottom of an object in the X direction, first add a Point SOP and set it's display flag, then add the channel *spare1* (use *Edit > Add User Defined Channels...*). Then in the Point SOP, change the Translate X to:

```
Pos X = $TX + $BBY * (ch("spare1"))
```

The amount of shearing can be animated using the spare channel.

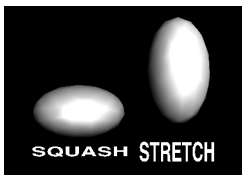
TAPER

Taper is similar to shear; you can taper in any direction. To taper in the X direction, use a Point OP, and add a spare channel. Change *pos x* to any of the following:

```
Pos X = $TX * ( ch("spare1") ^ $BBY )
Pos X = $TX * ( 1 / ch("spare1") ^ $BBY )
Pos X = $TX * ( 1 - ($BBY * ch("spare1") -1 ))
```

The amount of taper can be animated using the spare channel.

Note: For this expression, *spare1* must be greater than 0 or an error will occur.



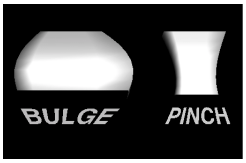
SQUASH AND STRETCH

Squash and stretch is a traditional animation name given to the physical property of conservation of volume. That is if an object scales up in one direction (stretch) there will be a corresponding scale down in the other direction.

A simple way of doing squash and stretch is with a Transform SOP and using only the scaling channels: *sx*, *sy*, and *sz*. Use the *sy* channel to scale the object using the any function (*ease()*, *bezier()*, etc.) then in the *sx* and *sz* channel segment use:

```
Scale X = 1/sqrt( ch("sy") )
Scale Z = 1/sqrt( ch("sy") )
```

Now by animating the *sy* channel, the relative volume of the object is preserved.



BULGE AND PINCH

To get a squash and stretch where the only the middle of the object is affected (so it bulges or pinches), use a Point OP with *spare1* channel as your controlling channel. Change *pos x* and *pos z* to:

```
Pos X = $TX + sin($BBY*180)*(1/(ch("spare1"))-1)*($BBX-0.5)
Pos Z = $TZ + sin($BBY*180)*(1/(ch("spare1"))-1)*($BBZ-0.5)
```

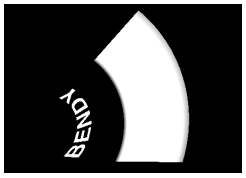
To get a smooth bulge/pinch, precede the Point OP with a Refine OP, and increase the U and V Divisions over a range of 0 - 1. You can also use a Twist or Lattice OP to get bulging effects.

TWIST

To twist an object around its centre, append a Point OP and create a spare channel. Change the *pos x* and *pos z* to:

```
Pos X = ($TX-$CEX) * cos( ch("spare1") * ($BBY-0.5)) -
 ($TZ-$CEZ) * sin( ch("spare1") *($BBY-0.5)) + $CEX
Pos Z = ($TX-$CEX) * sin( ch("spare1") * ($BBY-0.5)) +
 ($TZ-$CEZ) * cos( ch("spare1") *($BBY-0.5)) + $CEZ
```

For greater control, precede the Point OP with a Refine OP, and increase the U and V Divisions over a range of 0 - 1. You can also use a Twist or Lattice OP to get twist effects.



BEND

To bend an object, use a Point SOP and create a spare channel. Change the *pos x* and *pos z* to:

```
Pos X = $TX * cos($BBY*ch("spare1")) - $TY *
 sin($BBY*ch("spare1"))
Pos Y = $TX * sin($BBY*ch("spare1")) + $TY *
 cos($BBY*ch("spare1"))
```

As with twist, you may want finer control by preceding the Point OP with a Refine OP, and increase the U and V Divisions over a range of 0 - 1. You can also use a Twist or Lattice OP to get bend effects.

8 SEGMENT EXPRESSIONS

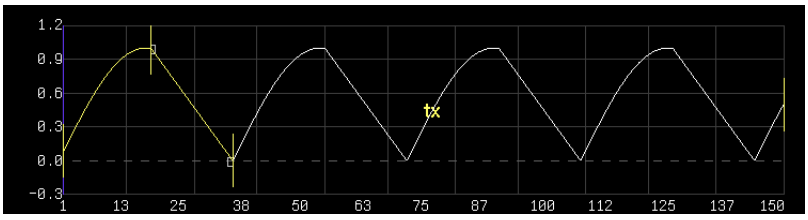
There are many built-in expressions specifically designed for segments. You should have already noticed that there are many variants of the *ease()* function. There are also *linear()*, *constant()*, *spline()* and *bezier()*; which are quite versatile. In particular, with *bezier()*, you can adjust the tension of the Bezier by dragging the handles of the Bezier, and the slope (by clicking and dragging on the tension line). In addition there are some specialized functions:

8.1 EXAMPLE FOR REPEAT()

REPEAT

Use the *repeat* function to copy or repeat repetitive motion.
The basic form for the repeat function is:

```
repeat ( start_keyframe, end_keyframe )
```



If you have keyframes at 0, 22, and 36, and you want to repeat the curve from 0 to 36, you select the segment from frames 36- , and enter:

```
repeat (0,36)
```

This works within a channel, without referencing another channel.

Note: Your repeat range must be outside the range of the currently selected segment. For example, if you have keyframes at frames 0, 22, and 36, you cannot select the segment from 0-22 and assign a *repeat()* function with values within the range of 0-22, because it would be trying to repeat values of itself that have not yet been determined.

9 ANATOMY OF A SINE WAVE EXPRESSION

9.1 INTRODUCTION

One of the most commonly used functions is the sine function. In general usage, it takes the form of:

sin (frequency * phase) * amplitude + offset

Whenever you need to cause something to oscillate back and forth over time, you can use a sine expression. For example, to get a bouncing ball, you can enter a sine expression in the ball's Translate Y parameter. You can control how the ball bounces by changing the various aspects of the sine function.

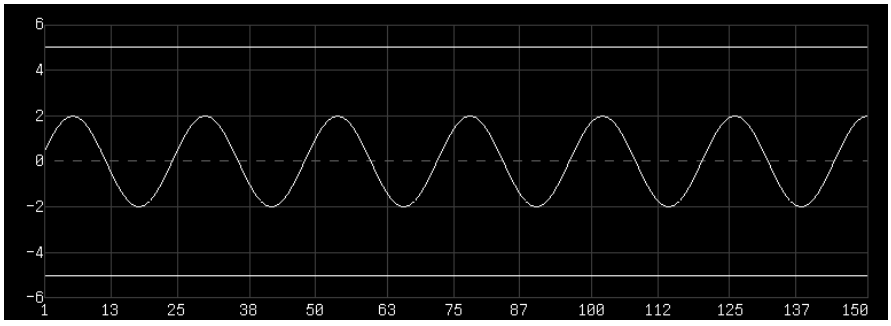
Enter the following expression in the Translate Y field of *geo1* to observe the changes in behaviour by using variations of the formula. Click *Play* to see the effect.

```
sin( $F*15 * 1 ) * 2 + 0
```

9.2 EXAMPLES

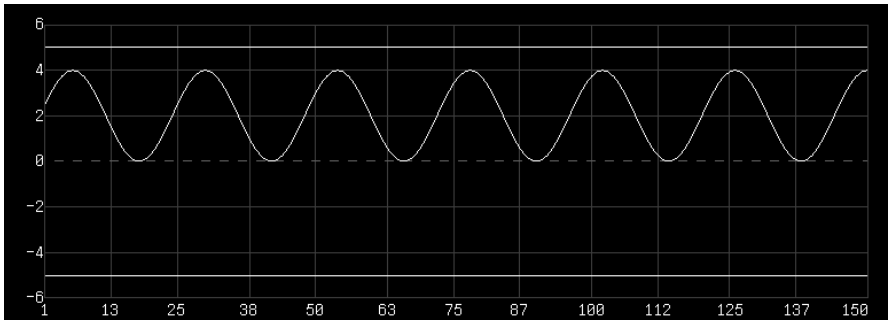
PLAIN SINE EXPRESSION

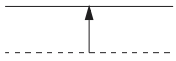
```
sin( $F*15 * 1 ) * 2 + 0
```



WITH OFFSET

```
sin( $F*15 * 1 ) * 2 + 2
```

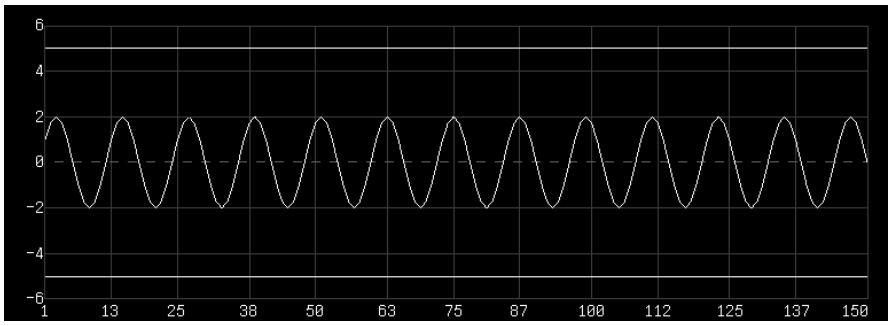




Wave is offset. Axis moves from a value of 0 to 2

INCREASED FREQUENCY

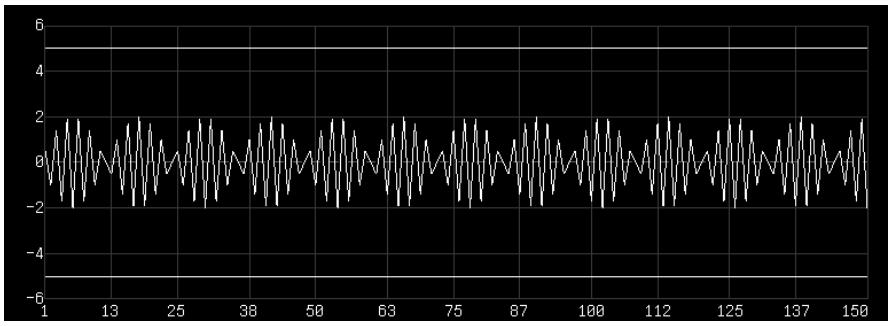
```
sin( $F*30 * 1 ) * 2 + 0
```



Increased frequency bunches the waves closer together

WITH FREQUENCY MODULATION (PHASE)

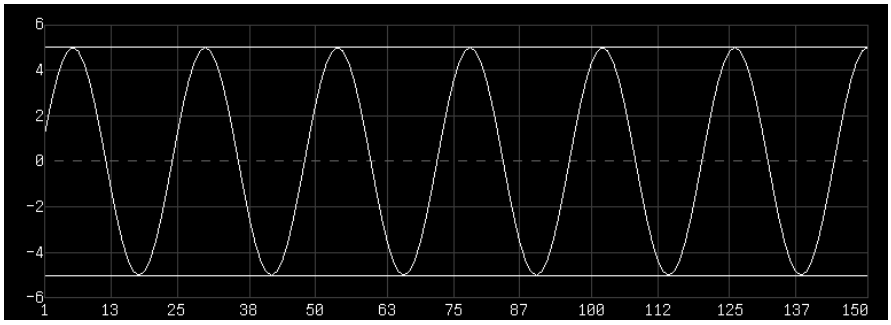
```
sin( $F*15 * 35 ) * 2 + 0
```

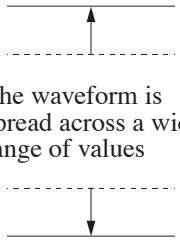


The two frequencies combine to form this type of pattern

SCALED AMPLITUDE

```
sin( $F*15 * 1 ) * 5 + 0
```

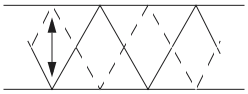




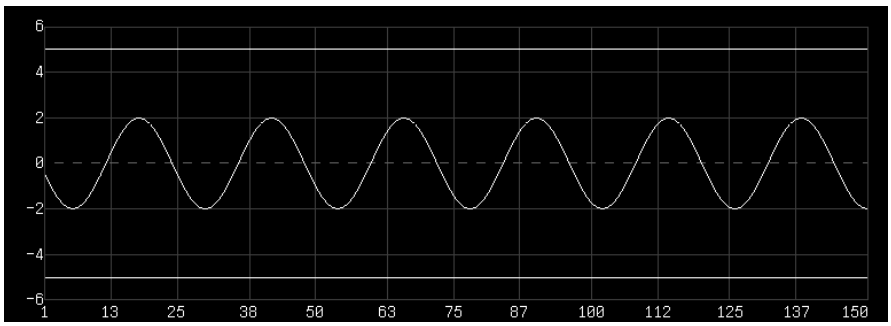
The waveform is spread across a wider range of values

INVERTED

```
sin( $F*15 * 1 ) * -2 + 0
```



The waveform is inverted



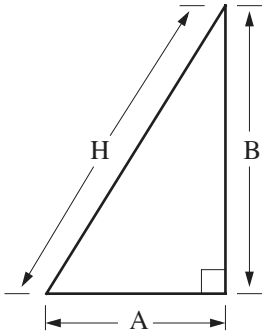
10 PYTHAGOREAN THEOREM

10.1 INTRODUCTION

It sometimes comes up that you need to calculate the length of a side of a triangle given the lengths of two other sides of a triangle. The Pythagorean theorem solves exactly that problem.

10.2 DISCUSSION

This simple formula is the key to finding the lengths of the sides of a triangle:



$$H^2 = A^2 + B^2$$

By rearranging of the formula, we can derive:

$$H = \sqrt{A^2 + B^2}$$

$$A = \sqrt{H^2 - B^2}$$

$$B = \sqrt{H^2 - A^2}$$

Using these simple formulas, we can calculate the hypotenuse or sides of any right angle triangle. These formulae are very useful in any number of applications using expressions.

Tip: A Houdini function exists to automatically calculate the three-dimensional hypotenuse for you. The *length()* function returns the value of:

```
sqrt(x² + y² + z²)
```

which is the length of the hypotenuse in 3D space. It's syntax is:

```
length(floatx, floaty, floatz)
```

See the *Expressions* section of the Reference manual for this, and many other useful built-in functions.

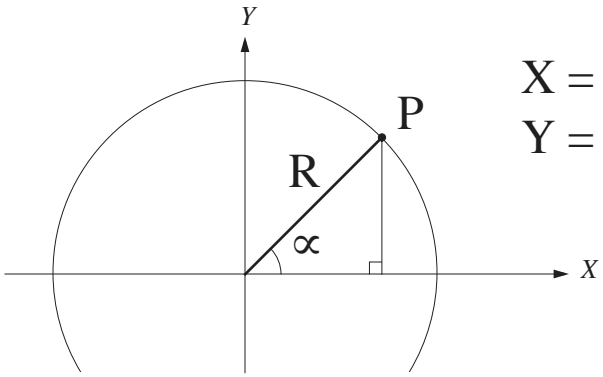
11 SINE COSINE AND TANGENT

11.1 INTRODUCTION

What do we do when we only know the length of one side and an angle? We can derive the lengths of the other sides and the other angles using Sine, Cosine and Tangent. This can also help us translate between Cartesian (XYZ) and Polar (Radius and Angle) systems.

11.2 DISCUSSION

With some simple high school trigonometry, we can solve these problems. The following formula is the key to finding the lengths of the sides of a triangle:



$$X = R * \cos(\propto)$$
$$Y = R * \sin(\propto)$$

By rearranging the formula, we can derive:

$$\propto = \text{acos}(X/R)$$
$$\propto = \text{asin}(Y/R)$$
$$\propto = \text{atan}(X/Y)$$
$$R = X / \cos(\propto)$$
$$R = Y / \sin(\propto)$$

To calculate the X and Y coordinates of a point along the circumference of a circle we can use the above two equations, where *R* is the radius of the circle (or distance of some object in space from another object), and ∝ is the angle formed by the X-axis, the centre of the circle, and point P.

Using these two simple formulas, we can calculate the X and Y coordinates for any point along the perimeter of a circle. These formulae are also very useful in any number of applications using expressions.

12 MATCHING HOUDINI CAM TO THE REAL WORLD

12.1 INTRODUCTION

This section describes how to match Houdini camera lenses to a real world camera, using some basic camera mathematics.

12.2 DISCUSSION

ANGLE OF VIEW AND HOUDINI FOCAL LENGTH

You can obtain a good fit between the Houdini camera and a real world camera by matching a measured lens' horizontal angle of view, and deriving a Houdini focal length value that reproduces it with the default aperture 41.4214.

This of course ignores pin-cushion and barrel distortion which must be set using shots of grids and render tests. You should measure the horizontal angle of view with respect to the camera's TV safe transmitted reticule aperture.

SCANNED FILM IMAGES

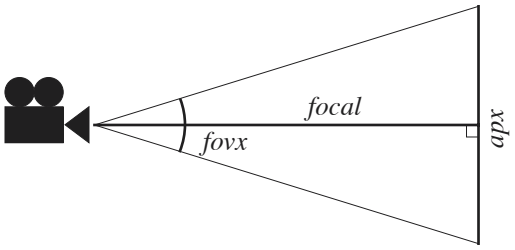
For scanned film images, you can simply divide the pixel width of the scanned image by the pixels/mm for the scanner, and plug this number into the aperture channel. Then set the focal length to the live action focal lens.

12.3 FORMULAS

There are some basic mathematics we can use to relate a real world camera to Houdini cameras. On the following page are some useful formulas.

Note: Notice how *fovx* is not dependent on *resx*, *resy* or, *aspect*, but *fovy* is. You can verify this for yourself by attaching a unit-spaced grid to the camera at the focal length distance and the number of units in the X direction will be exactly equal to the aperture. The number of grid units in the y direction (*apy*) will be dependent on *resx*, *resy* and *aspect*.

HOUDINI'S CAMERA



VARIABLE DEFINITIONS & CHANNEL DEFAULTS

Variable	Definition	Channel / Default
fovx	field of view in X	- / -
fovy	field of view in Y	- / -
apx	aperture in X	aperture / 41.4214
apy	aperture in Y	- / -
focal	focal length	focal / 50
resx	pixel resolution in X	resx / 320
resy	pixel resolution in Y	resy / 243
asp	pixel aspect X/Y	aspect / 1

THREE KEY RELATIONSHIPS

tan(fovx/2) = (apx/2) / focal
tan(fovy/2) = (apy/2) / focal
apx/apy = (resx * asp) / resy

USEFUL EQUATIONS DERIVED FROM THE ABOVE

fovx = 2 * atn((apx/2) / focal)
apy = (resy*apx) / (resx*asp)
fovy = 2*atan((apy/2) / focal)

SOLVING FOR THE ABOVE GIVEN THE HOUDINI DEFAULTS

fovx	= 2*atan((apx / 2) / focal)
	= 2*atan((cam1/aperture / 2) / cam1/focal)
	= 2*atan((41.4214 / 2) / 50)
	= 45°
apy	= (resy * apx) / (resx * asp)
	= (243 * 41.4214) / (320 * 1)
	= 31.454376
fovy	= 2*atan((apy/2) / focal)
	= 2*atan((31.454376 / 2) / 50)
	= 34.9213°

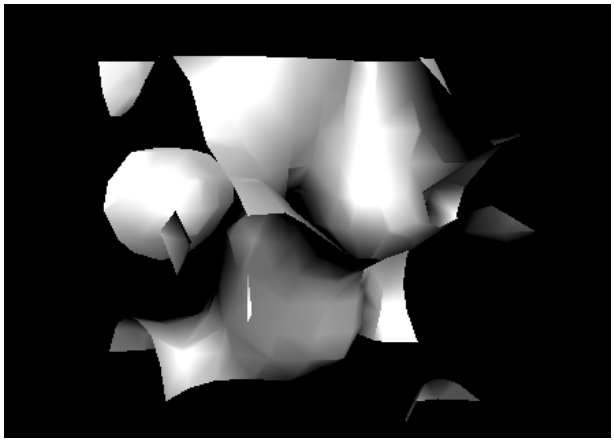
13 ISO SURFACE EXAMPLES

13.1 INTRODUCTION

With the Iso surface OP, we can create a multitude of interesting geometry based strictly on mathematical expressions. The three examples here will illustrate how to use this SOP to make a wedge of Swiss cheese, a flame (without a Particle OP), and a star.

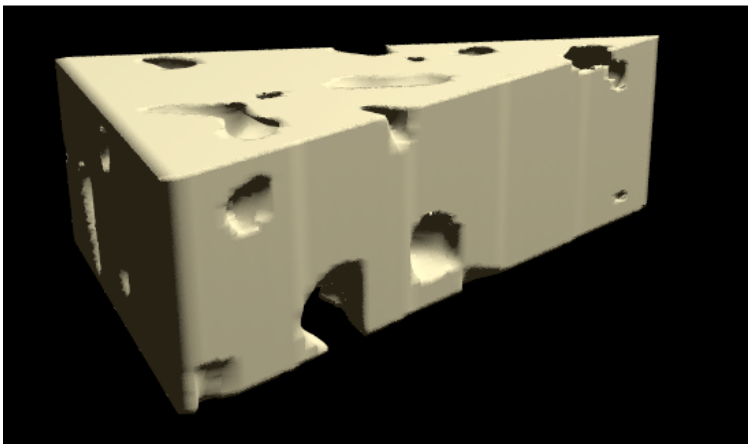
The action of the Isosurface OP is conceptually simple – it takes a user specified expression in R3 (a mathematical term meaning, "having three dimensions, each taking a Real value"), and creates a surface where the function goes from being positive to being negative. In the case of the default expression ($\$X^2 + \$Y^2 + \$Z^2$), the expression is less than zero within a unit sphere, and greater than zero outside. As the OP cooks, it marches through the bounding volume specified (by default from -1 to +1 in X, Y and Z), and creates geometry where the expression equals zero.

This may seem like a difficult way to define a sphere, but there's much potential beyond this simple example using the rich array of mathematical functions (see the Expressions section). A simple illustration is with the *noise()* function. Try inputting the following expression. Intriguiging?



Expression: *noise($X, $Y, $Z)*

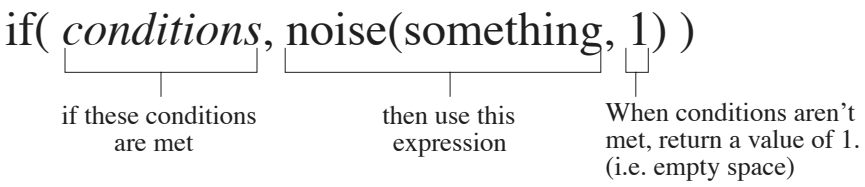
13.2 A SLICE OF SWISS CHEESE

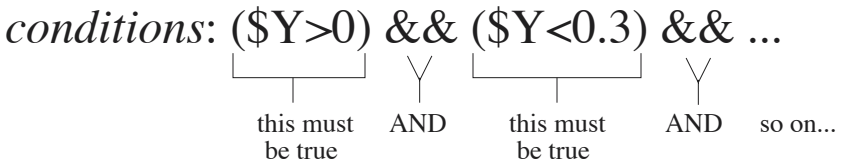


An important thing to know when sculpting with expressions is that the *if()* function is a valid component to use in creating an isosurface. This allows you to construct your function, and thus your geometry in a piecewise fashion. This has been made use of in the following example – a wedge of cheese. At first glance, this function can seem somewhat daunting, but when broken down piece by piece, it's easier to understand. Try entering this *Expression* (should be entered as a single line):

```
if( ($Y>0) && ($Y<0.3) && ($X>-$Z/3) && ($X<$Z/3) &&
  ($Z<.9), noise($X*10,$Y*10,$Z*10)-0.3, 1)
```

This can seem a bit overwhelming at first, but breaking it down piece by piece, it becomes understandable:





This function says that it should define a surface according to a *noise()* function if some things are true, and should equal to 1 if any of those things aren't true. Since geometry exists only where the whole expression is zero, no geometry will exist if the criteria are false. In this way, we set up boundary conditions for the geometry we are create. What are these conditions?

Several things; there are five conditions being met – each seperated by an "and" operator "&&". This means that each of the conditions; ($Y>0), ($Y<0.3), etc. must be satisfied in order for geometry to be created. If all five of these criteria are met, then it is up to the *noise()* function to determine what the geometry looks like.

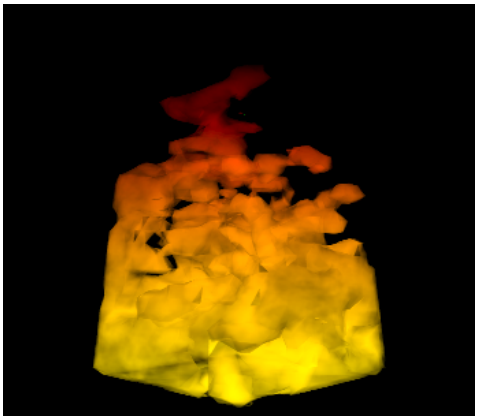
The first two conditions involving the $Y variable cause the upper and lower bounds of the geometry (in the Y direction) to be constrained between 0 and 0.3. This is what causes the upper and lower surfaces of the cheese wedge to be defined.

The next two criteria relating $X to $Z define the two side faces of the cheese. By relating the X component to the Z component, we can define a planar surface that is on an angle to any of the axis planes.

The last component, ($Z<0.9) defines the back plane of the triangular profile of the cheese. By altering the boundary conditions, we can alter the shape of the cheese.

If any of these conditions aren't met, the expression returns a value of 1, and no geometry results. If all of these conditions are met, the expression returns a noise function which can have a value between -1 and positive 1. If the noise function returns a positive value, then a hole is defined in the cheese. If the noise function returns a value less than zero, then cheese is defined.

13.3 A FLAME



Below is an expression to define a flame:

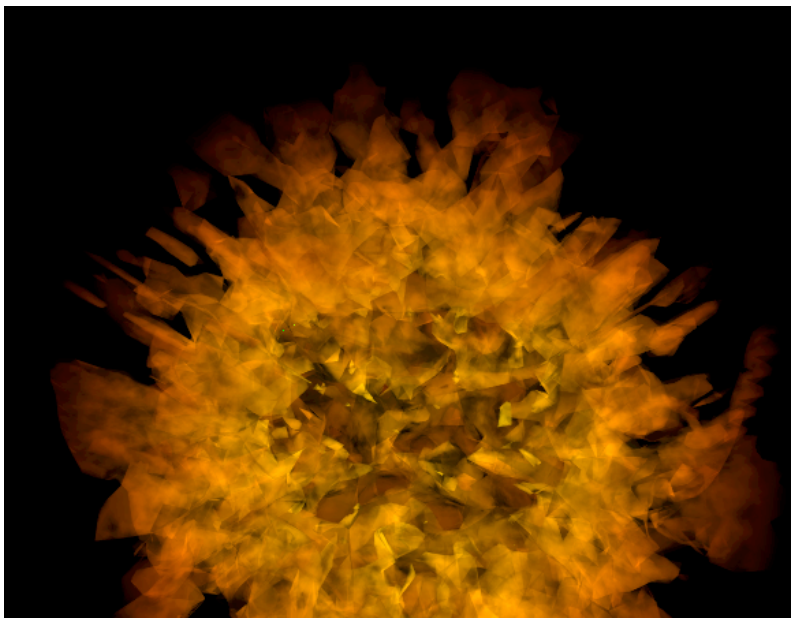
```
if( ($X>-.9) && ($X<.9) && ($Z>-.9) && ($Z<.9) &&
  ($Y>-.9), 0.2+turb($X,$Y-$T/2,$Z,2)-$Y/8, -1)
```

In this case, the conditions of the *if()* function limit the flame to exist within a square. The interesting features of this expression are the taper which has been introduced, and the animation of the Y parameter of the noise function.

The offset of "-$Y/8" at the end of the noise function (in this case, turbulence) has the effect of causing the noise function, which would normally return a value between -1 and +1 to be more and more biased to the negative as the coordinate increases in Y. That is, the surface, defined where the function as a whole returns a zero value, eventually lies totally on the negative side of that threshold at a great enough value of Y. The net result of this is that the thickness of the flames diminishes to nothing as the noise function is evaluated at greater values of Y.

The second interesting feature is that the Y value of the noise function is itself being animated by means of the time variable – $T. This causes the features of the noise function to translate upward in Y over time. The overall result is that the flame rises and diminishes into nothingness.

13.4 A STAR



Here is an expression to define a flaming star:

```
noise(($X*$X+$Y*$Y+$Z*$Z)^.5/3, atan2($Y,$X)/8,
  atan2($Z,$Y)/8)+(1-($X*$X+$Y*$Y+$Z*$Z)^.5/5)
```

You must also set the following parameters:

```
Minimum Bounds:      -10 -10 -10
Maximum Bounds:       10  10  10
Divisions:            40  40  40
```

In this equation, The rectangular (i.e. Cartesian) coordinate system has been moved to a spherical (i.e. polar) coordinate system. Where the flame diminished as the coordinate increased in Y, the star diminishes as the distance from the centre increases.

Apply a colour ramp and render to see a flaming ball of fire.

14 CUSTOM EXPRESSION FUNCTIONS

Houdini has a wide variety of expression functions built-in, and they will cover almost all your requirements. However, there come times when you may need a certain custom expression function that Houdini doesn't have. Houdini's *Expression Function Language* lets you program your own expression functions.

The expression functions are loosely based on the C language. The language is simpler in many respects and different in many others. The expression function language allows you to extend the Houdini expression language by writing your own functions which are integrated seamlessly into the internal expression language.

The expression language includes support for string, vector and matrix types. These can be used as return codes, parameters or variables inside the function.

You can also edit Expression Functions via the *Edit > Edit Aliases/Variables...* dialog. Once the dialog appears, click on the *Expressions* button to view the currently available custom expression functions. Select one, and click the *Edit* button to edit the function in a text editor (default = *vi* . Set this with the EDITOR environment variable – see *Reference > Interface > Edit Fields* p. 213 for how to change this).

14.1 BASIC SYNTAX

Comments are denoted by a '#' character anywhere within the line.

A function is declared as:

```
[Type]
functionName([Parameter], [Parameter]...)
{
    Scope_Block
}
```

The *Type* specifies the return type of the function and can be one of:

- float
- string
- vector
- matrix

If the return type is not defined, it is assumed to return a float.

14.2 PARAMETERS TYPES

Each parameter is defined as:

PARAMETER := [Type] parameter_name

If no *Type* is specified for a parameter, it is assumed to be a float parameter.

TYPES OF STATEMENTS

The body of the function is defined as a series of statements. There are several types of statements allowed by the function language:

Expession	<a single expression>
Operator	= | += | ++ | -- | -= | *= | /= | %=
Assignment	Symbol Operator Expression ;
For	for (Assignment ; Expression ; Assignment) Scope_Block
While	while (Expression) Scope_Block
If	if (Expression) Scope_Block [else Scope_Block]
Return	return Expression
LoopGoto	continue ; | break ;
Statement	Assignment | For | While | If | Return | LoopGote
Block	{ Scope_Block }
Scope_Block	Statement | Block

14.3 EXAMPLES

Following are several examples of custom expression functions.

FIND MINIMUM VALUE

```
#  Function to find the minimum value of two
#  floating point numbers

min(v1, v2)
{
  if (v1 < v2)         return v1;
  else                 return v2;
}
```

REVERSE ORDER OF STRING

```
#  Function to reverse the order of a string

string
strreverse(string in)
{
  float                len = strlen(in);
  string               result = "";

  for (src = len-1; src >= 0; src--)
      result += in[src];

  return result;
}
```

FIND MINIMUM ELEMENT

```
#  Example to find the minimum element in a vector

float
vecmin(vector vec)
{
  min = vec[0];
  for (i = 1; i < vsize(vec); i++)
      if (vec[i] < min)
                       min = vec[i];
  return min;
}
```

TRANSFORM A VECTOR

```
#  Example to transform a vector into the space
#  of an object passed in.

vector
opxform(string oname, vector v)
{
  matrix               xform = 1;
  if (index(oname, "/obj/"))
       xform = optransform(oname);
  else xform = optransform("/obj/"+oname);
  return v * xform;
}
```

FIND DISPLAY OBJECTS

```
#  Example to find all objects which have their
#  display flag set

string
opdisplay()
{
  string               objects = run("opls /obj");
  string               result = "";
```

```
  nargs = argc(objects);
  for (i = 0; i < nargs; i++)
  {
      string           obj = arg(objects, i);
      if ( index(run("opset " + obj), " -d on") >= 0 )
                       result += " " + obj;
  }
  return result;
}
```

14.4 HOUDINI / HSCRIPT INTERFACE

In addition to use the *Edit > Edit Variables/Aliases...* dialog, Houdini also has several Textport commands to read and manage expression functions. These commands are:

EXLS

Usage: exls

List all the current expression functions.

See also: *excat*, *exedit*, *exread*, *exrm*

EXRM

Usage: exrm *pattern*

All expression functions matching the pattern will be removed.

See also: *excat*, *exedit*, *exls*, *exread*

EXEDIT

Usage: exedit *[pattern]*

This command allows the user to edit expression functions. If no pattern is specified, you can add new functions to the current list. If a pattern is specified, the functions which match the pattern will be edited.

Warning: If a function is renamed or removed from the edit session, this does *not* mean that the old function will be removed from the current function list. This must be done through the *exrm* command.

See also: *excat*, *exls*, *exread*, *exrm*

EXCAT

Usage: excat *[pattern]*

Displays the source to all expression functions in the current .hip file. If a pattern is specified, only function names matching the current pattern will be listed.

See also: *exedit*, *exls*, *exread*, *exrm*

EXREAD

Usage: exread *diskfile [diskfile2...]*

This command can be used to source in external files of expression functions.

See also: *excat*, *exedit*, *exls*, *exrm*