Expr family objects by Shahrokh Yadegari

To see a directory listing of downloadable files, which also inlcudes older releases, click here .

Back to the Software Page

Expr, Expr~, Fexpr~

The **expr** family is a set of C-like expression evaluation objects for the graphical music language Pd and it is now part of the vanilla distribution.

New Additions in version 0.4

- Expr, expr~, and fexpr~ now support multiple expressions separated by semicolons which results in multiple outlets.
- Variables are supported now in the same way they are supported in C. Variables have to be defined with the "value" object prior to execution.
- A new if function if (condition-expression, IfTure-expression, IfFalse-expression) has been added.
- New math functions added.
- New shorthand notations for fexpr~ have been added.
    - $x ->$x1[0] $x# -> $x#[0]
    - $y = $y1[-1] and $y# = $y#[-1]
- New 'set' and 'clear' methods were added for fexpr~
    - clear - clears all the past input and output buffers
    - clear x# - clears all the past values of the #th input
    - clear y# - clears all the past values of the #th output
    - set x# val-1 val-2 ... - sets as many supplied value of the #th input;
        - e.g., "set x2 3.4 0.4" - sets x2[-1]=3.4 and x2[-2]=0.4
    - set y# val-1 val-2 ... - sets as many supplied values of the #th output;
        - e.g, "set y3 1.1 3.3 4.5" - sets y3[-1]=1.1 y3[-2]=3.3 and y3[-3]=4.5;
    - set val val ... - sets the first past values of each output; e.g.,
        - e.g., "set 0.1 2.2 0.4" - sets y1[-1]=0.1, y2[-1]=2.2, y3[-1]=0.4

expr runs in control rate and evaluates C-like expressions. See below for the list of operators . Multiple expressions separated by semicolon can be defined in a single expr object which result in multiple outlets. Expressoin are evaluated from right to left (which means that the last expression defined will be the first executed.) Access to inlets in **expr** take a few different forms:

$i1 - $i9	the first nine inlets taken as integers
$f1 - $f9	the first nine inlets taken as floats
$s1 - $s9	the first nine inlets taken as symbols (currently symbols are used for table lookups)

Tables and variables can be accessed the same way one dimensional arrays are accessed in C; for example, "valx + 10" will be evaluated to the value of variable 'valx' + 10 (variables have to be defined using the 'value' object) and "tabname[5]" will be evaluated to be the 5th element of table "tabname". The name of the table can be a variable as well; for example "$s2[5]" will be evaluated to be the 5 element of the array whose symbol has been passed in inlet 2.

Type conversion is done either automatically or explicitly by the use functions. See below for the list of functions .

expr~ is designed to efficiently combine signal and control stream processing by vector operations on the basis of the audio buffer size of the environment. The operations, functions, and syntax for **expr~** is just like **expr** with the addition the **$v** variable for signale input. The accepted inlets for expr~ are as follows:

$i1 - $i9	the first nine inlets taken as integers
$f1 - $f9	the first nine inlets taken as floats
$s1 - $s9	the first nine inlets taken as symbols (currently symbols are used for table lookups)
$v1 - $v9	the first nine inlets taken as signals (vectors)

The result of **expr~** is a vector. The inlet has to be a vector and type conversions are done either automatically or by the use of functions .

Note for MSP users : Currently in the MSP version all signal inputs should come first followed by other types of inlet. (There seems to be no way of mixing signal and other types of inlets in their order in Max/MSP, if you know otherwise , please let me know.) This means that signal inlets cannot be mixed with other types of inlets. For example, "**expr~ $v1*$f2*$v3** " is not legal. The second and third inlet should be switched and "**expr~ $v1*$v2*$f3**" should be used.

fexpr~ object provides a flexible mechanism for building FIR and IIR filters by evaluating expressions on a sample by sample basis and providing access to prior samples of the input and output audio streams. When fractional offset is used, fexpr~ uses linear interpolation to determine the value of the indexed sample. **fexpr~** evaluates the expression for every single sample and at every evaluation previous samples (limited by the audio vector size) can be accessed. $x is used to denote a singnal input whose samples we would like to access. The syntax is $x followed by the inlet number and indexed by brackets, for example $x1[-1] specifies the previous sample of the first inlet. Therefore, if we are to build a simple filter which replaces every sample by the average of that sample and its previous one, we would use "**fexpr~ ($x1[0]+$x1[-1])/2** ". For ease of when the brackets are omitted, the current sample is implied, so we can right the previous filter expression as follows: " **fexpr~ ($x1+$x1[-1])/2**". To build IIR filters $y is used to access the previous samples of the output stream.

$i1 - $i9	the first nine inlets taken as integers
$f1 - $f9	the first nine inlets taken as floats
$s1 - $s9	the first nine inlets taken as symbols (currently symbols are used for table lookups
$x1[n] -$x9[n]	accessing samples of the first nine signal inlets (vectors) , where 0 <= n < -{vector size }
$y[n]	accessing the output samples where 0 < n < -{vector size}

The operators (listed from highest precedence to lowest) are as follows:

~	One's complement
*	Multiply
/	Divide
%	Modulo
+	Add
-	Substract
<<	Shift Left
>>	Shift Right
<	Less than (boolean)
<=	Less than or equal (boolean)
>	Greater than (boolean)
>=	Greater than or equal (boolean)
==	Equal (boolean)
!=	Not equal (boolean)
&	Bitwise And
^	Exclusive Or
|	Bitwise Or
&&	Logical And (boolean)
||	Logical Or (boolean)

All expr family objects support a variety of functions as follows:

Functions	# of Args	Description
if()	3	conditional - if (condition, IfT rue-expr, IfFalse-expr) - in expr~ if 'condition' is a signal, the result will be determined on sample by sample basis (added in version 0.4)
int ()	1	convert to integer
rint ()	1	round a float to a nearby integer
float ()	1	convert to float
min ()	2	minumum
max ()	2	maximum
abs()	1	absolute value (added in version 0.3)
if()	3	conditional - if (condition, IfT rue-expr, IfFalse-expr) - in expr~ if 'condition' is a signal, the result will be determined on sample by sample basis (added in version 0.4)
isinf()	1	is the value infinite (added in version 0.4)
finite()	1	is the value finite (added in version 0.4)
isnan	1	is the value non a number (added in version 0.4)
copysign()	1	copy sign of a number(added in version 0.4)
imodf	1	get signed intergar value from floating point number(added in version 0.4)
modf	1	get signed fractional value from floating-point number(added in version 0.4)
drem	2	floating-point remainder function (added in versio n 0.4)
power functions		
pow ()	2	raise to the power of {e.g., pow(x,y) is x to the power of y}
sqrt ()	1	square root
exp()	1	e raised to the power of the argument {e.g., exp(5.2) is e raised to the power of 5.2}
ln() and log()	1	natural log
log10()	1	log base 10
fact()	1	factorial
erf()	1	error function (added in version 0.4)
erfc()	1	complementary error function (added in version 0.4)
cbrt()	1	cube root (added in version 0.4)
expm1()	1	exponential minus 1 (added in version 0.4)
log1p()	1	logarithm of 1 plus (added in version 0.4)
ldexp()	1	multiply floating-point number by integral power of 2 (added in version 0.4)
Trigonometric		
sin()	1	sine
cos()	1	cosine
tan()	1	tangent
asin()	1	arc sine
acos()	1	arc cosine
atan()	1	arc tangent
atan2()	2	arc tangent of 2 variables
sinh()	1	hyperbolic sine
cosh()	1	hyperbolic cosine
tanh()	1	hyperbolic tangent
asinh()	1	inverse hyperbolic sine
acosh()	1	inverse hyperbolic cosine
atan()	1	inverse hyperbolic tangent
floor()	1	largest integral value not greater than argument (added in version 0.4)
ceil()	1	smallest integral value not less than argument (added in version 0.4)
fmod()	1	floating-point remainder function (added in version 0.4)
Table Functions		

size()	1	size of a table
sum()	1	sum of all elements of a table
Sum()	3	sum of elemnets of a specified boundary of a table