A General Filter Design Language with Real-time Parameter Control in Pd, Max/MSP, and jMax

Shahrokh Yadegari
sdy@ucsd.edu
Center for Research in Computing and the Arts
University of California, San Diego

Abstract

Most signal processing environments for computer music, such as Pd, Max/MSP, and jMax, transfer audio data among their objects by vectors (blocks). In such environments, to implement Infinite Impulse Response (IIR) filters one either has to set the block-size to 1 or to write an external object which embeds the filter operations. Neither of these solutions are simple or trivial. In this paper we present the *fexpr*~ object which provides a general and flexible language for designing filters by simply entering the filter expressions at object creation time and controlling their parameters in real-time within the host environment. *Fexpr*~ also allows for multiple interdependent filters to be defined in a single object, and thus, it can be used for finding numerical solutions to differential equations (difference equations). The implementation and the filter definition syntax of the object are discussed along with a number of examples.

1 Introduction

This paper describes an external object, called *fexpr*~, implemented for Pd[5], Max/MSP[6], and jMax[1] environments, providing a general and flexible filter design mechanism with real-time parameter control. These signal processing environments transfer audio among their objects in blocks. Implementing IIR filters or generating numerical solutions for differential equations require specific calculations for every sample in which the result of previous calculations may be used. In an environment where data is transfered among objects by blocks, to implement such algorithms one is forced to write an external object or set the operating block-size to 1. Both of these solutions have serious drawbacks. Writing external objects requires knowledge of a programming language, such as C, and often involves a learning curve for the uninitiated to the internal workings of the environment. Providing real-time control of the filter parameters in an external object can prove to be time consuming, and recompilation of the object is required for every change to the filter definition or control mechanism of its parameters. Setting the environment block-size to 1 will make the patch creation difficult and inefficient. The *fexpr*~ object, presented in this paper, provides a flexible mechanism for defining filters as simple expressions whose parameters are regular control or audio streams of the environment. Thus, one is able to control the parameters of the filters in real-time within the processing environment.

2 Expr Family Objects

The author implemented the *expr* object in the original Macintosh version of MAX ("The Patcher") for expression evaluation of control streams. The expression syntax for *expr* is very similar to expression syntax of the C programming language.[3, p53] (None of the store, typecasting, nor any of the following operators "`-> . -- ++ ?:`" are supported at this time.) The rules for precedence of operators are also the same as those defined in the C language. In addition to arithmetic operations, *expr* supports access to variables (defined in the host environment by the *value* object) and tables (similar to arrays in C). It also provides a number of functions which include all the functions of C language's math library and a conditional (*if()*) function.

Expr~ is an extension of *expr* which efficiently combines signal and control stream processing by vector operations on the basis of the audio block-size of the environment. The outputs of *expr*~ are of

type signal. Typecasting is done by every operation where any data is turned into a buffer where necessary. Buffers are allocated at the time of expression evaluation and freed after processing of each block.

Fexpr~ is also an extension of *expr*. It is best to think of *fexpr~* as an *expr* which is evaluated for every sample. *Fexpr~* provides a syntax for accessing previous samples of the input streams as well as previous samples of the output streams in the filter expressions. One block of every input and output streams are buffered. All the *expr* family objects allow for definition of multiple expressions, separated by semicolon, which results in multiple outputs of the same type. This is specially needed when using *fexpr~* for finding numerical solutions to differential equations representing second (or higher) order dynamical systems with 2 (or more) variables.

3 Implementation

All *Expr* family objects parse and translate the given expressions into reverse polish notation at object creation time. The *expr* object evaluates its expressions any time a new value or a 'bang' is received in the first inlet. *Expr~* evaluates the expression(s) every time its service routine is called by the environment (normally at every block processing). Arithmetic operations and functions are efficiently performed on blocks where appropriate; therefore, in *expr~* every supplied expression is evaluated once every time the service routine is called. By contrast, *fexpr~* evaluates the expression(s) for every sample of every block, passing a current index number to all the functions which implement either the arithmetic operations or the supplied functions . The current index value is then used for determining the value of any indexed signal. A 4 point interpolation table lookup algorithm is used when a signal is indexed with a fractional value. *Fexpr~* supports a number of methods for starting or stopping the precessing, or for setting and clearing the previous values of the input or output signals.

4 Filter definition Syntax

In addition to access to global variables in the host environment, special variables are used for accessing input and output streams. Inlet values are denoted by the following syntax: `$T#` where, `T` specifies the type of inlet and #, the inlet number. Integer, float, and symbol inlets are available to all *expr* objects. For example, `$i1`, specifies the value of the first inlet as integer, `$f3`, the value of the third inlet as float, and `$s2[5]`, the value of the fifth element of an array specified by the value of the second inlet.

Signal inputs in *expr~* are specified by the `$v#` syntax where # specifies the inlet number. For example, the output signal of "`expr~ $v1*$f2`" is equal to the signal received in the first inlet attenuated by the floating point value received in the second inlet.

Signal inputs and outputs in *fexpr~* are specified by the `$x#[n]` and `$y#[m]` syntax respectively, where # specifies the signal inlet or outlet number, and `n` and `m` are the indexes for accessing the previous values of the signals. *Fexpr~* buffers one block of each of its inputs and outputs; therefore:

for `$x#[n]`, $0 <= n <= -blocksize$

for `$y#[m]`, $0 < m <= -blocksize$

A number of shorthand notations are available to make the filter definitions easier to code and read. The shorthand notation specified by (1) implies that when the inlet or outlet number is dropped the first inlet or the first outlet will be used. The shorthand notation specified by (2) implies that when an input signal is not indexed, the current sample is assumed, and when the output signal is not indexed the previous result (index of -1) is assumed. Applying both shorthands will result in the last shorthand (3).

`$x[n]` → `$x1[n]` `$y[n]` → `$y1[n]` (1)

`$x#` → `$x#[0]` `$y#` → `$y#[-1]` (2)

`$x` → `$x1[0]` `$y` → `$y1[-1]` (3)

5 Examples

In this section we present a number of examples which show various applications of the *fexpr~* object. The examples presented in this paper have been implemented in the Pd environment.

5.1 Linear Filters

If a linear, causal, and time-invariant IIR filter is mathematically described by the following equation:

$$y(n) = \sum_{i=0}^{M} a_i x(n-i) + \sum_{i=1}^{N} b_i y(n-i) \qquad (4)$$

the expression supplied to *fexpr~* for this filter would take the following form:

$$\sum_{i=0}^{B} a_i \texttt{ * \$x1[-i]} + \sum_{i=1}^{B} b_i \texttt{ * \$y1[-i]} \qquad (5)$$

where B is the block-size of the environment.

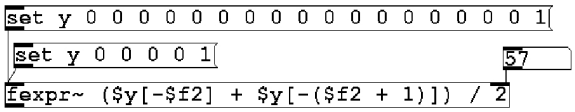
A simple FIR filter with a zero at π can be implemented by the following expression:

```
fexpr~ ($x1[0] + $x1[-1]) / 2
```

By applying the shorthands we can simplify the expression to the following:

```
fexpr~ ($x + $x[-1]) / 2
```

The following patch is an implementation of a Karplus-Strong string synthesis algorithm with simple parameters which can be controlled in real-time.[2]



The two 'set' methods set the previous values of the output for the current block and act as excitation methods. The first 'set' method sets `y1[-18]` to 1, which generates a string sound with a longer decay than the sound generated by the second 'set' method which sets `y1[-5]` to 1. The pitch of the generated sound is controlled by the second inlet which sets the delay value used in the two terms of this filter.

5.2 Numerical Solutions to Differential Equations

In this section we present an example of using the *fexpr*$\sim$ object for finding numerical solutions to differential equations. Such numerical solutions are found by defining difference equations which often take the form of a set of interrelated IIR filters.

The Lorenz equation set is one of the most famous tools for studying chaotic behaviors. Lorenz defined the following third order differential equation set as a model for convective flow in the atmosphere.[4]

$$\begin{aligned} \dot{X} &= Pr(Y - X) \\ \dot{Y} &= -XZ + rX - Y \\ \dot{Z} &= XY - bZ \end{aligned} \tag{6}$$

The variables Pr, r, and b are control parameters of the system, and X, Y, and Z are the unknowns for which we find signals as solutions. While there is no random element in the equations, numerical solutions to Lorenz equations can exhibit complex and seemingly random, or in other words chaotic, results. One can generate numerical solutions for the Lorenz equations by the following difference equations:

$$\begin{aligned} X_{n+1} &= X_n + (Pr(Y_n - X_n))\Delta t \\ Y_{n+1} &= Y_n + (-X_n Z_n + rX_n - Y_n)\Delta t \\ Z_{n+1} &= Z_n + (X_n Y_n - bZ_n)\Delta t \end{aligned} \tag{7}$$

The following patch implements difference equations (7). The constant parameters pr, r, b, and dt are defined as variables in the environment with the *value* object.

```
set 0 2.3 -4.4
fexpr~ $y1 + pr * ($y2 - $y1) * dt;
$y2 + (-$y1 * $y3 + r * $y1 - $y2) * dt;
$y3 + ($y1 * $y2 - b * $y3)*dt
```

The 'set' method sets the previous values of the 3 output streams and figure (1) is a graph of the proceeding 2048 output values generated by above patch in Pd.

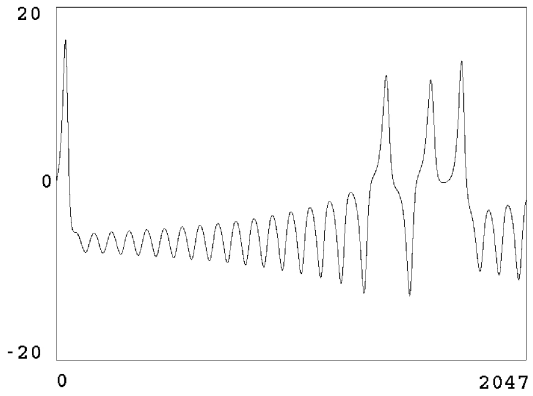


Figure 1: The first 2048 output values of the X signal for Lorenz equations (6) generated with *fexpr*$\sim$ object in Pd with $pr = 10$, $b = 2.66667$, $r = 18$, $dt = 0.01$, and initial values for $X[-1] = 0$, $Y[-1] = 2.3$, and $Z[-1] = -4.4$.

5.3 First Return Maps

Another way of generating chaotic signals is to use one-dimensional nonlinear maps of the form:

$$y_{n+1} = f(y_n) \tag{8}$$

where $f(x)$ is defined by the values of a one-dimensional table. This form of signal generation is

computationally very efficient as there are very little calculations involved for every sample. In *fexpr~* first return map operations can be coded as table lookups. However, we need to note that often such maps are defined between the values of 0 and 1 and since arrays in signal processing environments are kept with integer indexes, we have to scale the y_n value for indexing. If the values of our nonlinear function between the values of 0 and 1 are stored in a float array named "`retmap`" with a size of 2048, the following *fexpr~* expression would account for the scaling of the values:

```
fexpr~ retmap[$y*2048]
```

5.4 Real-time Control at Audio Rate

To keep the filter linear, causal, and time invariant in equation (5), a_i and b_i were defined as constant coefficients; however, with *fexpr~* one is allowed to define the expression as complex as one wishes, assigning various control and signal inputs as coefficients or signal index values. As mentioned before, *fexpr~* performs a 4 point interpolation table lookup anytime a signal is indexed with a fractional value; thus, the following patch defines a filter whose response oscillates smoothly between a flat response and a comb filter with its first zero point at $\pi/64$. The response will be oscillating at the frequency of the *osc~* object connected to the second inlet of the filter.

```
                                        osc~ 1
fexpr~ ($x1+($x2+1)*$x1[-($x2+1)*32])/(2+$x2)
```

In other words, the frequency value of the oscillator is controlling the shape and rate of change of the impulse response of the filter in real-time by changing the coefficient and delay value of the second term of the filter at audio rate. The above filter running within Pd under RedHat Linux 8.0 on a 2.2 GHz Pentium 4 utilizes 3.1% of the CPU (not including the overhead of Pd).

6 Summary

Finding numerical solutions to difference equations, IIR filter implementation, and experimentation with their design and control mechanism are often difficult tasks within the language of most computer music signal processing environments. In this paper we have introduced a new object called *fexpr~* which allows one to define multiple interdependent IIR filters by providing the filter expressions. Of course, such implementation of a filter will not run as efficiently as it would run as an external; however, *fexpr~* can often be safely used if processing power is not a highly urgent issue. It is also a suitable teaching and prototyping tool for experimenting with filter design.

7 Acknowledgment

It is a pleasure to acknowledge the ongoing input and helpful suggestions of Miller Puckette throughout the various design and development stages of the *expr* objects.

References

[1] F. Dechelle, R. Borghesi, E. De Cecco, M. Maggi, B. Rovan, and N. Schnell. jmax: a new java-based editing and control system for real-time musical applications. In *Proceedings, International Computer Music Conference*. San Francisco: ICMA, 1998.

[2] K. Karplus and A. Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55, 1983.

[3] Brian W. Kerninghan and Dennis M. Ritchie. *The C programming language*. Prentice Hall Press, 1988.

[4] Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20, March 1963.

[5] M. Puckette. Pure data: another integrated computer music environment. In *Proceedings, International Computer Music Conference*, pages 269–272. San Francisco: ICMA, 1996.

[6] D. Zicarelli. An extensible real-time signal processing environment for max. In *Proceedings, International Computer Music Conference*. San Francisco: ICMA, 1998.