A Real-Time Compliant Implementation of Linear Time-Invariant Digital Filters in the C Programming Language

Peter Wurmsdobler
Brighton, UK
peter@wurmsdobler.org

Suvendhu Laha
Charlotte, US
suvendhu.laha@eurothermdrives.com

Abstract

This paper describes a fixed-point arithmetic implementation of linear time-invariant digital filters in the C programming language, targeted for the use in real-time applications. FIR and IIR filters are presented in different configurations and representations, such as direct and transposed forms, as well as wave, state-space and normal forms. All these forms provide a first, second and N^{th} order tick or update function, and they may also be arranged in cascades. Scaling of parameters and input values is adressed as well as quantization of filtered output values with optional noise shaping. The implementation is independent of filter types (low pass, high pass, etc.) and filter families (Butterworth, Bessel, etc.). All filters are implemented in a dynamic linkable library, both real-time safe for the GNU/Linux user space and the Linux kernel.

1 Introduction

Generally speaking, a digital filter calculates a digital output value y for a given digital input value x depending on internal states, parameters and an algorithm. Applied on a digital signal $x[k]$, another signal $y[k]$, the filtered digital signal is produced, with k being the normalized time. A digital signal is a special sequence of numbers; it is the time discrete, quantized equivalent of an analogue signal, which is the result of sampling the analogue signal at a constant sampling frequency. The analogue signal itself may be the analogon of a physical quantity. There are several qualifiers for digital filters: time-invariant, time-variant, nonlinear or linear. Only linear time-invariant filters are considered in this paper.

1.1 Filter transfer functions

In the case of linear time-invariant filters (LTI filter), the digital filter processes a digital input signal $x[k]$ according to a specified time-invariant function G producing the digital output $y[k]$. Transformed in the z-domain, this transformation can be expressed as the linear z-transfer function $G(z^{-1})$. Applied to the z-transformed input signal $X(z^{-1})$ the output

$$Y(z^{-1}) = G(z^{-1}) \cdot X(z^{-1}). \qquad (1)$$

The parameters of this transfer function $G(z^{-1})$ are time-invariant and are the result of the filter design process (which is not in the scope of this paper). There are basically two variants of such digital LTI filters: FIR filters and IIR filters. In addition, both can be arranged to form cascades.

1.1.1 FIR filter transfer function

If an impulse is applied as input to a Finite Impulse Response filter (FIR filter), the output remains a finite sequence of non-zero values. The transfer function of such an FIR filter can be written as

$$\begin{aligned} G_{FIR}(z^{-1}) &= \sum_{i=0}^{N} b_i \cdot z^{-i} \qquad (2) \\ &= b_0 \prod_{i=1}^{N} \left(1 - z_{0_i} \cdot z^{-1}\right) \\ &= b_0 \cdot z^{-N} \cdot \prod_{i=1}^{N} \left(z - z_{0_i}\right) \end{aligned}$$

with b_i being the coefficents of the transfer function's numerator polynomial, or the (finite) impulse response with $h[i] = b_i$ for $i = 0$ to $i = n$; or z_{0_i} its zeros and b_0 its gain. N is the order.

1.1.2 IIR filter transfer function

If an impulse is applied as input to a Infinite Impulse Response filter (IIR filter), the output remains in general an infinite sequence of non-zero values, due to its recursive character. The transfer function of such an IIR filter can be written as

$$\begin{aligned} G_{IIR}(z^{-1}) &= \frac{\sum_{i=0}^{N} b_i \cdot z^{-i}}{1 + \sum_{i=1}^{N} a_i \cdot z^{-i}} \\ &= b_0 \cdot \frac{\prod_{i=1}^{N} \left(1 - z_{0_i} \cdot z^{-1}\right)}{\prod_{i=1}^{N} \left(1 - z_{\infty_i} \cdot z^{-i}\right)} \\ &= b_0 \cdot \frac{\prod_{i=1}^{N} \left(z - z_{0_i}\right)}{\prod_{i=1}^{N} \left(z - z_{\infty_i}\right)} \end{aligned} \tag{3}$$

with b_i being the coefficents of the transfer function's numerator polynomial, a_i being the coefficents of the transfer function's denominator polynomial; or z_{0_i} the zeros, z_{∞_i} the poles and b_0 the gain of the transfer function. N is again the order. The number of zeros is here assumed to be the same as the order, which may result in zeros at $z_{0_i} = 0$.

1.1.3 Filter cascade transfer function

The transfer function $G(z^{-1})$ may be split into a product of M stages,

$$G(z^{-1}) = \prod_{i=1}^{M} G_i(z^{-1}) \tag{4}$$

or into a sum of stages,

$$G(z^{-1}) = \sum_{i=1}^{M} G_i(z^{-1}) \tag{5}$$

whereas each stage can be either of first order, second order or of N^{th} order. Alternatively, a network of filters can be made up of a combination of series and parallel cascades. Hence a filter cascade contains filter instances, but also is a filter.

Splitting a digital filter into stages of lower order is done for numerical reasons, especially to guarantee the stability of the filter. For instance, as Eqn. 3 shows, the transfer-function numerator is the product of first order elements $(\prod_{i=1}^{N}(z - z_{0_i}))$. Since most zeros are within the unit circle, the convolution of many such zeros results in very small polynomial coefficients which may become zero in a fixed point implementation.

More importantely, however, poles must be, and must stay within the unit circle for IIR filters. The convolution in Eqn. 3 of many poles $(\prod_{i=1}^{N}(z - z_{\infty_i}))$ for the denominator may result in very small polynomial coefficients which may become zero in a fixed point implementation due to the limitation in precision. If then the poles were re-calculated, they could lie outside the scaled unit circle and cause instability.

1.2 Filter hierarchy

As mentioned earlier, two principal filter types are considered, FIR and IIR filters, as well as cascades of them.

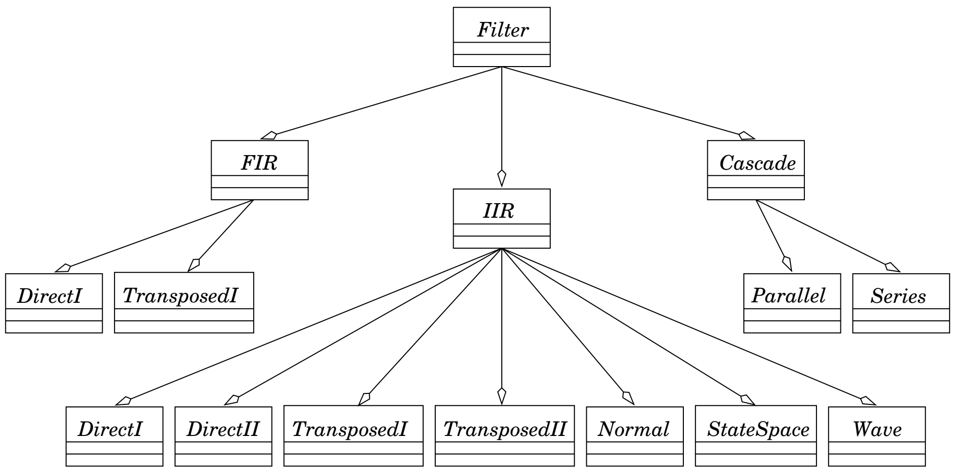

FIGURE 1: *The filter base class and its derivatives*

All can be implemented in different, mathematically equivalent ways, depending on how the transfer functions are calculated in the time domain, on how the parameters are stored, and on how the states are chosen. Even though the filters are implemented in C, a class diagram may de drawn as shown in Fig. 1, with an abstract base class, the Filter.

Before the implementation of all derived classes is presented, however, some general pre-requisites must be treated.

2 Pre-requisistes

When a digital filter is designed, its parameters are generally calculated as `float` or `double` values, i.e. as floating point values. If then the input to the digital filter was a floating point value, a floating point version of a digital filter would generate floating point outputs. However, when signals are sampled for a real-time digital filter, sampling occurs in general in an ADC with limited precision and the values are in general fixed point values, e.g. 12 bit or 16 bit integer values. On the other hand, the output of a digital filter may be fed back into the real world by DACs, using limited precision (fixed point values). Consequently, it makes sense to use a fixed point implementation of the digital filter. However, special means and measures are required for data types, basic operations, scaling of parameters and scaling of samples.

2.1 Data types

The sampling of a signal with a certain precision is reflected by a sample type `sample_t`. Further, a digital filter is characterized by its parameters, hence a parameter type `param_t` is required. A digital filter will carry out multiplications and additions. The result of multiplying a sample by a parameter is stored in an accumulator type `accu_t`, which also accumulates all sums of parameter/sample multiplications. Finally, a general purpose index type `index_t` is introduced.

The following combinations will be possible for the types (32 bit or 16 bit implementation).

```
typedef signed long int      sample_t;
typedef signed long int      param_t;
typedef signed long long int accu_t;
typedef unsigned short int   index_t;
```

An additional type to be used will the the filter type, low pass, high pass, etc. which is accomodated by:

```
typedef enum {
  LOWPASS, HIGHPASS, BANDPASS, BANDSTOP
  } fType_e;
```

2.2 Basic operations

There are a few principal operations carried out with the datatypes introduced, multiplications, summations and unit delays.

2.2.1 Multiplication

A simple gain with a parameter of `param_t` produces an `accu_t` for an input of `sample_t` which is the only type it accepts as input. This can be provided in C in:

```
accu_t multiply (param_t a, sample_t x);
```

2.2.2 Summation

A sum retains the type, `sample_t` or `accu_t`, but can only sum equal types. No special function is implemented in C for summation other than $+$ and $-$.

2.2.3 Unit delays

A unit delay stores a value, either `sample_t` or `accu_t`. No special function is implemented in C other than state variables.

2.3 Scaling

Both parameters and input samples need to be scaled as detailed in the following.

2.3.1 Parameter scaling

Given that all poles for an IIR filter must remain within the unit circle, the denominator coefficients would usually be in the approximate range of $-M...+M$ $((z-1)^N)$, with $\frac{M=(N}{N/2)}$. Casting these floating point coefficients to integers results obviously in a coarse truncation. In order to make use of the word length, i.e. the entire resolution of the fixed point parameters, the floating point paramter values have to be multiplied by a scaling factor S such that all parameter values fit into the fixed point range, e.g. by setting

$$S_{max} = \frac{\max(\texttt{param\_t})}{\max([|p_i|])}. \qquad (6)$$

with $[p_i]$ being the set of flaoting point parameters for the digital filters to be scaled. A maximum input sample multiplied by a maximum parameter would then fit into the maximum value of the accumulator.

It must be considered, however, that the accumulator must also hold the sum of parameter-sample products and an overrun must not occur. For this reason the scaling factor must be smaller. In case of an IIR filter for instance, the worst case would be that all parameters and past values assume their maximum values, which means that the worst case scaling factor

$$S_{worst} = \frac{\max(\texttt{param\_t})}{\sum |p_i|}. \tag{7}$$

Equation 7 is too pessimistic and makes a sub-optimal use of the dynamic range. A better approximation would be a geometric approach,

$$S_{geo} = \frac{\max(\texttt{param\_t})}{\sqrt{\sum p_i^2}}. \tag{8}$$

In this implementation, the scaling factor is restricted to be a power of 2, or

$$S = 2^r \leq S_{geo}, \tag{9}$$

and r has to be found appropriately by the filter design program as

$$r = \texttt{floor}\left(\log_2 S_{geo}\right). \tag{10}$$

2.3.2 Signal scaling

In some cases an input signal must be upscaled prior to further processing by an up-scaling gain, a special form of a gain. Using Eqn. 9, the output of the scaling gain is calculated as

$$Y = S(x) = 2^r \cdot x \tag{11}$$

which can be implemented as a shift-up operation by r bits in C, and provided by:

```
accu_t upscale (sample_t x, index_t r)
```

2.4 Quantizer

The digital filter introduces a gain due to the scaling factor, or scaled parameter. Therefore, the result of multiplications and sums will eventually have to be divided by the same scaling factor and casted to a sample type `sample_t`. This may occur either for the final output, or prior to any other multiplication by another parameter. The quantizer may contain state information and therefore is introduced with its own class, i.e. data type `quantizer_t` as a base class:

```
typedef struct quant_s * quant_p;
typedef struct quant_s {
  /* members */} quant_t;
```

This base class offers the following methods:

```
quant_t * quantizer_create   (index_t r, fType_e t);
void      quantizer_destroy (quant_t *q);
sample_t  quantizer_tick    (quant_t *q, accu_t X);
void      quantizer_reset   (quant_t *q);
```

Later, two derived classes of quantizers will be presented. They will implement similar methods, only the destructor will be the same for all. An enumeration type is introduced as:

```
typedef enum { NS0, NS1 } quant_e;
```

`quantizer_t` contains function pointers to the actual tick and reset function, both being set by the corresponding constructor to:

```
sample_t (*tick) (quant_p q, accu_t X));
void     (*reset)(quant_p q);
```

2.4.1 Simple quantizer

The simplest form of a down-scaling quantizer is of zero order. This quantizer has no states, is algebraic and offers the following methods:

```
quant_t * quantizerNS0_create (index_t r, fType_e t);
void      quantizerNS0_reset  (quant_t *q);
sample_t  quantizerNS0_tick   (quant_t *q, accu_t X);
```

2.4.2 1st order noise shaping quantizer

The division by the scaling factor, or actually the multiplication by 2^{-r} reduces the resolution and hence introduces a quantization error (Note that 2^{-r} can be implemented as a shift down operation, avoiding a division). For this reason a 1st order noise shaping, quantizing and down-scaling gain is used as shown in Fig. 2.

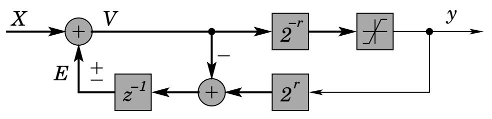

FIGURE 2: 1st *order noise shaping, quantizing gain*

This quantizer has one state `accu_t` and offers the following methods with the filter type (high or low pass) as additional parameter:

```
quant_t * quantizerNS1_create (index_t r, fType_t t);
void      quantizerNS1_reset  (quant_t *q);
sample_t  quantizerNS1_tick   (quant_t *q, accu_t X);
```

Adding up the quantization error will push the quantization noise to higher frequencies and is hence appropriate for low pass filters. Substracting the quantization error will push the quantization noise to lower frequencies and is hence appropriate for high pass filters.

2.5 Filter structure

A structure, a `filter_t` data type accumulates all attributes of all different filter types as shown in Fig. 1 and can be seen as an abstract base class for digital filters:

```
typedef struct filter_s * filter_p;
typedef struct filter_s {
  * members */} filter_t;
```

The base filter class `filter_t` offers the following methods:

```
filter_t * filter_create  (quant_e q);
void       filter_destroy (filter_t *f);
sample_t   filter_tick    (filter_t *f,
                           sample_t x0);
void       filter_reset   (filter_t *f);
```

All filter classes in Fig. 1 will implement a constructor with parameters depending on the filter type. The destructor will be the same for all, simply because the same structure is shared between all filter classes, namely the destructor of the base filter class `filter_t` mentioned above. All filter classes in Fig. 1 will implement at least one function to calculate the output $y0$ for an input $x0$ at each time tick, and a reset function to reset all internal states.

Later it will be seen that a filter class may implement several tick functions depending on the order. `filter_t` contains function pointers to the actual tick and reset function, both being set by the corresponding constructor to:

```
sample_t (*tick) (filter_p, sample_t);
void     (*reset)(filter_p);
```

Finally, `filter_t` contains a function pointer to an array of quantizers used in the filter.

```
 quant_t *Q;
```

3 FIR filters

Equation 2 for an FIR filter can be implemented either as direct I or tranposed I form. With $x[k]$ being the digital input at time k, the output $y[k]$ at time k can be calculated by the difference equation Eqn. 12 as

$$y[k] = \sum_{i=0}^{N} b_i \cdot x[k-i]. \qquad (12)$$

where N is the order (i.e. number of delay elements). Values of $x[k]$ with negative index are zero. This equation is the basis for all FIR implementations.

In the following k is set to zero for a given time instance and all parameters are assumed to be scaled. The FIR filter offers the following methods:

```
filter_t * FIR_create (param_t *b, index_t N,
                       index_t r, fType_t t);
sample_t   FIR_tick   (filter_t *f, sample_t x0);
void       FIR_reset  (filter_t *f);
```

3.1 Direct I form

The direct I form requires storing and updating the past N `sample_t` inputs x_1 to x_N with N being the order. The FIR direct I filter offers the following methods

```
filter_t * FIRdirectI_create (param_t *b, index_t N,
                              index_t r, fType_t t);
sample_t   FIRdirectI_tick   (filter_t *f,
                              sample_t x0);
void       FIRdirectI_reset  (filter_t *f);
```

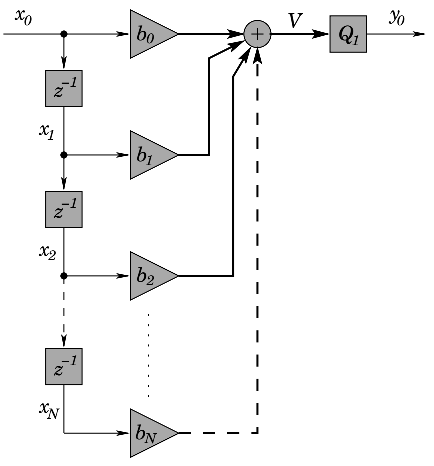

FIGURE 3: *N^{th} order FIR filter stage, direct I form*

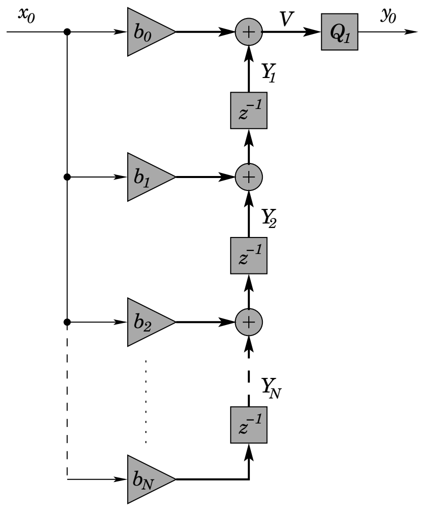

FIGURE 4: *N^{th} order FIR filter stage, transposed I form*

3.2 Transposed I form

The transposed I form requires storing and updating N accu_t states Y_1 to Y_N with N being the order. The FIR transposed I filter offers the following methods:

```
filter_t * FIRtranspI_create (param_t *b, index_t N,
                              index_t r, fType_t t);
sample_t   FIRtranspI_tick   (filter_t *f,
                              sample_t x0);
void       FIRtranspI_reset  (filter_t *f);
```

4 IIR filter

Equation 2 of an IIR filter may be implemented in a direct I, direct II, transposed I or transposed II form. With $x[k]$ being the digital input at time k, the output $y[k]$ at time k can be calculated by the difference equation Eqn. 13

$$y[k] = \sum_{i=0}^{N} b_i \cdot x[k-i] - \sum_{i=1}^{N} a_i \cdot y[k-i] \qquad (13)$$

where N is the order (i.e. number of delay elements). Values of $x[k]$ with negative index are zero. This equation is the basis for most implementations. The other implementations presented here, such as the normal form, state space and wave form, can be derived from them.

In the following k is set to zero for a given time instance and all parameters are assumed to be scaled. The IIR filter offers the following methods:

```
filter_t * IIR_create (param_t *a, param_t *b,
                       index_t N, index_t r,
                       fType_t t);
sample_t   IIR_tick   (filter_t *f, sample_t x0);
void       IIR_reset  (filter_t *f);
```

4.1 Direct I form

The direct I form requires storing and updating the N last sample_t inputs x_1 to x_N and the N last sample_t outputs y_1 to y_N with N being the order. The direct I form filter offers the following methods:

```
filter_t * IIRdirectI_create (param_t *a, param_t *b,
                              index_t N, index_t r,
                              fType_t t);
sample_t   IIRdirectI_tick   (filter_t *f,
                              sample_t x0);
void       IIRdirectI_reset  (filter_t *f);
```

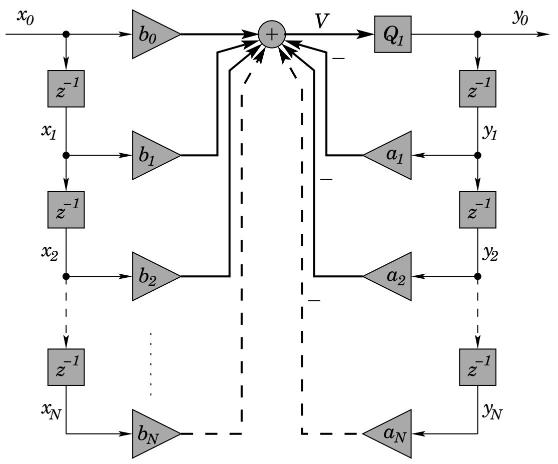

FIGURE 5: *N^{th} order IIR filter stage, direct I form*

4.2 Direct II form

The direct II form requires storing and updating N sample_t states z_1 to z_N with N being the order (first canonical form). The direct II form filter offers the following methods:

```
filter_t * IIRdirectII_create (param_t *a, param_t *b,
                               index_t N, index_t r,
                               fType_t t);
sample_t   IIRdirectII_tick   (filter_t *f,
                               sample_t x0);
void       IIRdirectII_reset  (filter_t *f);
```

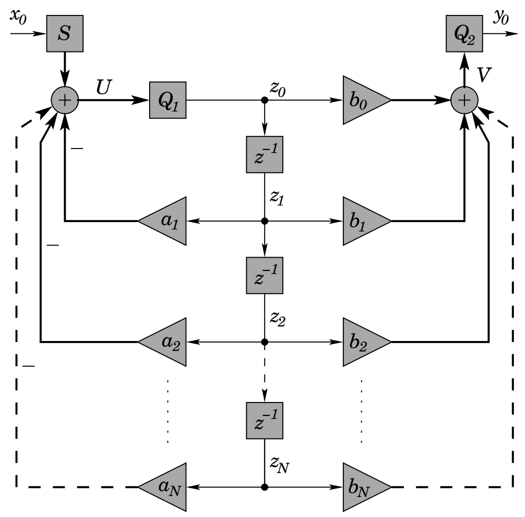

FIGURE 6: *N^{th} order IIR filter stage, direct II form*

4.3 Transposed I form

The transposed I form requires storing and updating N accu_t states X_1 to X_N as well as N accu_t states Y_1 to Y_N, with N being the order. The transposed I form filter offers the following methods:

```
filter_t * IIRtranspI_create (param_t *a, param_t *b,
                              index_t N, index_t r,
                              fType_t t);
sample_t   IIRtranspI_tick   (filter_t *f,
                              sample_t x0);
void       IIRtranspI_reset  (filter_t *f);
```

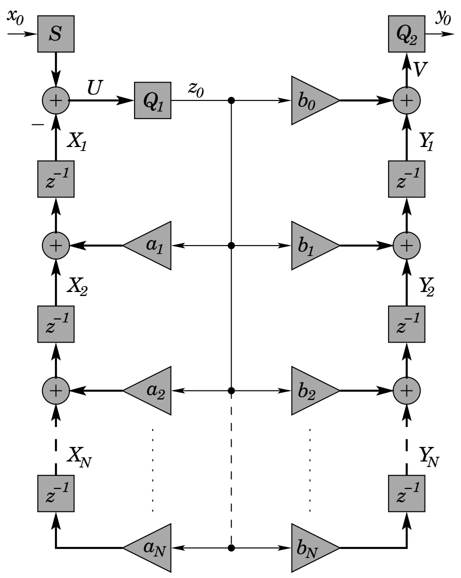

FIGURE 7: *N^{th} order IIR filter stage, transposed I form*

4.4 Transposed II form

The transposed II form requires storing and updating N `accu_t` states Z_1 to Z_N (second canonical form). The transposed II form filter offers the following methods:

```
filter_t * IIRtranspII_create (param_t *a, param_t *b,
                               index_t N, index_t r,
                               fType_t t);
sample_t   IIRtranspII_tick   (filter_t *f,
                               sample_t x0);
void       IIRtranspII_reset  (filter_t *f);
```

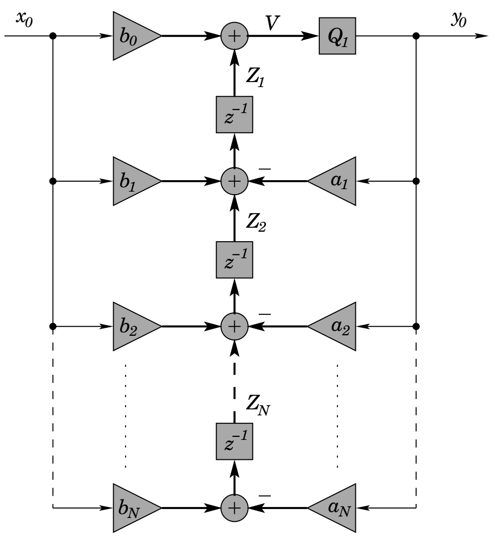

FIGURE 8: *N^{th} order IIR filter stage, transposed II form*

4.5 Normal form

The normal form implements a filter with the poles split into the real and imaginary part. Only poles strictly within the unit circle are considered. The normal form filter offers the following methods:

```
filter_t * IIRnormal_create (param_t *alpha,
                             param_t *beta,
                             index_t N, index_t r,
                             fType_t t);
sample_t   IIRnormal_tick   (filter_t *f,
                             sample_t x0);
void       IIRnormal_reset  (filter_t *f);
```

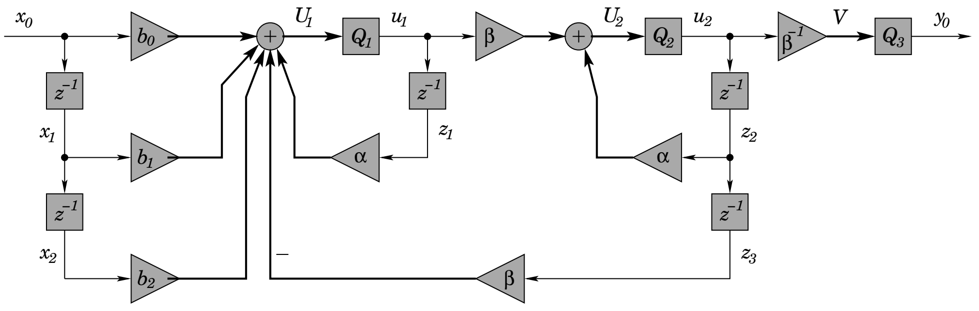

FIGURE 9: *2^{nd} order IIR filter stage, normal form*

A 1st order normal form is equivalent to the direct form I and will be redirected. The 2nd order normal form requires storing and updating past `sample_t` inputs x_1 and x_2, as well as the `sample_t` states z_1, z_2 and z_3. Higher order filters can only be realized by combinations of 1st and 2nd order elements. The relation between the normal form parameters and the second order IIR filter can be derived by comparison of polynomial coefficients.

4.6 State space form

The state space form requires storing and updating N `sample_t` state z_1 to z_N. The state space form filter offers the following methods:

```
filter_t * IIRstatesp_create (param_t *alpha,
                               param_t *beta,
                               param_t *gamma,
                               param_t delta,
                               index_t N, index_t r,
                               fType_t t);
sample_t   IIRstatesp_tick   (filter_t *f,
                               sample_t x0);
void       IIRstatesp_reset  (filter_t *f);
```

Only the second order implementation is detailed here. A higher order implementation is not considered for numerical reasons. The relation between the state space parameters and the second order IIR filter can be derived by comparison of polynomial coefficients achieved after solving the state space equation.

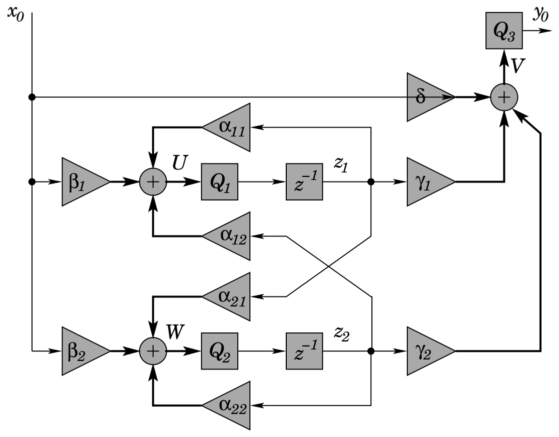

FIGURE 10: 2^{nd} *order IIR filter stage, state space form*

4.7 Wave form

The wave form requires storing and updating N `accu_t` state Z_1 to Z_N. The wave form filter offers the following methods:

```
filter_t * IIRwave_create (param_t *beta,
                           param_t *gamma,
                           index_t N, index_t r,
                           fType_t t);
sample_t   IIRwave_tick   (filter_t *f, sample_t x0);
void       IIRwave_reset  (filter_t *f);
```

Only the second order implementation is detailed here. Higher order filters can only be realized by combinations of 1st and 2nd order elements. The relation between the wave parameters and the second order IIR filter can be carried out by coefficient comparison.

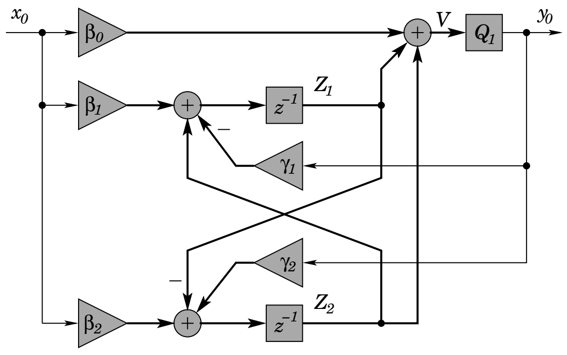

FIGURE 11: 2^{nd} *order IIR filter stage, wave form*

5 Filters cascades

The cascade filter offers the following methods:

```
filter_t * cascade_create  (void);
filter_t * cascade_destroy (void);
filter_t * cascade_add     (filter_t *f);
void       cascade_remove  (filter_t *f);
sample_t   cascade_tick    (filter_t *f, sample_t x0);
void       cascade_reset   (filter_t *f);
```

5.1 Serial cascade of filter stages

The serial cascade filter offers the following method:

```
filter_t * cascadeSerial_create (void);
sample_t   cascadeSerial_tick   (filter_t *f,
                                 sample_t x0);
```

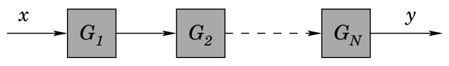

FIGURE 12: *Serial cascade of filter stages*

5.2 Parallel form of filter stages

The parallel cascade filter offers the following method:

```
filter_t * cascadeParallel_create (void);
sample_t   cascadeParallel_tick   (filter_t *f,
                                   sample_t x0);
```

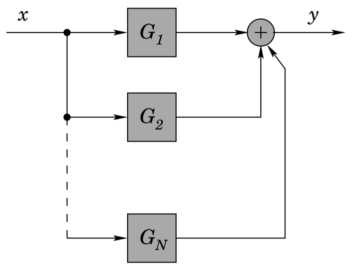

FIGURE 13: *Parallel form of filter stages*

References

[1] Andreas Antoniou: *"Digital Filters: Analysis, Design and Applications"*, March 1993, ISBN: 0071126007.

[2] D. Schlichtharle: *"Digital Filters: Basics and Design"*, August, 2000, ISBN: 3540668411.

[3] Thede: *"Analog and Digital Filter Design Using C"*, November 1995, ISBN: 0133526275.

[4] I. Ain: *"Digital Filters: An Introduction"*, http://www.dsptutor.freeuk.com/dfilt1.htm, http://www.dsptutor.freeuk.com/

[5] Julius O. Smith III: *"Introduction to Digital Filters"*, http://www-ccrma.stanford.edu/~jos/filters/filters.html

[6] Greg Welch and Gary Bishop: *"An Introduction to the Kalman Filter"*, http://www.cs.unc.edu/~welch/kalman/kamlan_filter/kalman.html

[7] The Mathworks: *"Digital Filter Design"*, http://www.mathworks.com/access/helpdesk/help/toolbox/dspblks/digitalfilterdesign.shtml

[8] Rice University: *"FIR and IIR Filter Design Algorithms"*, http://www.dsp.rice.edu/software/rufilter.shtml

[9] Mehmet Zeytinoglu: *"Digital Filter Package (DFP)"*, http://www.ee.ryerson.ca:8080/~mzeytin/dfp/index.html

[10] K. Steiglitz, T. W. Parks and J. F. Kaiser: *"METEOR - FIR Design Package"*, http://www.cs.princeton.edu/ ken/meteor

[11] Brian Wagner and Michael Barr: *"Introduction to Digital Filters"*, http://www.netrino.com/Publications/Glossary/Filters.html

[12] Schlichter, Timothy J.: *"Introduction to Digital Filter Design in C++"*, EE 4000, Introduction to Digital Filtering. Mississippi State University, 1999. http://www.isip.msstate.edu/publications/courses/ece_4000/papers/ifc_filter/

[13] Schlichter, Timothy J.: *"Digital Filter Design Using Matlab"*, EE 4000, Introduction to Digital Filtering. Mississippi State University, 1999. http://www.isip.msstate.edu/publications/courses/ece_4000/papers/digital_filters_matlab/

[14] Gernot Kubin *"Digital Signal Processing Laboratory"*, http://spsc.inw.tugraz.at/courses/dsplab/

[15] Tony Fisher: *"Interactive Digital Filter Design"*, http://www-users.cs.york.ac.uk/~fisher/mkfilter/

[16] Steve Moshier *"Astronomy and numerical software source codes"*, http://www.moshier.net/

[17] *"libDSP - C++ Library of Digital Signal Processing Routines"*, http://sourceforge.net/projects/libdsp/

[18] *"SciPy - Scientific tools for Python"*, http://www.scipy.org/

[19] *"Numerical Python - a fast, compact, multidimensional array language facility to Python"*, http://www.numpy.org/