# REAL-TIME DIGITAL MODELING OF ANALOG CIRCUITRY FOR AUDIO APPLICATIONS

A Major Qualifying Project Report:

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

Bryan Gleason

Date: April 22, 2010

Approved:

Professor William Michalson, Advisor

## Abstract

The goal of this project was to develop a scalable digital signal processing platform capable of modeling analog audio circuits using state-space modeling techniques. Using circuit theory as a foundation, the analog models were built around time-domain solution of circuit analysis. The resultant platform was capable of indistinguishably modeling variable analog filter circuits, with order being only restricted by hardware capabilities. Various continuous-to-discrete time conversion methods were investigated to determine the optimal sounding and performing algorithm.

## Acknowledgements

Firstly, I would like to thank my advisor, Professor William Michalson for his continued support, encouragement, and insight throughout the project. I would also like to thank my friend, family, and loved ones for their support and patience while I babbled on about things which, to the uninitiated, must have sounded like complete gibberish.

# Table of Contents

Abstract ..... i

Acknowledgements ..... ii

Table of Figures ..... v

1. Introduction and Background ..... 1

2. Project Goals ..... 4

3. Proof of Concept ..... 9

4. The Texas Instruments TMS320C6713 DSP Starter Kit (DSK) ..... 11

5. Preliminary Testing ..... 13

    5.1 Phase Issues ..... 21

6. State Space Modeling ..... 24

    6.1 Coding a State Space System Solver ..... 29

7. Discretization Methods ..... 42

    7.1 Frequency Issues ..... 47

    7.2 Algorithmic Issues ..... 50

    7.3 Filter Modification ..... 51

    7.4 Bilinear Transform ..... 52

    7.5 Zero-Order Hold ..... 56

    7.6 Choosing a Methodology and Algorithm ..... 58

8. Audio Testing ..... 60

9. LU Decomposition ..... 71

    9.1 Application of LU Decomposition to the Bilinear Transform ..... 73

10. Providing User Variability ..... 77

11. Putting It All Together ..... 79

12. Final Results ..... 82

13. Conclusions ..... 83

14. Further Considerations and Future Improvements ..... 84

Works Cited ..... 85

Appendix A: Variable Low Pass Filter (Final) C Code ..... 87

Appendix B: RC Filter C Code ..... 96

Appendix C: State Space Sallen-Key Filter Code ..... 98

Appendix D: Bilinear Transform Code ..... 100
Appendix E: Daughterboard Test Code ..... 106
Appendix F: Bilinear Algorithm Verification (MATLAB) ..... 109
Appendix G: Sampling Frequency Analysis (MATLAB) ..... 110
Appendix H: Sallen-Key Band Pass Verification and Analysis (MATLAB) ..... 111
Appendix I: Discretization Method Analysis (MATLAB) ..... 113

# Table of Figures

Figure 1: The Line 6 M13 Stompbox Modeler ..... 4
Figure 2: MXR Distortion+ Schematic ..... 9
Figure 3: Spectrum Digital Texas Instruments TMS320C6713 DSK ..... 11
Figure 4: Basic RC Low Pass Filter: http://www.physics.emich.edu/molab/lock-in/lpcircuit.gif ..... 13
Figure 5: Basic Test Setup Block Diagram ..... 15
Figure 6: Initial RC Filter Response ..... 15
Figure 7: Initial RC Filter Response (modified values) ..... 16
Figure 8: Capacitor Model with ESR, ESL
(http://www.ecircuitcenter.com/Circuits/cmodel1/cmodel1.htm) ..... 17
Figure 9: Capacitor Frequency vs. Impedance ..... 17
Figure 10: Improved RC Filter, 5nH ESL ..... 19
Figure 11: Improved RC Filter, 5nH ESL, to 22 kHz ..... 20
Figure 12: Phase Response Test Setup Block Diagram ..... 21
Figure 13: Phase Response of Analog and Digital RC Filters (Logarithmic scale) ..... 22
Figure 14: Phase Response of Analog and Digital RC Filters (Linear scale) ..... 23
Figure 15: Sallen-Key Low Pass Filter ..... 25
Figure 16: Sallen-Key Low Pass Filter with Ideal Op Amp Assumptions ..... 26
Figure 17: Sallen-Key Low Pass Filter with Node Names ..... 26
Figure 18: Continuous Model Bode Plot ..... 32
Figure 19: Discrete Model Bode Plot ..... 32
Figure 20: Multisim Simulation Circuit: Sallen-Key Low Pass ..... 33
Figure 21: Simulated Frequency Response ..... 34
Figure 22: Filter PSD Test Results ..... 35
Figure 23: Effect of Function Generator Output Impedence ..... 37
Figure 24: Single Pole Low Pass Filter With Output Buffer ..... 38
Figure 25: Effect of Filter Output Buffer ..... 39
Figure 26: Revised RC Filter Response ..... 41
Figure 27: Results of Various Methods of Discretization of a Continuous RC Filter Model ..... 43
Figure 28: Continuous and Discretized First Order Low Pass Filter Magnitude Response ..... 44
Figure 29: Continuous and Discretized Sallen-Key Low Pass Filter Magnitude Response ..... 45
Figure 30: Continuous and Discretized Sallen-Key Band Pass Filter Magnitude Response ..... 46
Figure 31: Sallen-Key Low Pass Filter Discretized at Various Sampling Frequencies ..... 48
Figure 32: Sallen-Key Bandpass: Analog vs. DSP ..... 60
Figure 33: Audio Signal Test Setup ..... 61
Figure 34: Magnitude Response of Band Pass Filter with Audio Input ..... 62
Figure 35: Sharp Cutoff Low Pass Filter Magnitude Response ..... 64
Figure 36: Magnitude Response of Band Pass Filter Audio after Low Pass Filtering ..... 65
Figure 37: Audio Test: Bilinear vs Zero-Order Hold vs Audio Magnitude Response ..... 66
Figure 38: Sallen-Key Band Pass Filter Sampled at 44.1kHz, 96kHz ..... 68
Figure 39: Block Diagram: Prewarping in Parallel Filter System ..... 69

Figure 40: Block Diagram: Prewarping in Series Filter System ..... 70
Figure 41: General Guitar Amplifier "Tone Stack" ..... 70
Figure 42: DSK6XXXHPI Daughterboard ..... 77
Figure 43: Schematic of Analog Control Input ..... 78
Figure 44: Code Flow Diagrams: Main Fuction, Infinite 'while' loop, ISR ..... 81
Figure 45: Spectrogram of Variable Sallen-Key Low Pass Model ..... 82

# 1. Introduction and Background

Most of the sought-after tones of the musical world are produced by analog equipment. This is likely largely due to historical precedence, given that all electronic musical equipment began as analog. Thus, many of the sounds that musicians wish to emulate were produced by vacuum tube amplifiers, bucket brigade analog delay chips, germanium transistors, et cetera.

Criticisms of early digital equipment called it harsh and sterile. Many of these complaints persist today. Though tremendous advances have certainly been made in the realm of digital audio effects, there is still some stigma attached. This may be especially true when applied to the digital modeling of analog devices. The issue lies in the fact that the digital realm is highly conducive to precision, and error occurs in the form of round-off and overflow. In the analog realm, error occurs in the form of non-ideality. It is, in fact, this non-ideality that has become the desirable trait of analog musical equipment.

There are certain advantages and disadvantages to the digital processing of audio signals. In general, digital systems have some distinct advantages over their analog counterparts. Once an analog signal is digitized, it is theoretically infinitely reproducible. That is, there is no signal degradation associated with reproduction as the signal has been transformed into data that can be easily, cheaply, and transparently copied.

In recording studios, recordings were traditionally made with analog equipment. With the advent of digital recording techniques and digital audio workstations (DAWs), analog audio processing equipment has often been replaced by digital “plug-ins”. Plug-ins are software modules that interact with an audio editor to provide audio effects or synthesis. In digital studios, instead of using an outboard compressor, for example, the audio signal can be soft-routed to a compressor plug-in to perform the same duties.

Digital effects have the benefit to consumers of taking up no physical space, often being less expensive, and often being more flexible than their analog counterparts. They have the benefit to manufacturers of being less expensive to produce, as the effects are not necessarily hardware dependent, and in the case of plug-ins and software effects, distribution can be essentially free.

Digital modeling effects have come a long way, especially in the studio realm, but digital modeling effects for guitarists are often found to be lacking. In the words of one online reviewer of the Boss FBM-1 Fender '59 Bassman effects pedal, "Very (easy to use), but no amount of knob twiddling matches a real tweed Bassman!" One finds this sentiment often in reviews of modeling equipment.

Digital signal processing is an incredibly powerful and flexible toolset. It is capable of things that analog circuits simply cannot be. In many cases, it is unknown how exactly modeling is performed as it is usually a proprietary technology. This leaves the independent investigator to his or her own devices. With the assumption that circuit theory is able to accurately predict the behavior of an analog circuit subjected to audio signals, hypotheses may be formed as to how to reconstruct analog audio processor behavior on a digital system. If real time performance is not a requirement, processing can be incredibly time consuming and complex, and furthermore can operate in the frequency domain if necessary. If real time is a requirement, such as in this application, a time domain solution must be sought so that transformation into the frequency domain can be avoided. Therefore, a methodology to extract the relevant information determined from circuit analysis must be developed. Another approach would be the individual analysis of elements of a circuit as two-port filters and the matching of filter performance by curve fitting. Stringing these together, one might reconstruct a circuit bit by bit as a series of wave shaping algorithms and digital filters. The benefit of a methodology informed by circuit analysis is that it is scalable. Copious analysis must still be performed, but the method will be consistent for any analog circuit. In fact, there will be an honest attempt to make all elements of the resultant system scalable

such that the upper limits of the system will be determined by processing power, memory, or both. As DSP technology continues to rapidly evolve and improve it can only be assumed that processors will become faster and space requirements less stringent. If the system developed by this investigation proves useful, circuits of incredible complexity maybe be solvable in real time on the proper platform without modification to the algorithm.

This project centers around the investigation of a novel method of modeling analog circuits in real time on a digital signal processor (henceforth referred to as a DSP). To begin, it was necessary to start simply so as to first prove the principle before attempting application to a complex system.

## 2. Project Goals

The ultimate goal of this project is develop a novel digital amplifier and effects modeling system for guitarists or other musicians. Though digital "multi-effects" processors have existed for some time, the goal of this project is more specific. There are several key elements to the concept.

First, let us define some of the key concepts and requirements. A digital effects modeler seeks to emulate the sounds of classic or desirable effects pedals or amplifiers. The Boss FBM-1 Fender '59 Bassman pedal mentioned earlier is an example of this. Other modelers are more complex, such as the Line 6 M13 Stompbox Modeler, shown in Figure 1, which is comprised of 19 delay effects, 23 modulation effects, 17 distortion effects, 12 compressors and equalizer effects, 26 filter effects, and 12 reverb effects (Line 6). Each of the effects is modeled after an actual (often vintage) analog effects pedal or amplifier.


Figure 1: The Line 6 M13 Stompbox Modeler

The M13 is a better example than the FBM-1 of the potential of digital modeling devices. It not only takes up far less space than a collection of analog pedals, but it costs far less per effect as well. Routing is simplified as well as all signals are internally soft-routed. These devices are self-contained

units. Some offer the ability to modify and store settings and "patches" using a computer and provided software package.

The primary aim is to develop a digital audio system which can model analog audio signal processing devices. Furthermore, given long-standing complaints against digital modelers, we seek to develop a *superior* system. To do this, a novel modeling paradigm must be developed and investigated.

As time passes, electrical engineers rely more and more on circuit simulation to accurately predict the behavior of their designs before they are built. Tremendous effort has been invested in this field, resulting in continuously better models and simulation data. It is also possible to obtain time-domain solutions from simulated circuits, though not in real time. Since circuit simulation can accurately model complex behavior in circuits, it follows that the same techniques ought to be able to be used to simulate analog effects pedals and amplifiers. The biggest issue with this is that circuit simulators do not operate in real time, as there is no need to. If circuit simulation techniques are to be utilized to model audio effect circuitry in real time, a method must be developed that *can* compute solutions in real time.

A circuit used to process signals such as audio can be examined in two domains: time and frequency. Analysis in the frequency domain is often most useful as when dealing with audio, frequency-dependent behavior is usually the focal point. Thus, a transfer equation for a given circuit can be developed, and from this, frequency-domain behavior can be analyzed and predicted. Take for example a simple RC low pass filter. The time domain equation governing this circuit is:

$$v_{out}(t) = \frac{1}{RC}\int_{t_0}^{t}(v_{in}(\tau) - v_{out}(\tau))d\tau + v_{out}(t_0)$$

Whereas the transfer equation is:

$$\frac{v_{out}}{v_{in}} \stackrel{\text{def}}{=} H(s) = \frac{1}{1 + RCs}$$

Where s =σ+jω.

As can readily be observed, if we are operating in the frequency domain, computation becomes much simpler, and the effect of circuit more apparent. Unfortunately, we do not exist in the frequency domain, so for a processed signal to be useful, it must be transformed to the time domain.

Let's look at how frequency domain processing is done. Since frequency is defined very simply as occurrences in a period of time, this means that to analyze in the frequency domain from a time domain signal, we need to acquire more than a snapshot, that is, more than a single instance of the signal to be able to examine its frequency content. In practical application, this means *batch processing*. Batch processing is where the signal is recorded for a period of time, then this set of samples is transformed to the frequency domain, processing is performed, the signal is transformed back to the time domain, and then finally output. One issue with this type of signal processing is that there is necessarily a delay between the input(s) and output(s), as a 'batch' of sample must be collected before the transformation can be performed. Much delay is unacceptable, as any noticeable delay from input to output will destroy the playability of the system.

Despite the benefits of frequency-domain processing, it is more desirable for this application to develop a system which can operate entirely in the time domain. This means that a discrete time signal can be processed on a sample-by-sample basis, and the only latency between the input and the output will occur as a result of processing done to each sample (at most one sampling period). If this delay is kept small, it will be imperceptible.

Operation in the time domain means the solution of systems of differential equations. Since we seek a time domain solution, we need to find a way to efficiently solve systems of linear equations. The

phrase “systems of linear equations” should bring to mind Linear Algebra, the field of mathematics that deals with just that. One technique, most commonly found in control systems engineering, but born of linear algebra, is called *state space modeling*.

State space modeling defines a system by a combination of state variables such that “every possible signal in the system can be expressed as a linear combination of these state variables.” (Lathi, 2005). These state variables are taken to be the energy storage devices (capacitors and inductors) in the system. Thus, given state space model which represents a system of interest, the output for any arbitrary can be modeled. This is the core concept of this project. Another benefit to the state space approach is that it is readily scalable and thus can handle large and complex systems. Furthermore, it can easily handle systems with multiple inputs and multiple outputs (MIMO). Though in this project, we will only attempt single-input single-output (SISO) systems, it is nice to know that we can expand to MIMO systems without major adjustment.

Now that the core concepts have been defined and the methods explained, the over-arching goal can be explained. We know from SPICE and other circuit simulation packages that circuits can be analyzed algorithmically based upon node definitions. Therefore, it should be possible to algorithmically determine state space models from circuit descriptions and input/output definitions. What we would seek to do is to develop a system wherein a user could enter a circuit, as in a schematic capture program, and from this circuit, export a state space model (or series of models) to a digital signal processor which could then process arbitrary signals as though it were the circuit which the user had entered into the program. Essentially, the system would be a real time circuit simulator.

Unfortunately, this goal is overly ambitious. Given the limited timeframe, the focus must be narrowed to a manageable aspect of the project. It was decided that the project would focus on the real time solution of state space models of circuits.

Starting with simple analog filter circuits, each circuit will be hand-analyzed and a continuous-time state space model will be developed. Then, the continuous-time state space model will be converted to a discrete-time model. On a standalone digital signal processor (DSP) platform, code will be written to compute an output based upon an arbitrary input and the state space model. Once this has been completed and is functioning, the next step is to develop a method of making the filter time-varying. That is to say, we want for there to be user-modifiable parameters that can be adjusted while the system is running. The end result will be a digital model of a time-varying analog filter circuit. Success of the system will be described in terms of basic functionality, and ultimately in terms of its ability to replicate and be indistinguishable from the analog circuit it is emulating.

## 3. Proof of Concept

To begin, an analog audio processing device was simulated in National Instruments Multisim. In this case, the schematic of an MXR Distortion+, a common guitar 'stompbox' effect was entered into Multisim. This way, a very common effect could be simulated and qualitatively compared to expectations for the resultant sound. Included in the Multisim package are two Labview instruments which allow a simulated circuit to process arbitrary signals as captured by a microphone attached to the computer and outputted from the computer's speakers. The circuit can be seen below in Figure 2.

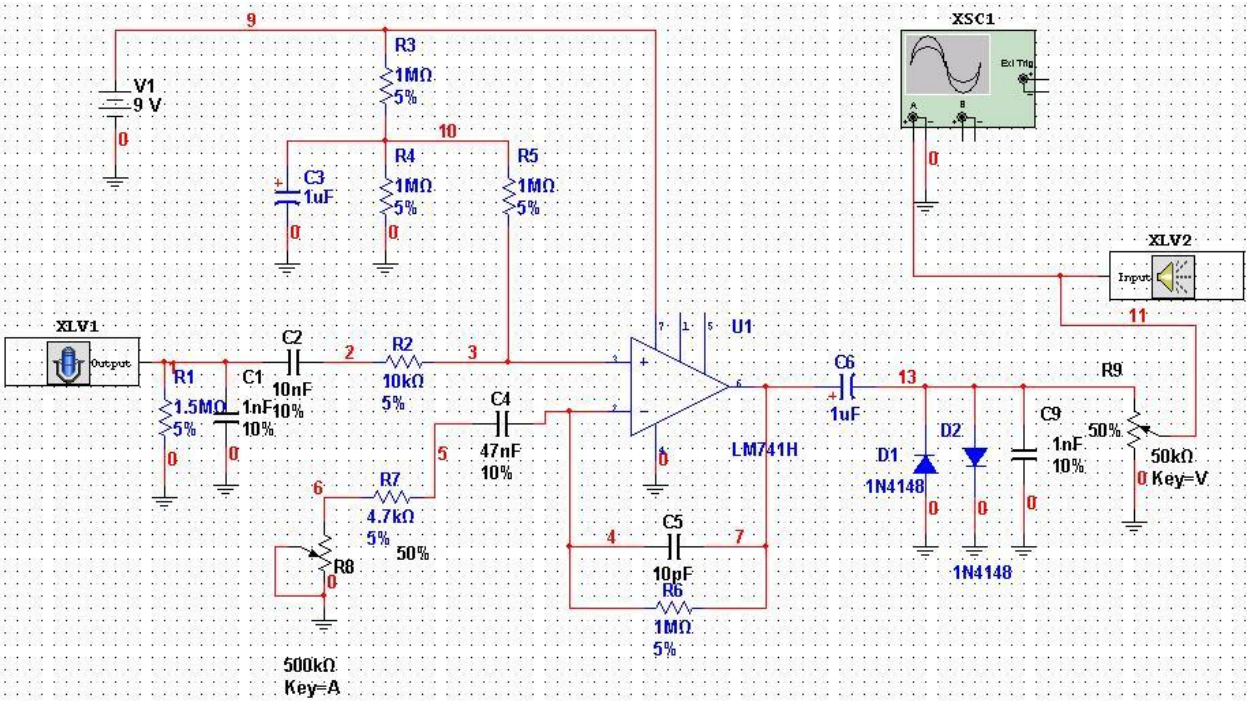



Figure 2: MXR Distortion+ Schematic

This circuit contains several filter elements, an operational amplifier in the noninverting gain configuration, set up for variable gain (via R8), a symmetric diode clipping stage, and a variable output (via R9). It is a very common guitar distortion effect. The input and output devices, XLV1 and XLV2 allow for an external arbitrary signal to be processed by the circuit. XSC1 is a software oscilloscope which is set up to monitor the signal at the output.

Though initial results sounded promising, this initial test setup proved too inflexible to be useful, as it was very difficult to provide a consistent input and to extract data for analysis. Nevertheless, it was established that SPICE modeling did indeed seem useful for the modeling of circuits as systems for processing arbitrary audio signals.

# 4. The Texas Instruments TMS320C6713 DSP Starter Kit (DSK)

Figure 3: Spectrum Digital Texas Instruments TMS320C6713 DSK

Since the aim of this project to is produce a standalone digital signal processing device and not merely a software package, a platform had to be chosen which could operate independently of a computer. High quality analog to digital and digital to analog converters would be necessary for the sampling and reconstruction of the audio signals.

The Spectrum Digital DSK for the TI TMS320C6713 (seen in Figure 3) was chosen for a variety of reasons. The primary reason it was chose is that there was significant familiarity with the platform and the with the development environment, Code Composer Studio (CCS), from a class on digital signal processing. This would allow for reduced development time to an already established understanding of much of the basic functionality of the DSK and of CCS.

From a hardware standpoint, the TMS320C6713 meets all required specifications and provides most of the necessary functionality. It is a 225MHz floating point processor, capable of handling word lengths of up to 32bits and sampling at up to 96kHz (Spectrum Digital Inc., 2010). It is necessary that the processor be capable of floating point processing as precision is paramount and the round-off error of a

fixed point processor would be unacceptable. Similarly, handling 32bit words means higher sample resolution. Since this platform needs to processor high quality audio information, high resolution is desirable. Though the signals will most likely be sampled at 44.1kHz, having flexibility in the sampling rate is not a bad thing, as it may be determined down the road that oversampling would be beneficial. Lastly, the high clock rate of 225MHz allows for high speed processing of the data which, given the potential complexity of the application, is highly desirable.

Sampling is provided by the TLV320AIC23B stereo audio codec, which provides a microphone input, a stereo line input, stereo line output, and an amplified headphone output. There are also 4 user-definable dip switches and 4 user-definable LEDs. In addition to the audio interface, there are three headers for interfacing with memory, peripherals, and external hosts. In a later section it will be described how the host port interface (HPI) will be used to overcome one of the DSK's shortcomings.

Some functionality that the DSK does not natively provide is any DC coupled analog inputs or fast serial communication. This makes it difficult to communicate with the DSK while it is running. This can however be worked around.

## 5. Preliminary Testing

For the next step, it was decided that an informative and simple circuit to attempt to model would be a simple passive single pole low pass filter, as can be seen in Figure 4.

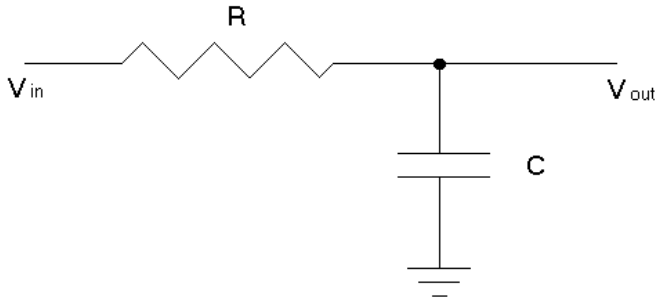



Figure 4: Basic RC Low Pass Filter: http://www.physics.emich.edu/molab/lock-in/lpcircuit.gif

In this circuit, the output voltage is the voltage across the capacitor. To obtain the voltage across the capacitor, we look to the current-voltage relationship of the capacitor:

$$v_C(t) = \frac{1}{C}\int_{t_0}^{t} i_C(\tau)d\tau + v_C(t_0)$$

If we take Kirchhoff's current law (KCL) at the output node $V_{out}$, we recognize that all current flowing into the resistance R flows out through the capacitance C. Thus, $i_R = i_C$. The current through the resistor can be found with the following equation:

$$i_R(t) = \frac{v_{in}(t) - v_{out}(t)}{R}$$

Since $i_R = i_C$, we can substitute the above relationship into the current-voltage relationship and rearrange to get the following:

$$v_C(t) = \frac{1}{RC}\int_{t_0}^{t} (v_{in}(\tau) - v_{out}(\tau))d\tau + v_C(t_0)$$

Examining the above equation, we find that it can easily be discretized. Taking dτ to equal T, the sampling period of a discrete time system, we know that:

$$\int_{t_0}^{t} (v_{in}(\tau) - v_{out}(\tau))d\tau = \big(V_{in}(\tau) - V_{out}(\tau)\big) - (V_{in}(t_0) - V_{out}(t_0))$$

Where V is the antiderivative of v. Taking $t_0$ to equal 0 and t to equal T,

$$\int_{t_0}^{t} (v_{in}(\tau) - v_{out}(\tau))d\tau \sim T(v_{in}(T) - v_{out}(T))$$

Entering this approximation back into the equation for $v_c$,

$$v'_c(t) = \frac{T}{RC}\big(v_{in}(t) - v_{out}(t)\big) + v_c(t_0)$$

This essentially says that the *next* value of $v_c$ is equal to the difference between $v_{in}$ and $v_{out}$ over a period of time equal to T (the incremental change) plus the last value of $v_c$. This can be easily made into a difference equation of the form:

$$v_{out}[n] = v_{out}[n-1] + k(v_{in}[n-1] - v_{out}[n-1])$$

Where,

$$k = \frac{T}{RC}$$

This difference equation is easily coded and implemented in the interrupt service routine (ISR). To test this code, a passive single pole analog low pass filter was constructed to match our schematic. Testing was performed by inputting Gaussian noise from a Tektronix AFG3021 Arbitrary Function Generator at a rated amplitude of $10V_{pk\text{-}pk}$ to both the analog filter and the DSK and then recording the

outputs via the computer audio line in port directly into Matlab for analysis. The test setup is described in Figure 5,

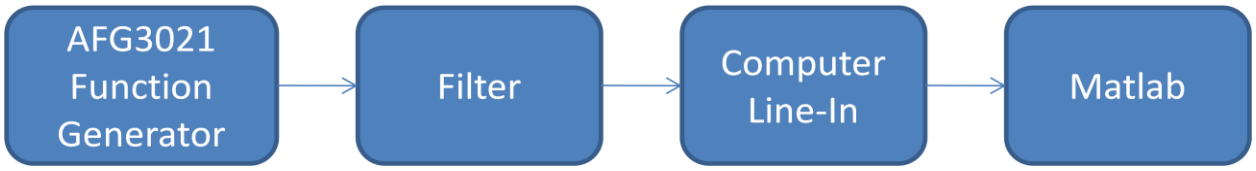



Figure 5: Basic Test Setup Block Diagram

Figure 6 below shows the magnitude of the input noise, DSK output, and the analog output, normalized to 0 dB.

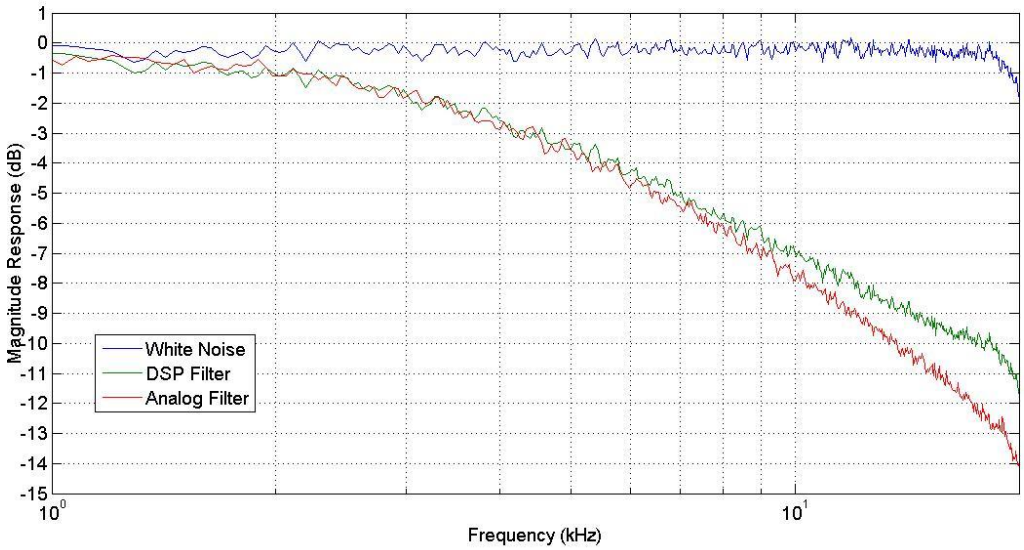



Figure 6: Initial RC Filter Response

As can be seen above in Figure 6, the two filter track reasonably well up to approximately 6 kHz, at which point the slope of the attenuation of the analog filter can be seen to decrease relative to the DSP filter. To try to obtain a more accurate model, component values were adjusted and the effects noted. Figure 7 shows the magnitude response with slightly modified resistance and capacitance values.

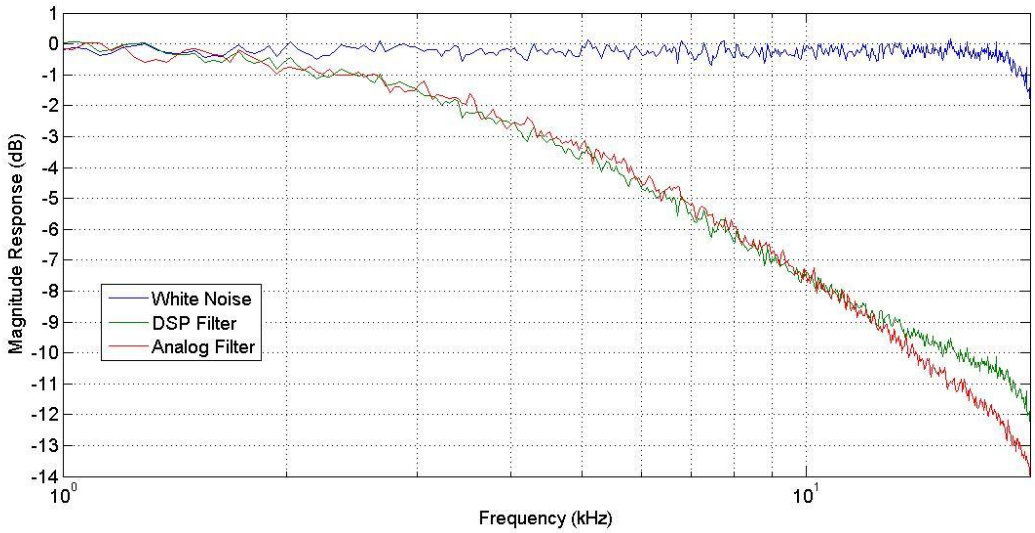



Figure 7: Initial RC Filter Response (modified values)

Though this is certainly closer to the desired response, the two responses definitely have different shapes, and we still see divergence between the two within 11 kHz. When these results are compared to the ideal filter response, we can see that the DSP filter tracks more closely than the analog one. Under normal circumstances, this is a positive result, but since we are trying to accurately model analog circuitry, we are also trying to accurately model *error*. From the results of these initial tests, it was hypothesized that there is error that the most basic R and C models do not account for. Since they are relevant for the band in question (about 20Hz to 20kHz), we must determine a method for accurately modeling these errors, at least within this band.

To begin this investigation into error, we search for more advanced models of basic components, namely capacitors and resistors. For example, a more advance model of a capacitor is shown in Figure 8.

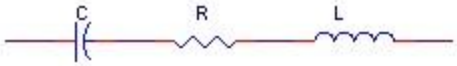



Figure 8: Capacitor Model with ESR, ESL (http://www.ecircuitcenter.com/Circuits/cmodel1/cmodel1.htm)

For this particular circuit, the equivalent series resistance (ESR) may be lumped with the resistor value. From examination of manufacturers' data, it can be concluded that values of equivalent series inductance (ESL) tend to be low (<5nH). Nevertheless, it is apparent from the experimental results that more than pure capacitance is at work here. Figure 9 shows a graph from Kemet shows the frequency dependence of the impedance of their capacitors. We see that the impedance is frequency dependent, decreasing by a factor of $10^2$ over two decades of frequency. ESL is taken to be a constant value.

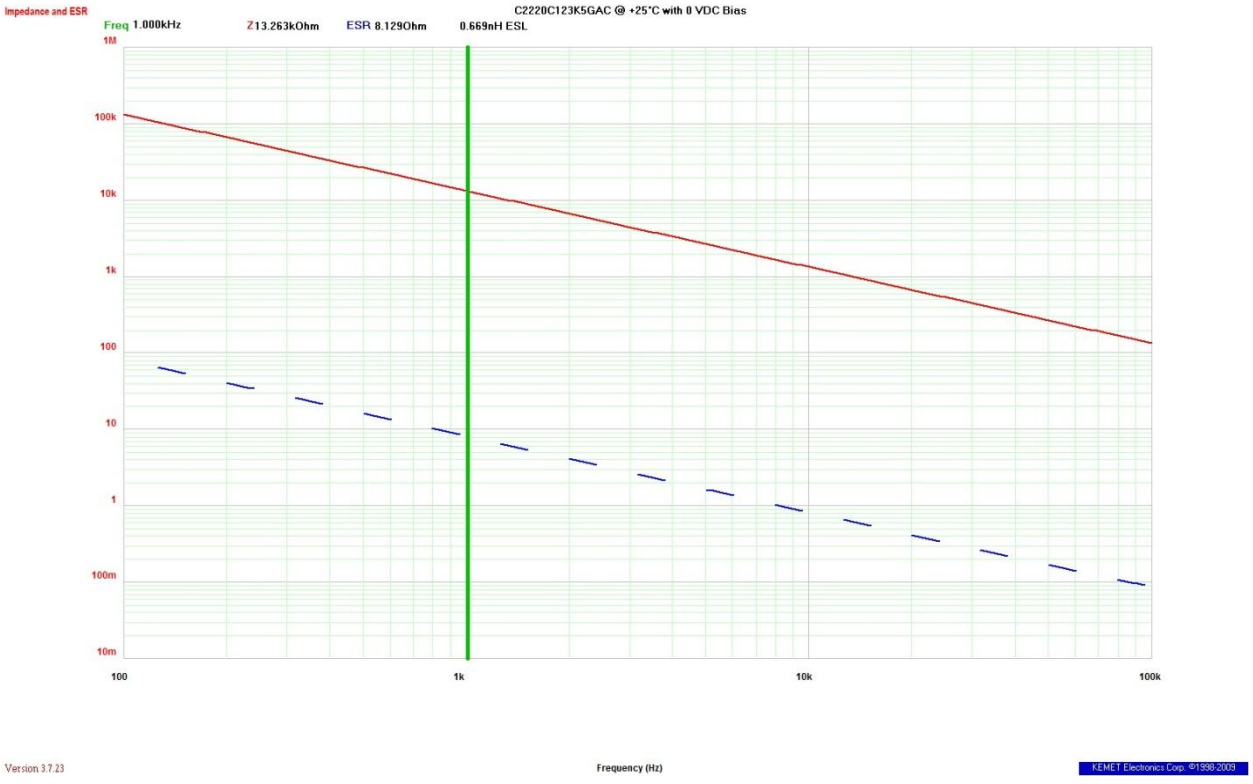
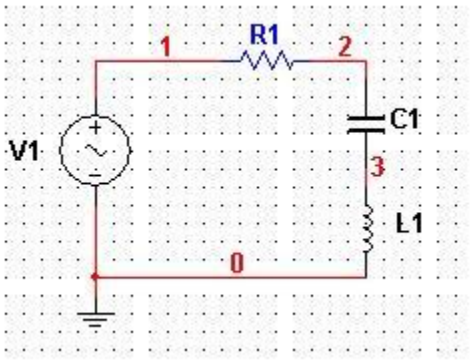



Figure 9: Capacitor Frequency vs. Impedance

To attempt to take the unanticipated aspects of the capacitor into account in the model, a small ESL was added. The resultant equivalent circuit is as follows:




L1 is the ESL, C1 is the original capacitance, and $V_{out}$ is taken at node 2.

Since the voltage across an inductor is given by:

$$v_L(t) = L\frac{di_L(t)}{dt}$$

Thus, using a similar approximation method as in the RC circuit, this can be rewritten as:

$$v_L(t) = L(i_L(t) - i_L(t_0))$$

We can readily add this into our RC filter difference equation to give us:

$$v_{out}[n] = v_{out}[n-1] + k(i[n]) + L(i[n] - i[n-1])$$

where,

$$i[n] = v_{in}[n-1] - v_{out}[n-1]$$

One notable difference between this set equations and the single difference equation of the basic RC circuit is that this implementation requires the storage of past values of the input and output, but also the past current value. Since this entail only the storage of three floating point values, it is of little

consequence, but in a complex system, the advanced capacitor model will mean a marked increase in the complexity of the code.

Implementing this as C code on the DSK, the filter results show marked improvement. Figure 10 shows the improved filter results from 100Hz to 12 kHz.

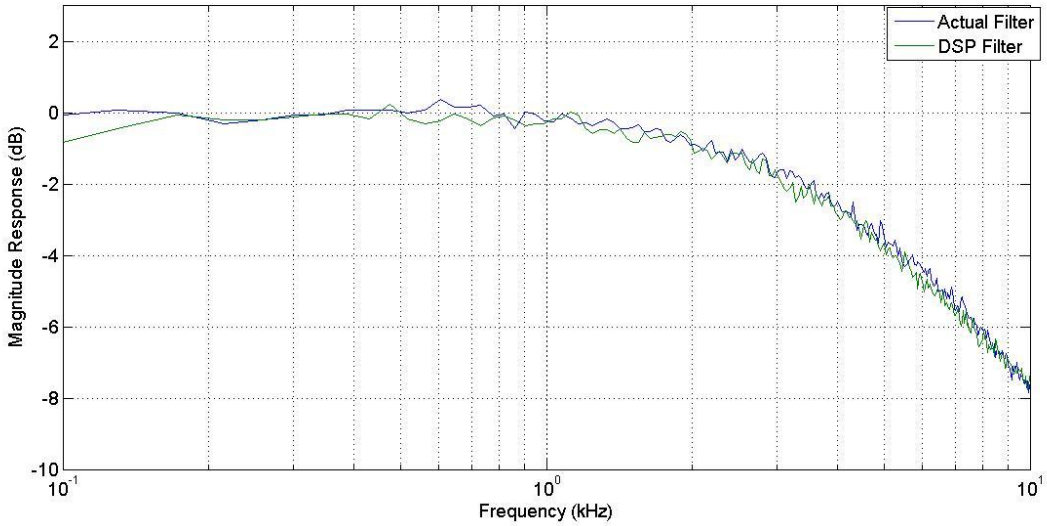



Figure 10: Improved RC Filter, 5nH ESL

Within this specified range, the DSP filter tracks the analog filter quite well. The response is close enough that it can be assumed (until proven otherwise) that any differences would be unnoticeable. There is however some divergence at higher frequency, as can be seen in Figure 11, which is the same data as Figure 10, but plotted up to 22 kHz.

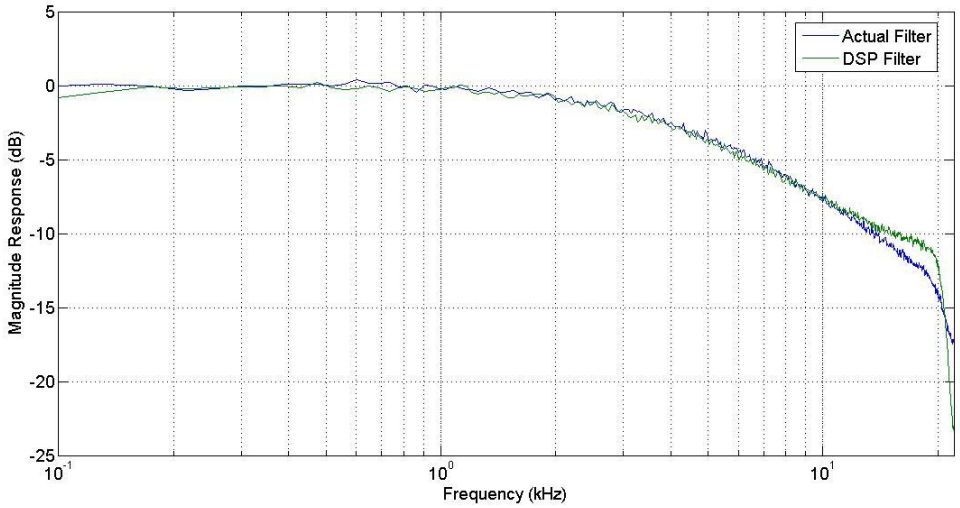



Figure 11: Improved RC Filter, 5nH ESL, to 22 kHz

This shows a maximum difference of about 2.5 dB, occurring near 20 kHz. This likely is due to the inductance in the circuit, whose effect will become more prevalent as frequency increases. It is possible that what is being observed in the analog filter has less to do with impedance and more to do with the presence of parallel capacitance, in the capacitor itself (as some highly advanced models show) or from the breadboard on which the test circuit was built. This certainly merits investigation or simply reconstruction of the circuit with minimized trace length to reduce any lead inductance, capacitance, or noise pick up.

## 5.1 Phase Issues

In addition to potentially differing from their analog counterparts in magnitude response, digital filters also have the potential to differ significantly in their phase response. In passive filter such as have been examined here, capacitors and inductors are responsible for phase lead and lag, respectively. Therefore, the phase response of an analog circuit will be determined by its components and topology, and will also be related to its magnitude response.

There is debate as to whether or not phase distortion is readily perceived by humans. The interesting situation here is that, as previously mentioned, one of the goals of this study is to accurately model error in analog systems. It can be said that nonlinear phase response is phase distortion, and therefore can be categorized as an error produced in analog processing. Many modern digital systems attempt to produce linear phase response, or *constant group delay*. This means that there is a constant delay for all frequencies passing through the system.

Phase distortion is easily noticeable in stereo sound field applications or in applications involving feedback and this distortion is considered undesirable. However in mono applications, "this distortion is of no importance." (Slump, et al.) Proceeding with this assumption, the phase response ought to nevertheless be characterized such that if aurally perceptible differences between the analog and digital filters are found, there may already be a basis for explanation.

To explore the phase response of the filters, the modified test setup of Figure 12 was used:


Figure 12: Phase Response Test Setup Block Diagram

Here, the AFG3021 was set up to output a sine wave, whose frequency was varied. The oscilloscope was used to observe the input and output waveforms. For each frequency, the magnitude of the input and output were recorded, as well as the time delay between the two. By performing this test at 100, 200, 300, 400, 500, 600, 700, 800, 900, 1k, 2k, 3k, 4k, 5k, 6k, 7k, 8k, 9k, 10k, and 20kHz there were enough points to create a phase plot. To convert time delay to phase, the following equation is used:

$$\theta = -360° \left(\frac{t_d}{T}\right)$$

Where *T* is the period of a given frequency and $t_d$ is the measured time delay between the input and output. The results of this are seen in Figure 13 below. The analog filter exhibits the sort of phase response that is expected, shifting approximately -180° over the frequency range. The phase behavior of the digital filter is not as apparent on logarithmic axes. Figure 13 shows the same plot on linear-linear axes.

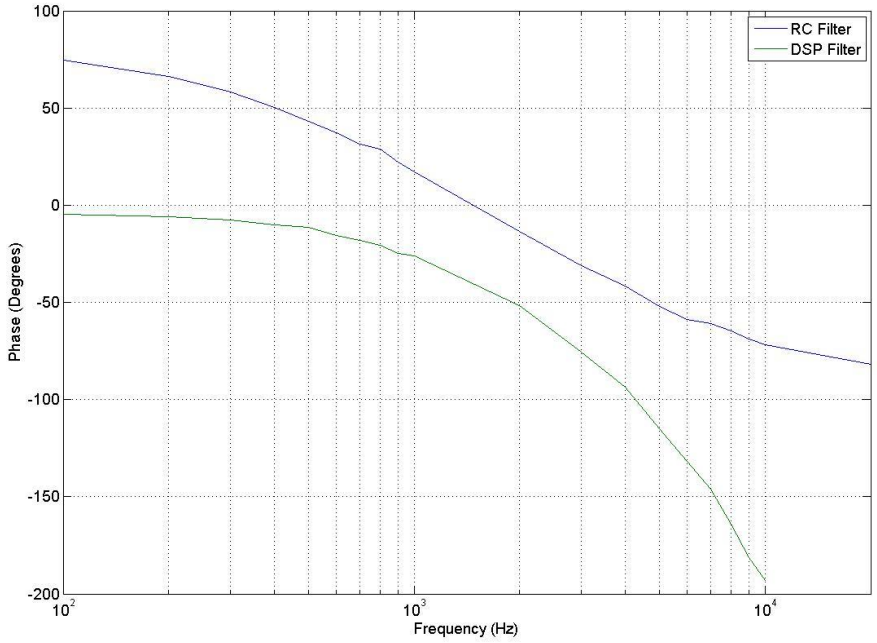



Figure 13: Phase Response of Analog and Digital RC Filters (Logarithmic scale)

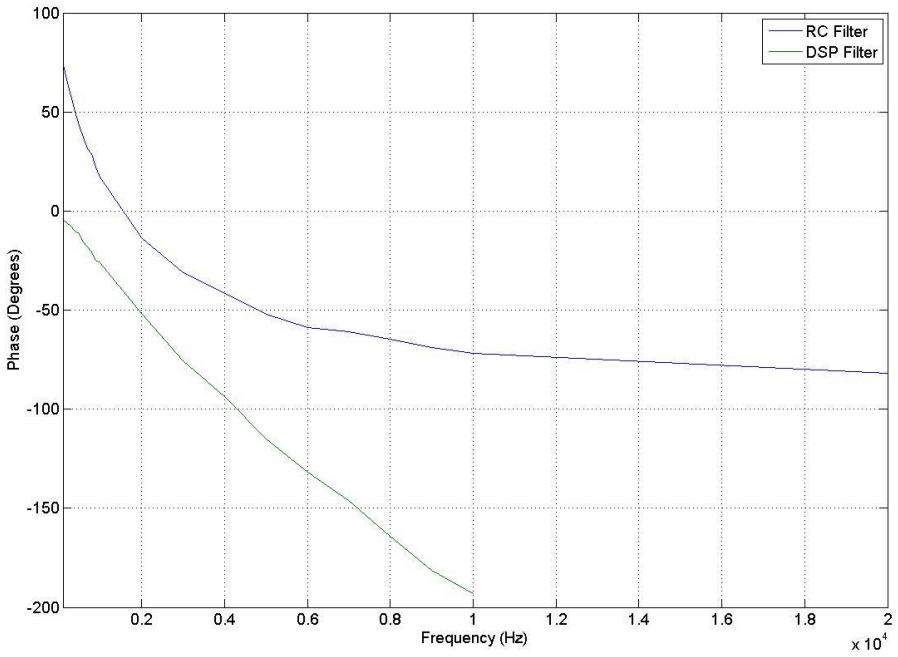



**Figure 14: Phase Response of Analog and Digital RC Filters (Linear scale)**

In Figure 14, the phase responses of the analog RC filter and the digitized RC filter are shown. Though the previous plot and most of the plots in this paper are presented on a semi-log (logarithmic x-axis) scale, the results of this phase analysis are most apparent on linear axes. Here it can be seen the phase response of the digital filter is linearly related to frequency. In other words, it exhibits constant group delay. This due to the fact that time delay between each input and output is constant. This time delay is constant because all outputs are calculated in an identical fashion, regardless of frequency. As can be inferred from the above equation, if $t_d$ is constant, phase is inversely proportional to frequency. If the computational techniques this paper proposes are utilized, the result with respect to phase will be linearity. Later qualitative testing will be the only way to determine if phase distortion introduced by the analog circuitry is an important characteristic of that type of sound. In the meantime, operation under the assumption that it is not is necessary.

# 6. State Space Modeling

State space modeling is a technique used to describe a system in terms of its inputs, outputs, and various states, all represented by first order differential equations, in the time domain. The states represent energy storage devices within the system. As applied to circuits, this means capacitors and inductors, primarily. State space representation is ideal for our purposes for a number of reasons. Our primary goal is *real time* circuit modeling. State space representation provides a time-domain solution and is conducive to numerical solution. Since state and output information is encoded in a series of matrices, solution of our outputs and next states boils down to a series of multiply-accumulates. This is precisely the type of computation a DSP chip is designed to do well. State space equations take the following form:

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

Where A is known as the state matrix, which governs how the current value of a state will affect the next value of that state. B is the input matrix, which determines how the input to the system affects the value of the next state. C is the output matrix, which determines how the current state affects the output. D is the feedforward matrix, which determines how the input affects the output (the D matrix is often null).

The first circuit to be tested using state space representation was chosen to be a simple active Sallen-Key filter. Utilizing design guidelines provided by Texas Instruments (Texas Instruments, 2002), the following circuit, seen in Figure 15, was constructed on a breadboard. The op amp selected is a Texas Instruments TL072 dual op-amp, which is a high performance op amp suitable for audio applications and can be found in numerous guitar effects pedals.

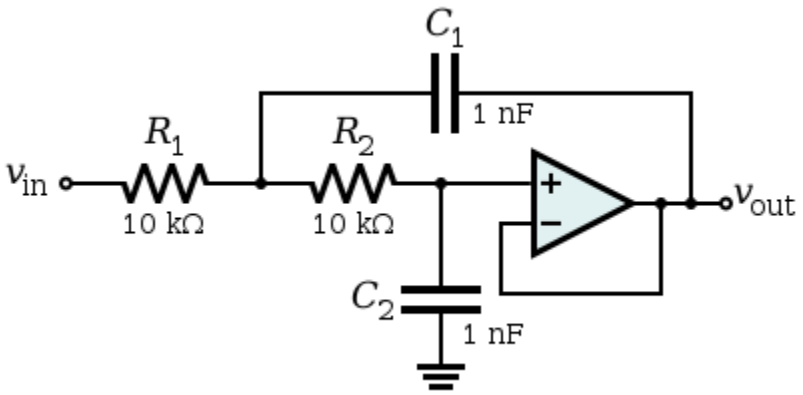



Figure 15: Sallen-Key Low Pass Filter

To create a state space representation of this circuit, circuit analysis must first be performed. For simplified initial testing, the following ideal operational amplifier assumptions were made:

1) Input current $i_{in}$= 0A
2) Input offset voltage $V_{OS}$= 0V
3) Input Impedance $Z_{IN} = \infty$
4) Output Impedance $Z_{OUT} = 0\Omega$
5) Gain a = ∞

In making the above assumptions, analysis of the circuit becomes much simpler. Figure 16 below shows the equivalent circuit with ideal op-amp assumptions. It is acceptable to make such assumptions until results of the resultant analysis indicate that the model is over-simple. If it is found that the results of assumptions are incongruent with the behavior of the analog circuit, more advanced models may be implemented. However, it that can be avoided, it will be.

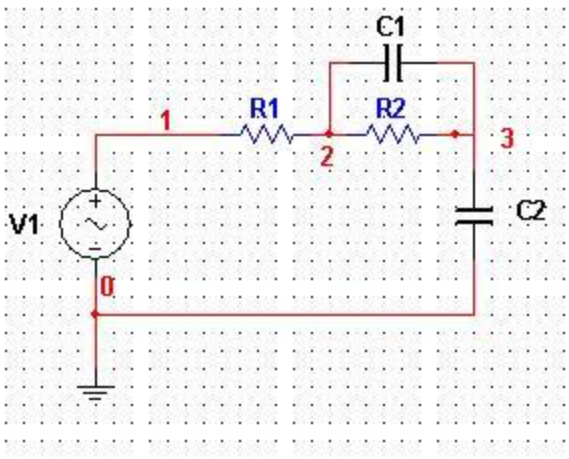



Figure 16: Sallen-Key Low Pass Filter with Ideal Op Amp Assumptions

Figure 15 shows that the op amp in the circuit in question is set up as a non-inverting unity gain buffer. Since the offset voltage is assumed to be zero, we can conclude that $V_+ = V_-$.

With these assumptions, we can begin analysis. First, we name reference nodes. These can be seen below in Figure 17.

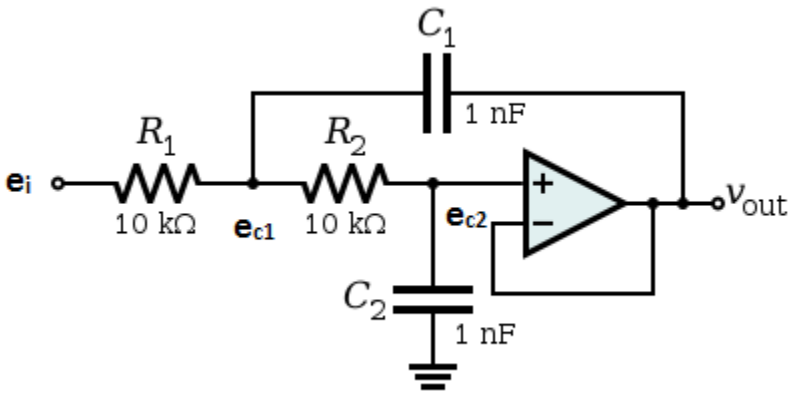



Figure 17: Sallen-Key Low Pass Filter with Node Names

First, we perform Kirchhoff's Current Law at Node $\mathbf{e_{c1}}$:

$$\frac{e_i - e_{C1}}{R_1} + \frac{e_{C2} - e_{C1}}{R_2} + i_{C1} = 0$$

Rearranging, we find that

$$i_{C1} = \frac{e_{C1} - e_i}{R_1} + \frac{e_{C1} - e_{C2}}{R_2}$$

Since

$$i_C = e_C^{'} * C$$

$$e_{C1}^{'} = \frac{e_{C1} - e_i}{R_1 C_1} + \frac{e_{C1} - e_{C2}}{R_2 C_1}$$

Where $e'_C$ is the new or next value of $e_C$. Rearranging,

$$e_{C1}^{'} = \left(\frac{1}{R_1 C_1} + \frac{1}{R_2 C_1}\right) e_{C1} + \left(-\frac{1}{R_2 C_1}\right) e_{C2} + \left(-\frac{1}{R_1 C_1}\right) e_i$$

Similarly,

$$e_{C2}^{'} = \frac{e_{C2} - e_{C1}}{R_2 C_2}$$

and

$$e_{C2}^{'} = \left(-\frac{1}{R_2 C_2}\right) e_{C1} + \left(\frac{1}{R_2 C_2}\right) e_{C2}$$

Lastly, since $V_{out} = V_-$, $V_- = V_+$, and $V_+ = e_{C2}$,

$$e_O = e_{C2}$$

To finish the state space representation, we must put these equations in matrix form:

$$\begin{bmatrix} e_{C1}^{'} \\ e_{C2}^{'} \end{bmatrix} = \begin{bmatrix} \left(\frac{1}{R_1 C_1} + \frac{1}{R_2 C_1}\right) & \left(-\frac{1}{R_2 C_1}\right) \\ \left(-\frac{1}{R_2 C_2}\right) & \left(\frac{1}{R_2 C_2}\right) \end{bmatrix} \begin{bmatrix} e_{C1} \\ e_{C2} \end{bmatrix} + \begin{bmatrix} \left(-\frac{1}{R_1 C_1}\right) \\ 0 \end{bmatrix} e_i$$

$$e_O = [0 \quad 1] \begin{bmatrix} e_{C1} \\ e_{C2} \end{bmatrix} + [0] e_i$$

## 6.1 Coding a State Space System Solver

Since all of the data is in matrix form and time-invariant, we can pre-compute all values within the matrices. This is done within the main section of the code, before it enters the infinite 'while' loop. This way, computation of these values will not affect the performance of the filter. Since the input has no way of directly affecting the output, the D matrix is null. Here it should be noted that if the system were not time-invariant, that is to say if values in one or more of the matrices were allow to change over time, such as in the case of user-variable controls, this simple pre-computation would be insufficient. In this case, a range of values may be computed and stored in a lookup table. Methods for addressing time-varying filters will be discussed in a later section. Thus, based on input from a variable control, pre-computed values from the lookup table could be placed into one or more of the matrices with minimal affect on system performance. There is a speed-size tradeoff in this decision, but barring numerous interactive controls in a large and complex system, data size should not be an issue. This may need to be addressed in future implementations of this system, but for the time being, this method should be more than sufficient.

There are two parts to the algorithm: one to compute the output, and another to compute the next state values. First, the "next states" computed on the previous iteration must be moved into the current state buffers. Since the output for a given iteration relies on the current state values, we may then compute it using the following code:

```
for(n=0;n<N;n++){                    //loop to compute output
        output+=out[n] * x[n];
    }
```

Once the output has been computed, it may be written to the DAC and outputted. Next, the next states, for use in the following ISR may be computed. Since all matrix values can be stored in no

more than two-dimensional arrays, a nested 'for' loop algorithm can be written to perform all of the "next state" calculations for a system of any size. In the following code, N is the number of states in the system.

```
for(k=0;k<N;k++){                          // choose which matrix row
      x_next[k] = B[k] * u;                // do B matrix multiplication
      for(n=0;n<N;n++){                    // choose which matrix column
            x_next[k] += A[k][n]*x[n]; //do A matrix multiplication
            }
      }
```

Using the methods above, a state space representation of a system of N states can be calculated. At this time, the upper bound of N is unknown.

The implementation of the state space model of the Sallen-Key filter using the solver algorithm proved difficult. Entering the values into the system and running it, it was found that the system was unstable. The states and output were found to increase exponentially. Much debugging resulted in no results, so an identical state space model was created in MATLAB so as to better examine the workings of the system.

MATLAB allows for the creation of two types of state space models: continuous and discrete. Initially, a discrete state space model was created. In performing a linear time-domain analysis (lsim), behavior identical to the DSP implementation was observed. This was an informative result in that it showed that the code was correct, but that there was an error within the representation. Developing a continuous model in place of a discrete one, it was found that the filter model behaved correctly, albeit with a slightly different frequency response (a lower -3dB point).

Much effort was put into determining the reason for the discrepancy between the continuous and discrete models, but no conclusion arose. MATLAB also allows for the discretization of continuous models using the c2d() function. This function samples a continuous model at a specified frequency and develops a discrete model from the result (The MathWorks, 2010). The assumption was made that the output of this conversion would be the same as the original discrete-time state space model, but it was found that the matrix coefficients were very different. Running this model, it was found that the time domain response and the frequency and phase responses were nearly identical to the continuous state space model responses, with the exception of some aliasing error in the discrete time model. Figure 18 shows the bode plot of the continuous model and Figure 19 shows the bode plot of the discrete time model. As can be observed, the responses are identical, excepting the aliasing error. Taking the coefficients from the converted model and entering them as the coefficient values in the DSP filter, it was found to have the response predicted by the MATLAB models.

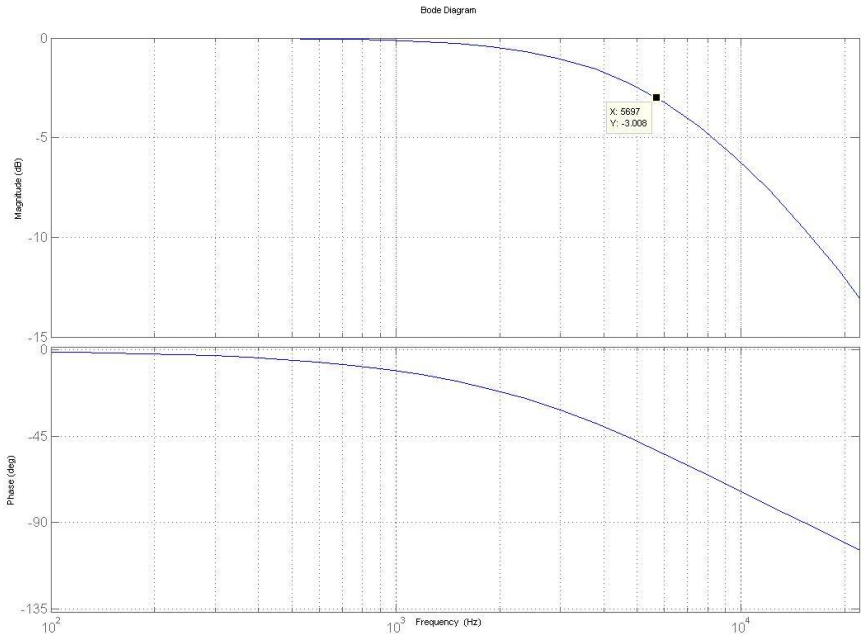



**Figure 18: Continuous Model Bode Plot**

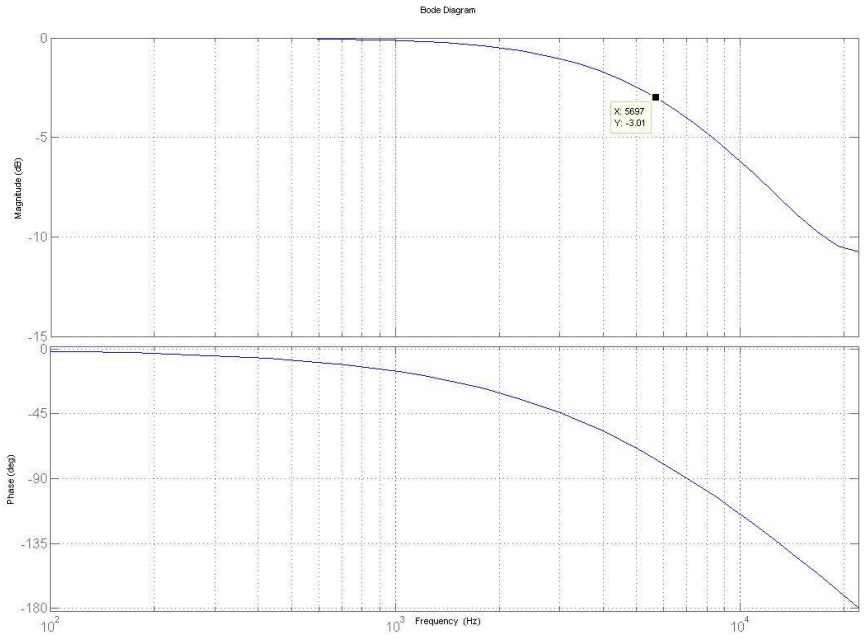



**Figure 19: Discrete Model Bode Plot**

To determine the expectation for the analog implementation of the circuit, the circuit was simulated in National Instruments Multisim and it frequency response captured. The simulated circuit was built to be close to the actual circuit. Since a Texas Instruments TL072 operational amplifier is being utilized in the filter, its closest available relation, the TL071 model (the single op amp package version of the TL072) was chosen for the op amp in the simulation circuit. The simulation circuit may be observed in Figure 20. Refer to Section "State Space Modeling" for in depth analysis.

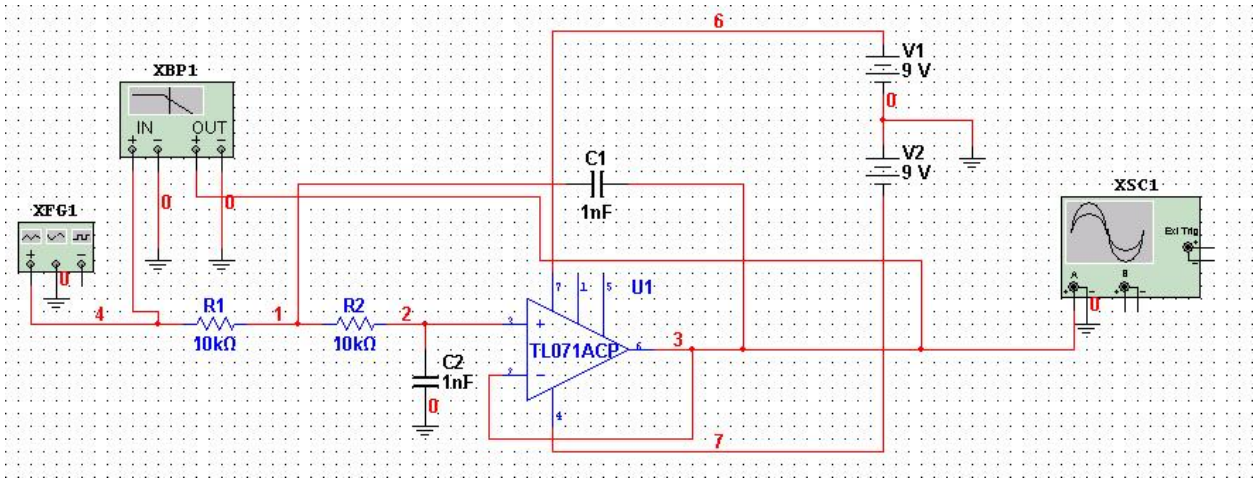



Figure 20: Multisim Simulation Circuit: Sallen-Key Low Pass

This circuit produced the frequency response which can be observed in Figure 21. We see that the -3dB point for this filter is at approximately 10 kHz. As was observed in Figures 18 and 19, the -3dB points for the modeled filters were at approximately 5.7 kHz. This is a notable discrepancy which will be investigated forthwith.

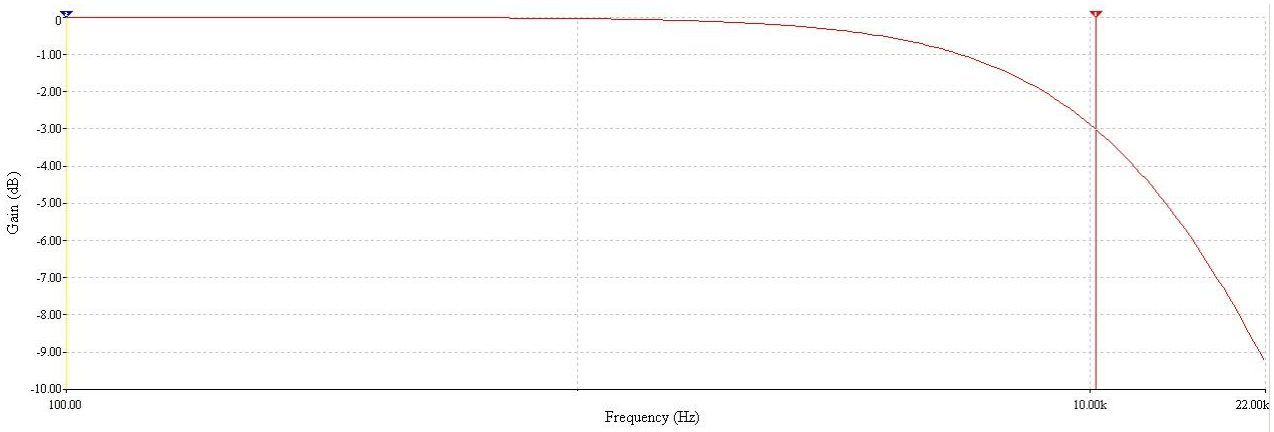



Figure 21: Simulated Frequency Response

Since the nature of the discrepancy between the state space model responses and the simulation response is as of yet unknown, it was decided to proceed with testing to confirm or deny the results of the various simulations. Using the physical test platform described in Section (RC circuit), Gaussian noise was inputted to both filters, and the outputs recorded and analyzed. Figure 22 shows the magnitude response of both filters. It appears that both filters more or less match their predicted results.

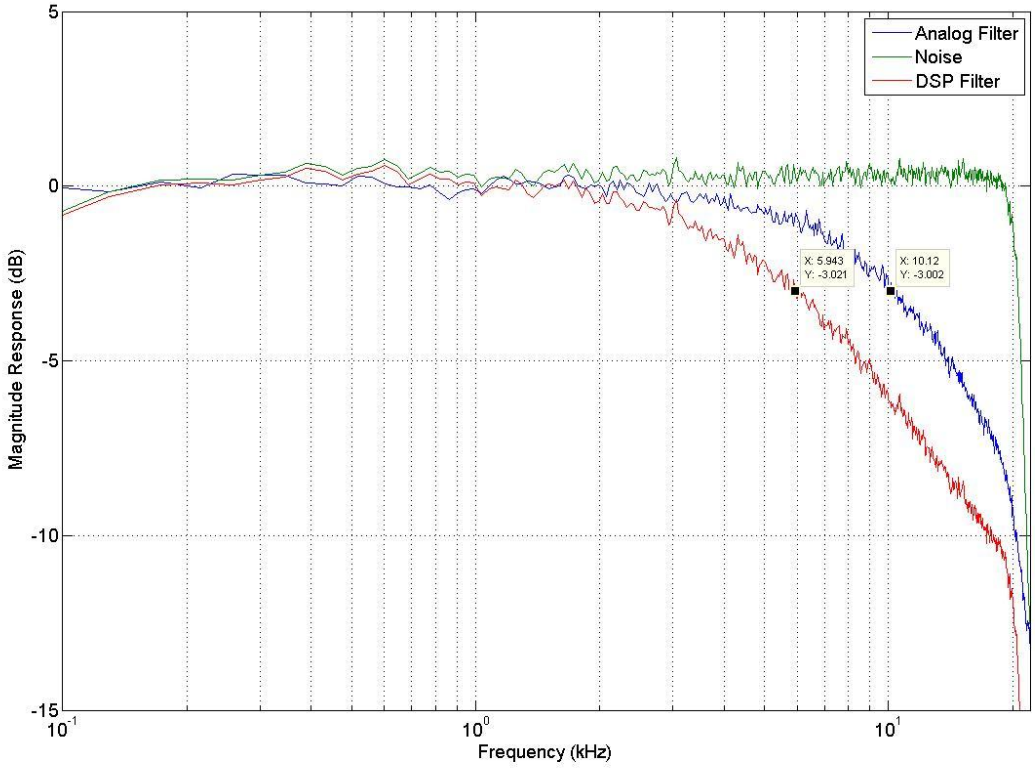



Figure 22: Filter PSD Test Results

This indicates that the simulation was a reasonably accurate model of the analog filter and that the MATLAB state space model was a reasonably accurate model of the DSP filter. This is a positive result in that it confirms the modeling techniques. Obviously, there is still a serious difference between our analog and DSP filters and between our state space model and simulated circuit. Furthermore, the curvature of the analog filter response is reminiscent of the curvature of the analog RC filter, again differing from the DSP filter. As such, there are three problems to solve:

1) It must be determined how the continuous time model is converted to a discrete one so that there is not a reliance on MATLAB to generate the coefficients, and to understand the shortcomings of the model.

2) It must be determined why there is a difference between the frequency responses of the analog and state space modeled filters. Essentially, why does the state space filter model have a lower -3dB point?
3) Lastly, it can be assumed that even if the roll-off frequency issue is corrected, there will still be differences in the frequency responses of the two filters due to differences in curvature. Though this issue was worked around using small inductances in the RC filter, this is not the total solution. This issue merits further investigation. One theory is that changing the test setup from a breadboard to a soldered board with minimized lead length may lead to a more predictable response from the analog filter.

To determine the answers to the above questions, it was necessary to return to the basic assumptions. Firstly, the difference in frequency responses was examined. It had been assumed that the disparity between the digital and analog filters was due to the representation or calculation method. This proved to be true, but not in the way expected. Upon comparing the analog filter to its ideal transfer equation response in MATLAB, it was observed that the digital filter response was closer to that of the ideal, and it was the analog filter that was performing unpredictably. The circuit analysis and transfer function were reanalyzed and it was discovered that there were fundamental errors in the analysis. The analysis was based on information found in a reputable source, and thus went unquestioned. This result was in some ways positive however, because it confirmed that the digital models were accurate and were performing as expected. Additionally, the issue of output loading was addressed.

In testing the analog RC filter, the signal generator was connected directly to the filter input, and the output was connected directly to the 'line in' input of the computer so that it could be recorded directly

into MATLAB for analysis. Issues of impedance matching and output loading had not been considered. It was hypothesized that the output impedance settings on the function generator might affect the output, and that buffering the output might also affect the output without affecting its desired response. As it was determined that it was the *analog* filter which was behaving unpredictably and the circuit under test was such a basic one, these theories seemed sound.

First, as it was easiest to test and therefore rule out, the effect of output impedance settings of the function generator on the circuit was tested. Using an identical output load circuit, the function generator output mode was set at 50Ω and High Z. The 50Ω mode assumes a lower input impedance for the following stage, where as High Z mode assumes a high impedance input to the following stage. Figure 23 below show the result of changing this.

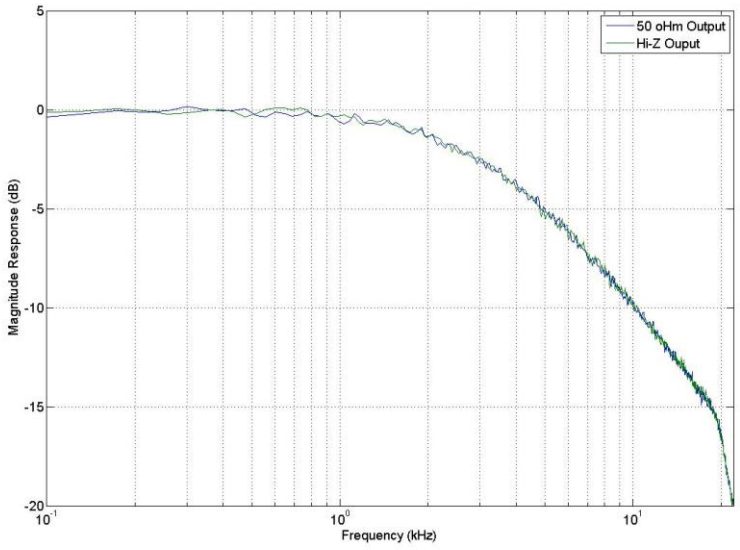



**Figure 23: Effect of Function Generator Output Impedence**

As can be seen in the figure above, altering the output impedance of the function generator does not affect the response of the filter. Next the output loading of the filter circuit will be tested. It is hypothesized that the 'line in' input to computer is loading down the filter and causing it to behave

differently. To test this, a unity gain op amp buffer will be added between the filter and the computer input. The circuit can be seen below in Figure 24:

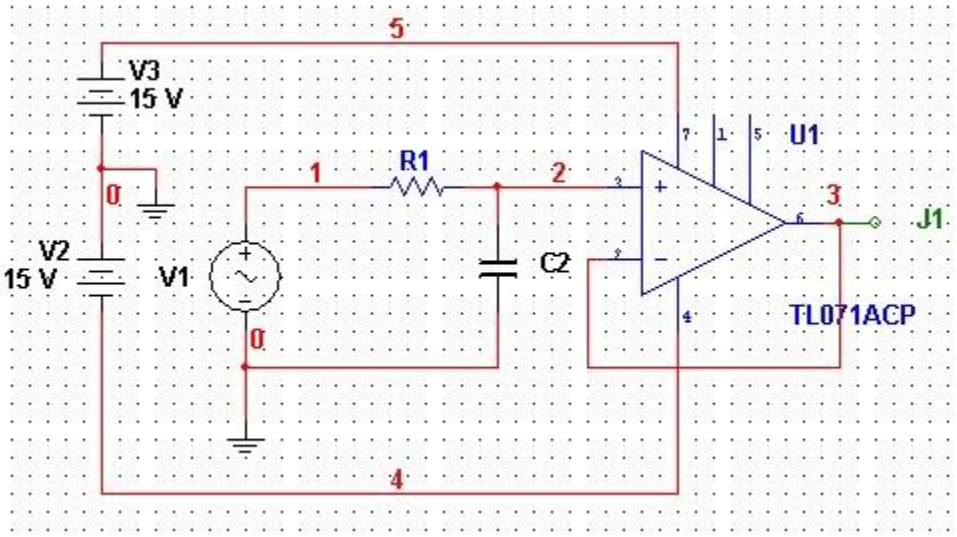



Figure 24: Single Pole Low Pass Filter With Output Buffer

In this circuit, the TL072 is configured as a noninverting unity gain buffer, with the output connected directly to the inverting input. If the input waveform is nominally 10Vpk, supplying the op amp with power rails at +/- 15V will provide more than sufficient headroom for the signal. The extremely high input impedance of the op amp ($10^{12}\Omega$ for the TL072) will prevent loading of the filter circuit, and the low output impedance should not encounter trouble driving the input of the computer. Despite the presence of the op amp, the equivalent circuit is same as the RC circuit of Figure 4, provide ideal op amp assumptions are made. The results can be found in Figure 25.

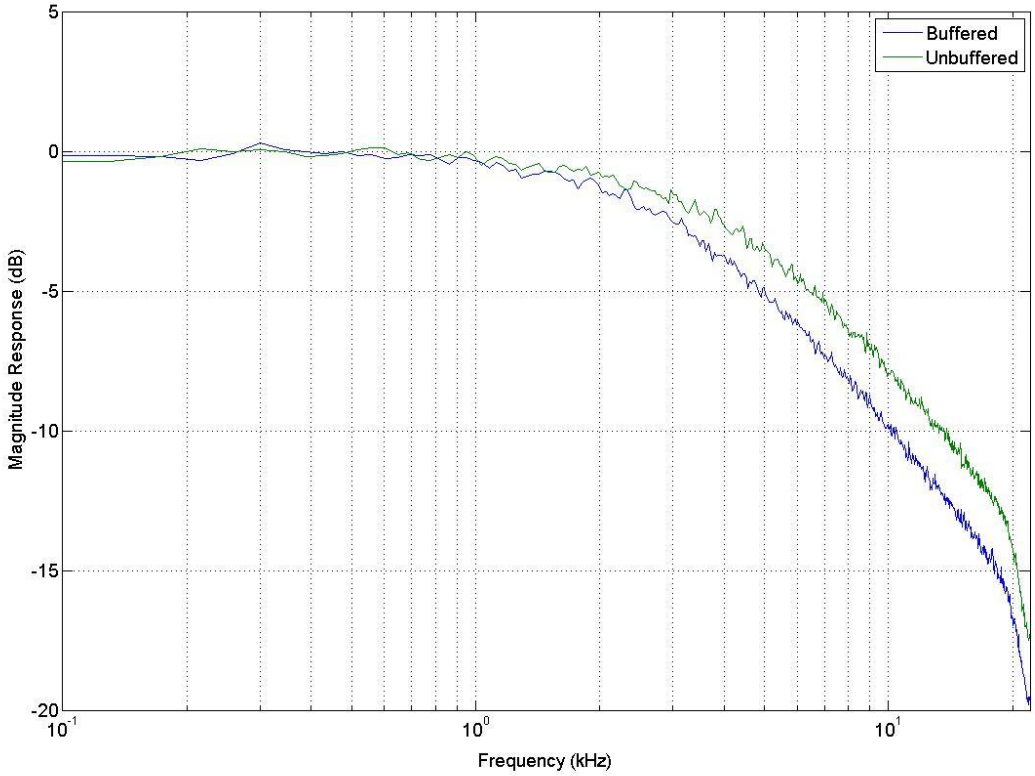



Figure 25: Effect of Filter Output Buffer

As can readily be observed, there are noticeable differences between the responses of these two filters. To check against the theoretical behavior, the cutoff frequency of the filter will be examined. The cutoff frequency of a single pole RC low pass filter is given by:

$$f_c = \frac{1}{2\pi RC}$$

The nominal component values used in this particular filter are R = 10kΩ and C = 4.7 nF. There for the cutoff frequency should be approximately 3.39kHz. Given that the component values in the actual filter are not exactly the nominal values, some divergence from the theoretical cutoff frequency is expected. The plot clearly shows that the buffered filter performs much more closely to the theoretical response than the un-buffered filter.

From these results we can conclude that loading effects were dramatically altering the frequency response of the filter. This resulted in the erroneous hypothesis that a simple capacitor model was insufficient to accurately model the filter response. In attempting to introduce inductive effects into the circuit, a sort of wild goose chase commenced.

Upon discovery of the error in the transfer equation, a new one was developed that was confirmed to be accurate. Using MATLAB, the continuous transfer function can be converted to a discrete time version at the sampling frequency of 44.1kHz. The conversion procedure will be explained in the following section. Using the discrete time transfer equation, a state space model can be extracted. The method for extracting a state space model from a transfer function will be described in a later section. Using this state space model in conjunction with the state space solver code, the digital and analog filters were rebuilt and re-tested. The results of this may be observed in Figure 26. The results shown here are reminiscent of the results from earlier testing, found in Figure 6. In Figure 26, we see that the digital filter follows the analog one closely up until approximately 8 kHz, at which point the slopes begin to differ. However, the digital filter matches the discrete time transfer function for the filter perfectly until aliasing issues near the Nyquist frequency come into play. Furthermore, at 10 kHz, there is only a 1 dB difference between the two. From this observation it can be concluded that the digital filter is functioning as it should, and that any differences between the analog and digital filters are due to the continuous to discrete time conversion. There are several methods for converting continuous time models to the discrete time domain, each with benefits and drawbacks. A necessary step will be assessing and comparing these methods so as to be able to select the method most suited to our application.

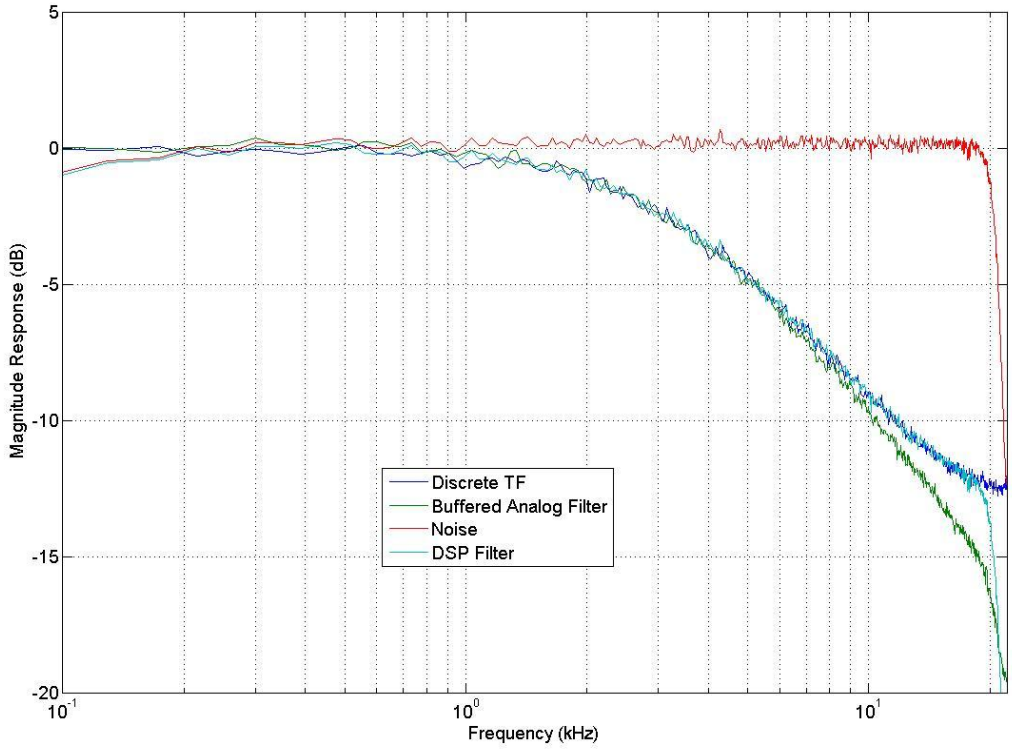



Figure 26: Revised RC Filter Response

# 7. Discretization Methods

When transforming from a continuous to a discrete time model, MATLAB allows for a variety of transformation methods. By default, the "zero-order hold" method is used. This is a rectangular step interpolation that assumes that a sample value is held constant for the duration of the sampling period. Next is the "first-order hold". This is a triangular approximation method of interpolating the signal over the sampling period. Another method is the impulse invariant method. The impulse invariant method discretizes the filter using an impulse train methodology. Its goal is to model, as accurately as possible, the impulse response of the continuous filter. Also available is the bilinear transformation (or 'Tustin' method). The bilinear transformation is a type of conformal mapping using to map between the s and z planes. At its core is an approximation of first order differential equations with difference equations. It is considered to be "generally more useful than the impulse invariant method." (Williams, 1986).

To determine which method of discretization will yield the best results, multiple factors must be taken under consideration. The filters vary is three primary ways: magnitude response, phase response, and algorithm. Due to the manner of implementation we know that our filter will yield a linear phase-frequency characteristic, and therefore we will largely ignore phase, unless it is found to be relevant later.

Returning to the simple passive first-order low pass filter of Figure 4, a continuous state space model in MATLAB was used to test the various discretization methods. Figure 27 shows the results.

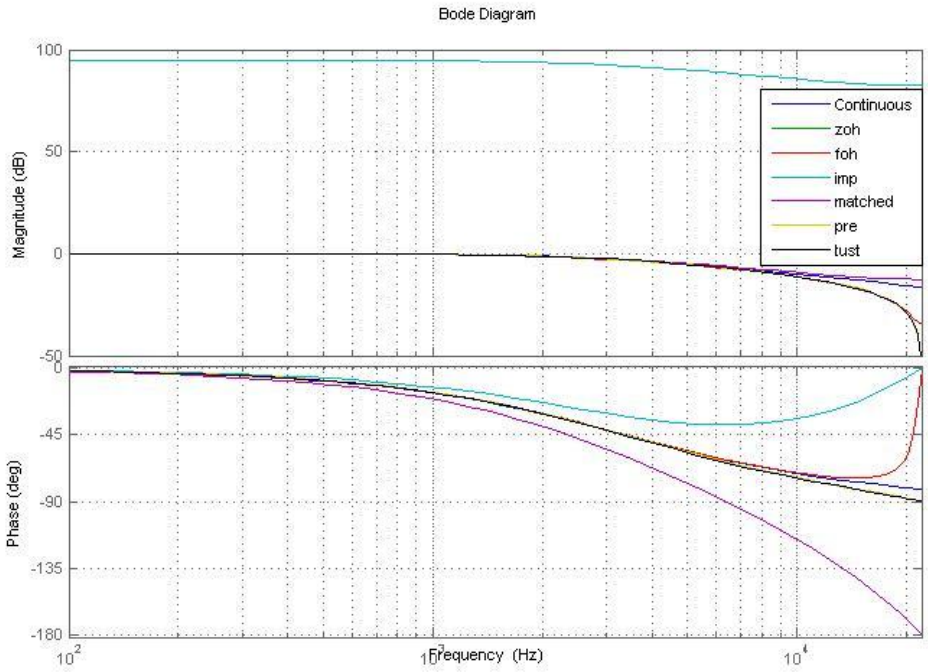



**Figure 27: Results of Various Methods of Discretization of a Continuous RC Filter Model**

As can be seen, the impulse invariant method can be rejected outright, as its DC magnitude response is nowhere near that of the continuous filter. We see strong similarities in the responses of the Tustin, pre-warped Tustin, and first-order hold methods, and between the zero-order hold and pole-zero matched methods. It appears that, though none of the methods provide a perfect representation of the continuous filter, the zero-order hold and pole-zero matched methods provide the closest approximation, given the sampling rate and bandwidth of interest.

To further investigate, the effects of various discretization methods should be tested on a variety of filter types to observe any differences. First, the passive RC low pass will be inspected.

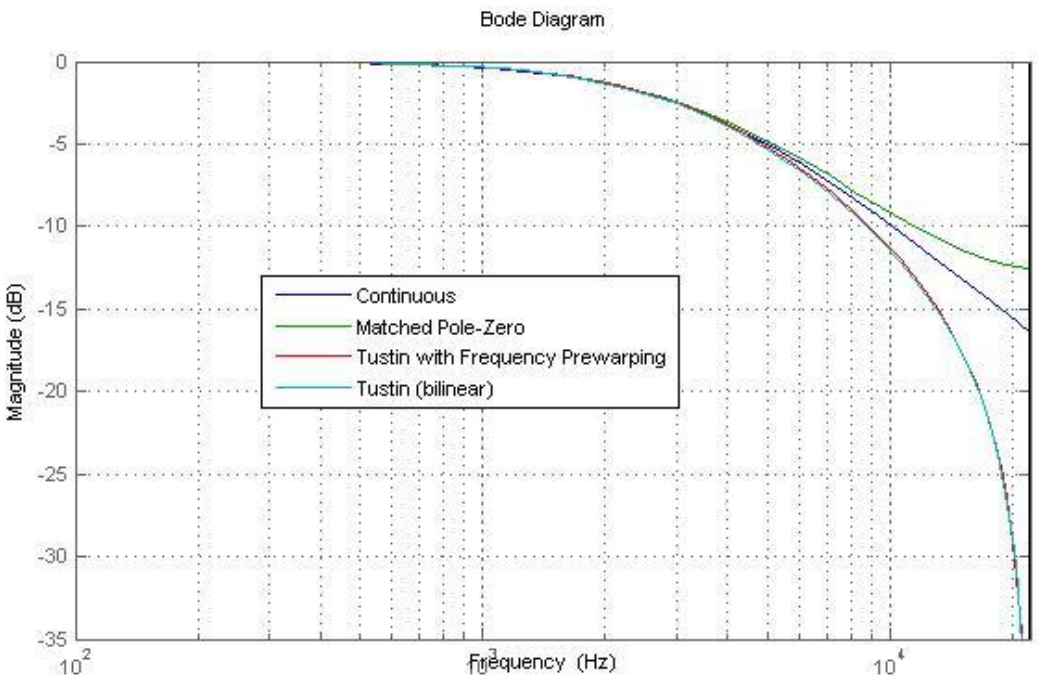



**Figure 28: Continuous and Discretized First Order Low Pass Filter Magnitude Response**

Figure 28 shows the magnitude response of the passive RC filter with the responses of the matched zero-pole, Tustin, and pre-warped Tustin discretized filters. As can be seen, the pre-warped Tustin and Tustin discretized filters behave almost identically for this filter. At the -3dB point, all of the filters behave the same. Throughout the bandwidth, it appears that the zero-pole matched method provides a closer estimate of the continuous filter. Next, a Sallen-Key topology low pass filter will be tested.

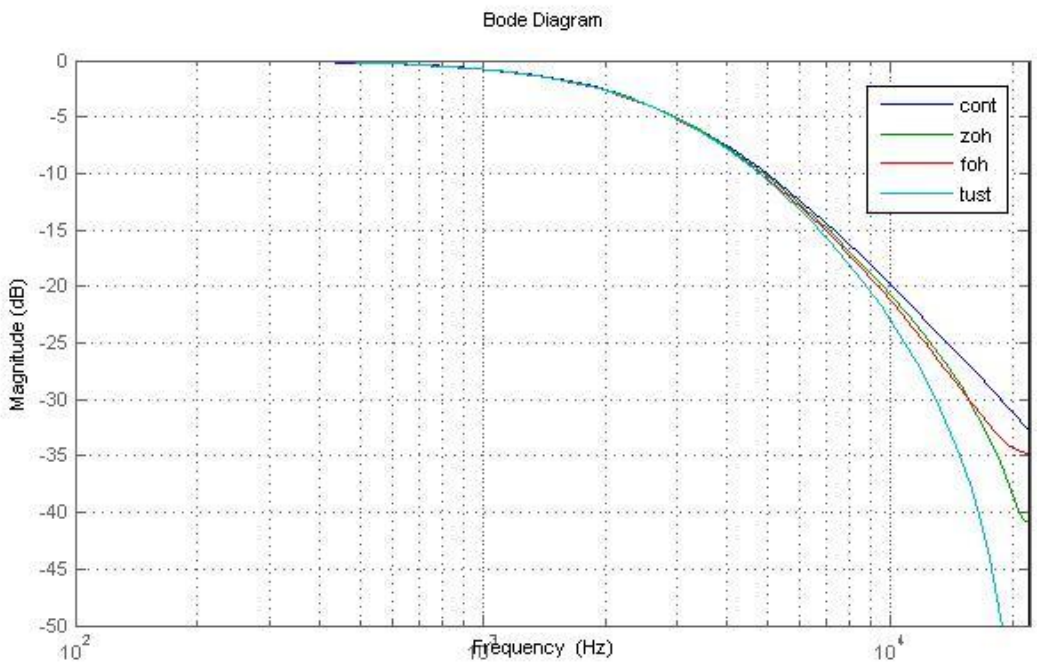



Figure 29: Continuous and Discretized Sallen-Key Low Pass Filter Magnitude Response

Figure 29 shows the magnitude response of a Sallen-Key low pass filter and its response as discretized by the zero-order hold, first-order hold, and Tustin methods. As can be observed, the zero-order and first-order hold responses are the most accurate, with the Tustin response differing significantly toward the Nyquist frequency. Figures 28 and 29 seem to imply that the accuracy of the Tustin method towards the Nyquist frequency reduces with filter order, but this has yet to be seen. Tests involving higher order filters must be performed before such a conclusion can be drawn. Lastly, a second-order Sallen-Key band pass filter will also be analyzed. The magnitude response can be observed in Figure 30.

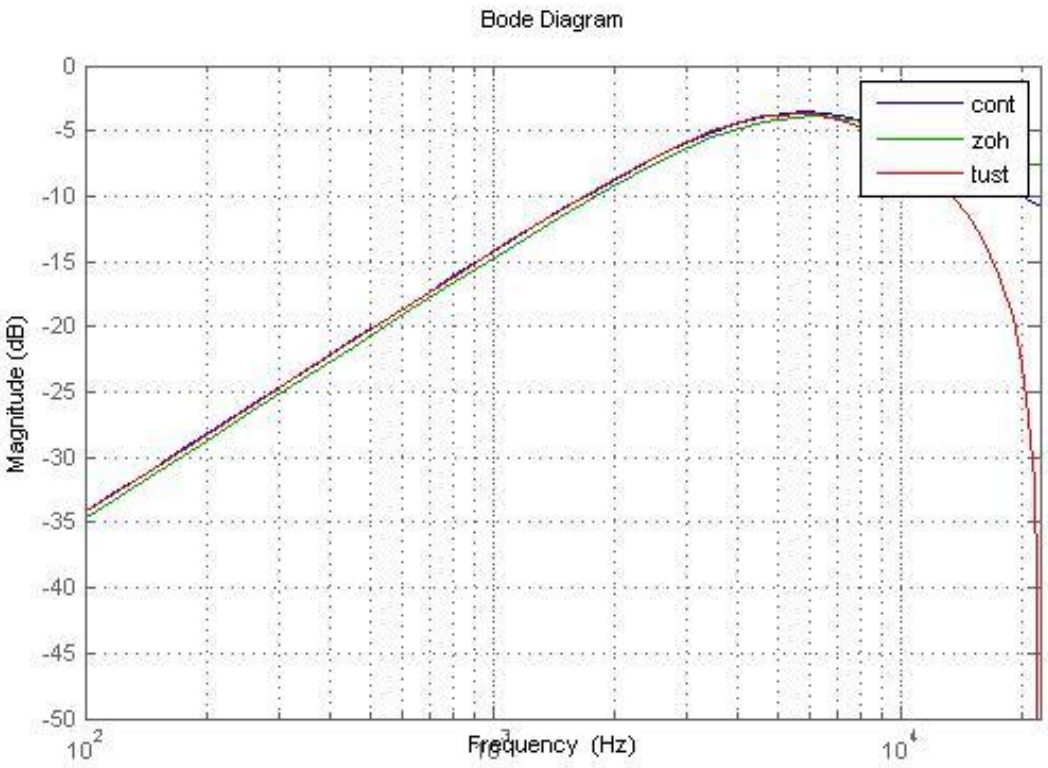



Figure 30: Continuous and Discretized Sallen-Key Band Pass Filter Magnitude Response

Again, the superiority of the zero-order hold algorithm may be observed relative to the Tustin method. It appears that the Tustin method, though highly accurate over most of the bandwidth, suffers from serious aliasing issues.

## 7.1 Frequency Issues

Although it appears that, with respect to the frequency response, the zero-order hold method retains more accuracy throughout the bandwidth that has been examined, that bandwidth must also be questioned. The sampling rate that has been used is 44.1kHz, which is the most common sampling rate for digitized music. Most notably, it is the standard sampling rate for compact disks (National Digital Information Infrastructure & Preservation Program, 2008). This sampling rate was chosen because the upper limit of human hearing is considered to be approximately 22kHz. 44.1kHz is just over twice 22kHz. According to the Nyquist sampling theorem, for a sampled analog signal to contain all of the information of the original signal and be reconstructable, the sampling rate must be *at least* twice the highest frequency of interest contained in the original signal (Olshausen, 2000). Thus, a bandwidth of approximately 22 kHz is considered acceptable for consumer audio applications. If the application of the filters and processing addressed by this project is musical equipment, such as guitar effect processing, such a high bandwidth may not be necessary. The highest fundamental frequency of a 24-fret guitar is 1,318.51 Hz. This does not mean that sampling at 2.8kHz will be sufficient, as the harmonics of an instrument are what give its sound its distinct characteristics. Tests show that guitars have essentially no frequency information above 15kHz, and very little above even 10kHz. Thus, it seems reasonable that an effect processor of a guitar signal need not accurately reproduce information all the way up to 22kHz. Thus, sampling at 44.1kHz will be *oversampling*, which, if high enough, should negate any aliasing issues introduced by either the sampling or filtering.

Let us examine the effects of oversampling on the effect of aliasing error. To test this, we will examine the Sallen-Key low pass filter of Figure 15. In Matlab, we will construct the continuous time model and then convert it to discrete time at sampling frequencies of 44.1kHz, 48kHz, 96kHz, and 192kHz. The TMS320C6713 is capable of sampling up to 96kHz. It cannot sample at 192kHz; this is only

provided for reference and out of curiosity. Figure 31 shows the results of the Bode plot of these systems.

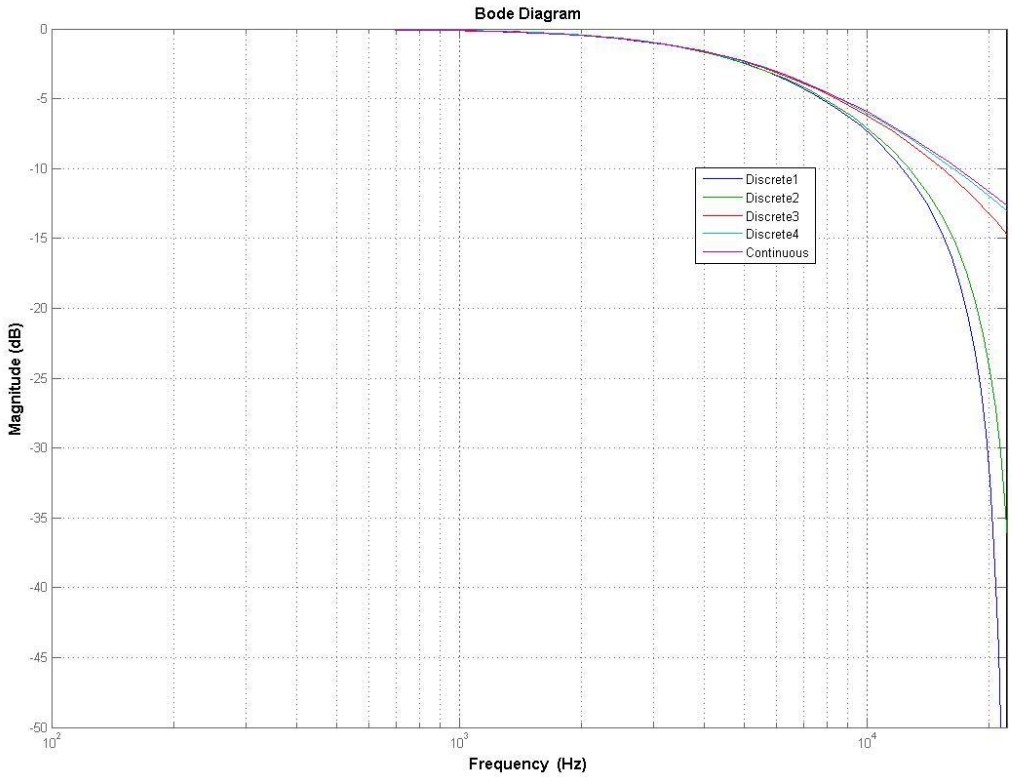



Figure 31: Sallen-Key Low Pass Filter Discretized at Various Sampling Frequencies

For all of the discrete systems, the magnitude approaches negative infinity as the frequency approaches the Nyquist frequency. By increasing the sampling rate, the aliasing effects are not removed or negated, but merely "pushed" to a higher frequency, ideally out of our bandwidth of interest. As can be seen, with a sampling frequency of 192kHz (Discrete4), the differences between the discrete and continuous systems are negligible. Even at 96kHz, there is no more than a 2.5dB difference at 22kHz. Since the frequency content of the guitar is well below this, it should be safe to assume that the aliasing error would be unnoticeable. This will ultimately need to be determined in qualitative listening tests.

If an oversampling means less error in the bandwidth of interest, why not sample as fast as possible? To answer this, we need to understand how the code is operating. The sample processing is being performed in an interrupt service routine (ISR). The code enters this routine every time the sampling clock sets an interrupt flag, which is at the sampling rate. The processor on the TMS302C6713 operates at 225MHz. The number of available clock cycles in which to process a given sample is given by:

$$\frac{Cycles}{Sample} = \frac{Processor\ Speed}{Sampling\ Rate}$$

Thus, at a sampling rate of 44.1kHz, there are approximately 5100 clock cycles available to process each sample (without compiler optimization or pipelining). At a sampling rate of 96kHz, there are only about 2300 clock cycles available. Thus, the limitation is the complexity of the code. If it can execute in less than the available time, for a given sampling frequency, we can sample at that frequency.

## 7.2 Algorithmic Issues

For a fixed value filter, the discretization method is irrelevant as long as it produces the desired results, as this does not have to be done in real time or even on the DSP at all. The method with which the filter is discretized only becomes an issue when the filter is variable. If a filter is variable it follows that the coefficients in the representation will vary in time. Since there is not a linear mapping between the values in the continuous and discrete time models, new discrete model coefficients would have to be calculated in real time. Since the operations to convert a continuous to a discrete model are rather complex, the method with which the model is discretized will matter from a computational perspective. Given the simulated frequency responses of various discretization methods, the Bilinear Transform method and the Zero-Order Hold method will be investigated further.

## 7.3 Filter Modification

In analog filter circuits, it is common to have user-variable controls, usually in the form of potentiometers. These control often govern parameters such as gain level, filter corner or center frequency, filter width (in the case of band pass or notch filters), and more. In effects processors for musical instruments, controls are nearly ubiquitous. Therefore, to model a useful analog circuit for these purposes, it is necessary to be able to handle variable values in the models. It is rather easy to map the movement of a control to a changing component value in an analog circuit, but as we have seen, there is not an obvious relationship between the continuous and discrete time models for a given filter. Thus, to determine how to add user-variable controls, we must determine how to transition from the continuous model to the discrete one.

## 7.4 Bilinear Transform

Though there are several methods for the conversion of continuous time model to discrete time, the bilinear transform produces the most desirable results. At its core is a method of trapezoidal interpolation of discrete sample values. During the transformation process, frequencies in continuous time are mapped from the s domain to the z domain. To account for distortion of frequency in the mapping process, the bilinear transform allows for the "pre-warping" of frequencies. This, in essence, sets a static point in the mapping so that magnitude and phase responses for both the continuous and discrete filters will be equal at this pre-warp point. This frequency is commonly chosen to be the corner frequency, for a high or low pass filter, or the center frequency of a band pass filter, for example.

The bilinear transform is found as follows:

$$H(z) = H(s)|_{s=2f_s\frac{z-1}{z+1}}$$

Where s=jω and $f_s$ is the system sampling frequency.

The MATLAB function c2d() defaults to the 'zero-order hold' method of discretization, though the bilinear transform method may be specified. The bilinear transform may be applied to either transfer functions, or to a state space model directly. In the case of direct conversion of state space models from continuous to discrete, the following algorithm is used:

$$\boldsymbol{A}_d = \left(I + \left(\frac{1}{k}\right)\boldsymbol{A}\right)\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}$$

$$\boldsymbol{B}_d = \frac{2k}{r}\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}\boldsymbol{B}$$

$$\boldsymbol{C_d} = r\boldsymbol{C}\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}$$

$$\boldsymbol{D}_d = \left(\frac{1}{k}\right)\boldsymbol{C}\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}\boldsymbol{B} + \boldsymbol{D}$$

where **A**, **B**, **C**, and **D** are the matrices of a continuous time state space model, $\mathbf{A}_d$, $\mathbf{B}_d$, $\mathbf{C}_d$, and $\mathbf{D}_d$ are the matrices of the discrete time state space model, k=2*fs, and $r=(2/k)^{½}$. (The Mathworks, 2010).

From visual examination of the above algorithm, several observations relevant to implementation can be made. There is a single matrix inversion operation that must be performed. Prior to the inversion, the multiplication of a matrix by a scalar must be performed. After the scalar multiplication, matrix addition must be performed. Once the $(I-(1/k)A)^{-1}$ term has been calculated, all discrete state space matrices may be computed.

This presents the issue of matrix inversion. We know from linear algebra that for a matrix to be invertible, it must have a non-zero determinant (Lay, 2003). Thus, it is probably a reasonable precaution to calculate a given matrix's determinant before attempting to invert it. For example, the determinant of a 2x2 matrix is given by:

$$\det[A] = ad - bc$$

Since the inverse of a matrix is given by:

$$A^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

It can be seen that if the determinant is zero, the inverse cannot be solved.

There is potentially an issue here in that all of the above have high order computational complexity. Both determinant solution and matrix inversion traditionally have complexity $O(n^3)$, though faster algorithms do exist. Matrix multiplication complexity is dependent on the size of the matrices involved, but it still has a high order of complexity. Despite the demanding computational

requirements, this only becomes a worry if the bilinear transformation must be calculated *on the fly*. In other words, all discrete state space values may be pre-calculated, either prior to compiling or in the code prior to execution of any actual signal processing. Thus, the only time the bilinear transformation will need to be calculated during operation would be if values in the continuous model change during operation. For instance, if there were a variable resistor in an analog filter circuit which was able to be adjusted by a user, mapping this response to the discrete domain would involve re-computing the bilinear transform to determine the new values of the discrete state space model.

Though this still requires significant amounts of computation, it should be noted that the time domain resolution of any user-variable controls need not be as high as for the signal itself. If we are sampling at 44.1kHz, we need not refresh controls at this rate, thus, computation of new state space values need not be performed in the interrupt service routine. A potential methodology is proposed:

1) Within the infinite 'while' loop, there exist a routine to monitor the state of an input, perhaps the output of a rotary encoder interfaced with the DSK.
2) If this input changes, the value is noted and an equivalent analog value is stored in the continuous time model.
3) Calculation of the new discrete time coefficients begins, using a bilinear transform algorithm. This consists of:
    a. Calculation of the determinant of the A matrix to verify non-singularity.
    b. Assuming A is not singular, Perform scalar multiplication, and subtract result from properly-sized identity matrix (if A is an n*n matrix, then I must be as well).
    c. Invert the result. Several algorithms are under consideration and will be discussed in a later section.

d. Given the result of the inversion, calculate the Ad, Bd, Cd, and Dd matrices using a combination of techniques.

4) When all new matrices have been calculated, set a flag to notify the ISR that new values are ready.
5) When the next ISR occurs, the state space solver function will compute the output using the new matrices.
6) Continue monitoring input until change occurs.

It has been noted in studies on computer-human interaction that, “latency of more than about 75 to 100 ms harms user performance for many interactive tasks.” (Jacko & Sears, 2003) Therefore, the goal for modifiable parameter updating will be that it be performed in under 75 ms, if possible. In fact, there may be noticeable and annoying latencies at well under 75 ms of delay, so it will be attempted to minimize this latency. Thus, the values would have to be updated at a rate of no less than 13.3 Hz. Given that the clock speed on the development board runs at 225MHz, this means that operations should be completed in no more than 169,172 cycles. The actual limit will be lower as the operation will be interrupted by the ISR, whose operations take priority. Though this limit seems very high, it must be restated that many of the operations to be performed are of a high computational complexity. Time to complete is also highly dependent on the size of the matrices, which is governed by the complexity of the analog circuit being modeled.

The issue of complexity can be partially tackled through careful circuit analysis. In many cases it will be possible to isolate the section of the circuit containing a control element and treat it as a two port network of minimum order. The order will still depend on the circuit itself, but by this method it is possible to break the overall circuit into a series of cascaded state space models wherein only the sections with controls will need to be recalculated using the bilinear transform method.

## 7.5 Zero-Order Hold

The Zero-Order Hold (heretofore referred to as ZOH) method is essentially a rectangular approximation method that assumes that sample values are held constant over the course of the sampling period. According to the MathWorks, the ZOH method is appropriate when "You want an exact discretization in the time domain for staircase inputs." (The MathWorks, 2010)

To compute the discrete versions of the matrices of a state space model, the following algorithms are used:

$$\boldsymbol{A}_d = e^{\boldsymbol{A}T}$$

$$\boldsymbol{B}_d = \left(\int_{\tau=0}^{T} e^{\boldsymbol{A}T} d\tau\right)\boldsymbol{B} = \boldsymbol{A}^{-1}(\boldsymbol{A}_d - I)\boldsymbol{B}$$

$$\boldsymbol{C}_d = \boldsymbol{C}$$

$$\boldsymbol{D}_d = \boldsymbol{D}$$

The ZOH method appears far less complex than the bilinear transform method. For one, the discrete C and D matrices are the same as the continuous time matrices, meaning zero computation. Then again, the computation required to calculate the C and D matrices in the bilinear case is certainly not trivial, but only involves the multiplication and addition of matrices and scalars. For low-order matrices as likely to be used, the amount of computation would be small. The ZOH method differs in that it involves the calculation of a matrix exponential. The exponential function is defined as follows:

$$e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k$$

This is a power series which always converges. Thus the exponential function is fully defined. The matrix exponential is commonly calculated with the Jordan form of a matrix. Achieving the Jordan form is unfortunately not simple or computationally light. The Jordan form is as follows:

$$\boldsymbol{A} = \boldsymbol{P}^{-1}\boldsymbol{B}\boldsymbol{P}$$

where **B** is a diagonal matrix, and **P** is an invertible matrix composed of the eigenvectors of **A**. Firstly, the matrix **A** must be diagonalizable. Not all matrices are. Secondly, the computation of matrix eigenvectors is extremely complex, difficult, and computationally heavy, as it is an iterative process (Press, Teukolsky, Vetterling, & Flannery, 1999). Thus, it would be essentially impossible to compute a conversion using the ZOH method in real time on a DSP processor.

## 7.6 Choosing a Methodology and Algorithm

Having had an introduction to both the bilinear transform and zero-order hold discretization methods, it is necessary to determine which will both emulate the analog filter acceptably and be less computationally complex. Since it is anticipated that the system will need the ability to handle time-varying filters, the filter coefficients may need to be recalculated during operation. Therefore, the method with which the continuous system is transformed has the requirement of being readily and accurately computed.

Though ZOH discretization seems to be truer to the continuous time filter for more of the frequency spectrum, the frequency at which both discrete filters begin to diverge from that of the analog may be high enough that the discrepancy can be ignored.

If this assumption can be made, the algorithms must be examined. The ZOH approach requires the computation of a matrix exponential, which theoretically requires the convergence of a potentially infinite series. There are certainly ways of approximating the result of the series, but this still assumes that the matrix in question is diagonalizable, which it may not be. Furthermore, the approximations may not be accurate enough to be useable.

On the other hand, the bilinear transform requires that a matrix inverse be calculated. The algorithmic computation of the matrix exponential is a well-trodden path. Modern 3-d graphics rely heavily on the computation of the matrix inverse, and several algorithms exist. There are several common algorithms: Gauss-Jordan elimination, the Strassen algorithm, the Coppersmith-Winograd algorithm, and the application of LU decomposition. Gauss-Jordan elimination and LU decomposition are more common and require about the same amount of computation (Press, Teukolsky, Vetterling, & Flannery, 1999). The Coppersmith-Winograd algorithm is theoretically the fastest, but it is considered only really advantageous when applied to very large matrices (Robinson, 2005). The Strassen algorithm,

despite its minor improvement in complexity, is better applied on processors optimized for the task (Karkaradov, 2004). Gauss-Jordan elimination and LU decomposition are well established, albeit slower algorithms that may be used for the computation of the matrix inverse. The focus will be directed at determining which of these algorithms is optimal for this application.

In calculating the inverse of a matrix, Gauss-Jordan elimination and LU decomposition "have practically the same operations count" (Press, Teukolsky, Vetterling, & Flannery, 1999). LU decomposition is faster overall if the calculated inverse is then going to be multiplied by another matrix. Since for this application the inverse will be multiplied by another matrix for the computation of all discrete state space coefficients, it seems that the LU decomposition method would be a superior approach.

In conclusion, though the ZOH method seems to provide better results than the bilinear transform, the range of frequencies for which the bilinear transform is less accurate are less important, and since it is necessary that the coefficients be modifiable yet true to the original continuous time system, the bilinear transform has been chosen.

# 8. Audio Testing

Before getting too far into development based on magnitude plots alone, it was necessary to conduct qualitative audio listening tests to determine if the filters were producing the desired results. To do this, it was decided that it would be more natural to listen to filtered music instead of noise, and thus easier to differentiate between the filters. To do this, an mp3 player was connected to either the analog or the digital filter, which was connected to the computer's line in jack as previously described. As currently the analog and digital filters have different DC gains, the output of the mp3 player was adjusted between tests such that analog and digital filters output signals at the same magnitude.

The filter under test is a Sallen-Key band pass filter. The filter was discretized using the Bilinear transform method. Using the noise test setup, the magnitude responses to wideband 0dB noise were first compared.

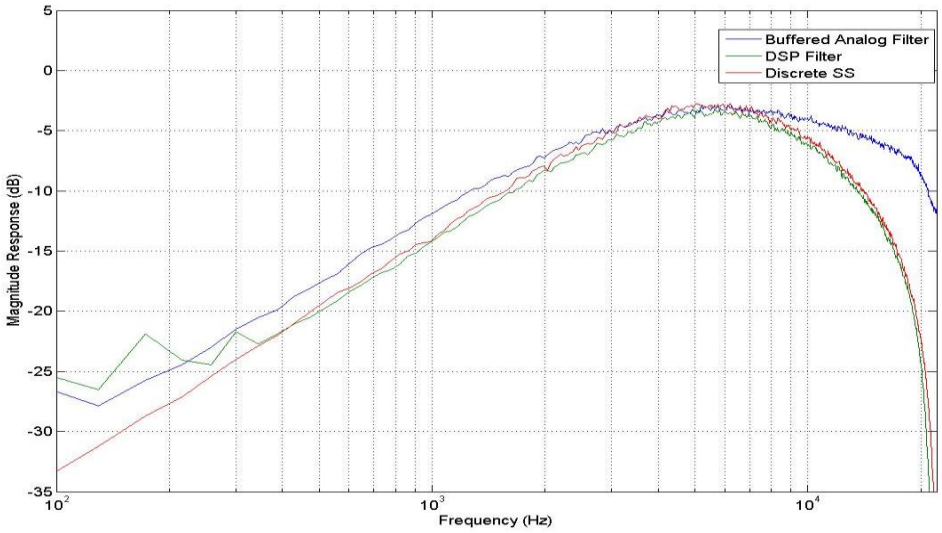



Figure 32: Sallen-Key Bandpass: Analog vs. DSP

As can be seen in Figure 32, there is a significant difference in the frequency response of the analog and digital filters above 10kHz. Based on this, it seems reasonable to predict that a listener will

be able to differential between the two filters by the relative lack of frequency content between 10kHz and 20kHz.

The first ten seconds of a song (Built To Spill's "Pat" on the album There Is No Enemy) was chosen as the audio sample for the test. The test setup was as is found in Figure 33:

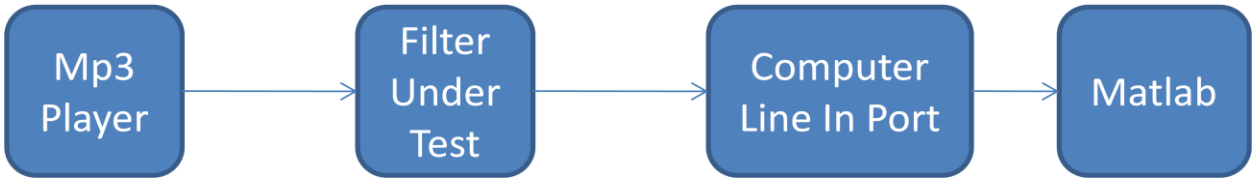



Figure 33: Audio Signal Test Setup

Once the sample was recorded by Matlab, the samples were normalized so that differences in overall magnitude would not influence the test. The original unfiltered sound sample, the sample filtered by the analog filter, and the sample recorded by the digital filter were all recorded for comparison. The following is the magnitude response plot of the test.

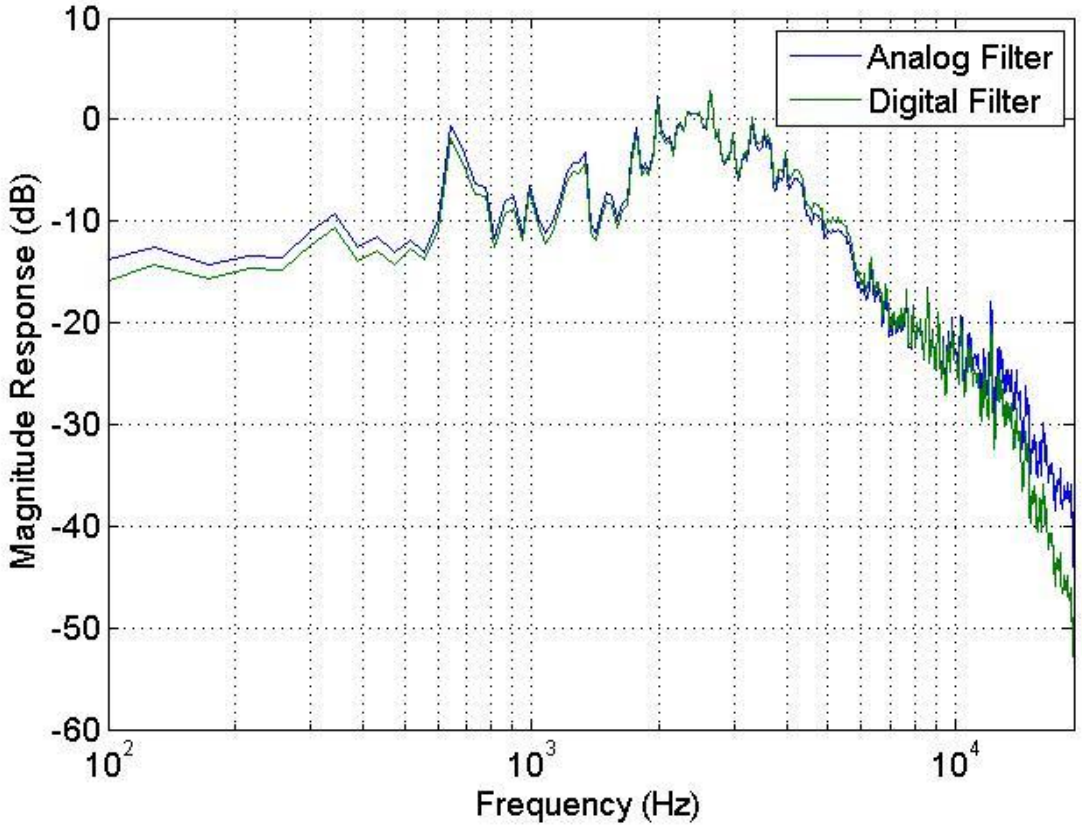



Figure 34: Magnitude Response of Band Pass Filter with Audio Input

The response shown in Figure 34 is about what was expected. There is noticeably more attenuation in the frequency range near 20kHz. The audio listening tests however were somewhat surprising.

For the preliminary listening tests, the sound samples were exported as .wav sound files using the Matlab wavwrite() function. Then, the audio files were blindly listened to in random order using a computer media player. The listeners were my advisor Professor Michalson, who has extensive experience with audio recording and processing, and I. We both observed that one of the sound samples had more noticeable high frequency content, what is often described in the audio world as “ice pick highs”. There was “sharpness” to this particular sample, that we found to be less musically desirable, and a rounder and softer quality to the other. The sample with the “sharper” high frequency

content was not in fact the analog filter, but the digital one. Based solely upon the magnitude response plot, one might expect that there would be more noticeable high frequency content in the analog-filtered sample, but we found it to be the opposite.

In multiple blind tests, the qualitative result was that the analog filter sounded smoother, less harsh, and overall, better. The results were initially a bit confusing, as it was expected that the analog filter would have a more pronounced high frequency characteristic. It is theorized that the high frequency content sounds as though it has more high frequency content *due to* the greater attenuation slope towards the Nyquist frequency. Thus, there is more apparent separation of frequency, resulting in the appearance of more pronounced highs in the 10-13kHz range. The analog filter, due to its smaller attenuation slope, sounds smoother, and there is not the pronouncement of high frequencies that occurs in the digital filter.

This result is in line with many of the complaints of digital modeling devices and digital signal processors in general. It begs three questions: is the apparent difference between the analog and digital filter contained solely in their high frequency divergence? Will another transformation method sound better; that is, closer to the analog result? Also, can the undesirable effects of the bilinear-transformed digital filter be negated?

First, the audio samples will be processed with a sharp-cutoff finite impulse response (FIR) filter with an order of 559. It has a corner frequency of 5.8kHz. This will essentially remove all frequency content just after the peak of the band pass filter and will allow us to compare the samples independent of their divergent high frequency content. The filter's magnitude response is as is found in Figure 35.

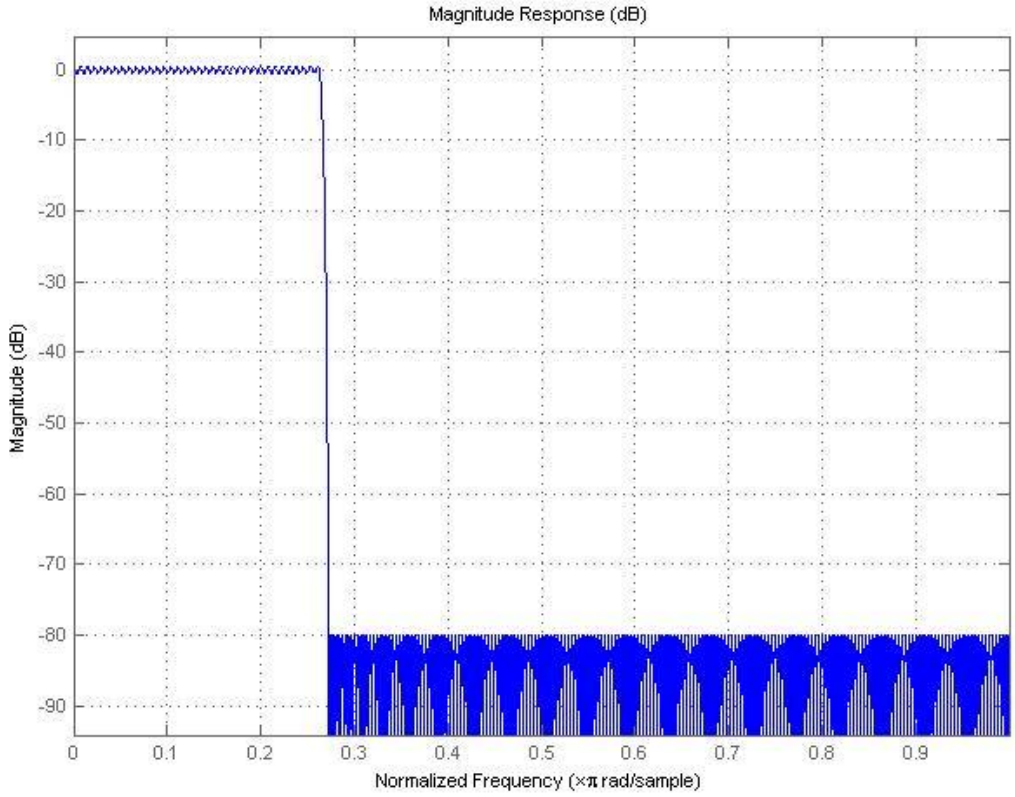



Figure 35: Sharp Cutoff Low Pass Filter Magnitude Response

As can be observed, there is 80dB of attenuation in the stop band, and minimal ripple in the pass band. The slope of the attenuation is very steep. Both samples were processed with this filter in Matlab and subjected to the blind audio test. It was found that no distinction between the two could be found, thus confirming the hypothesis that the only perceptible difference between the two filters lie in the high frequency spectrum. The plot of the magnitude responses of the two resultant audio file confirms this:

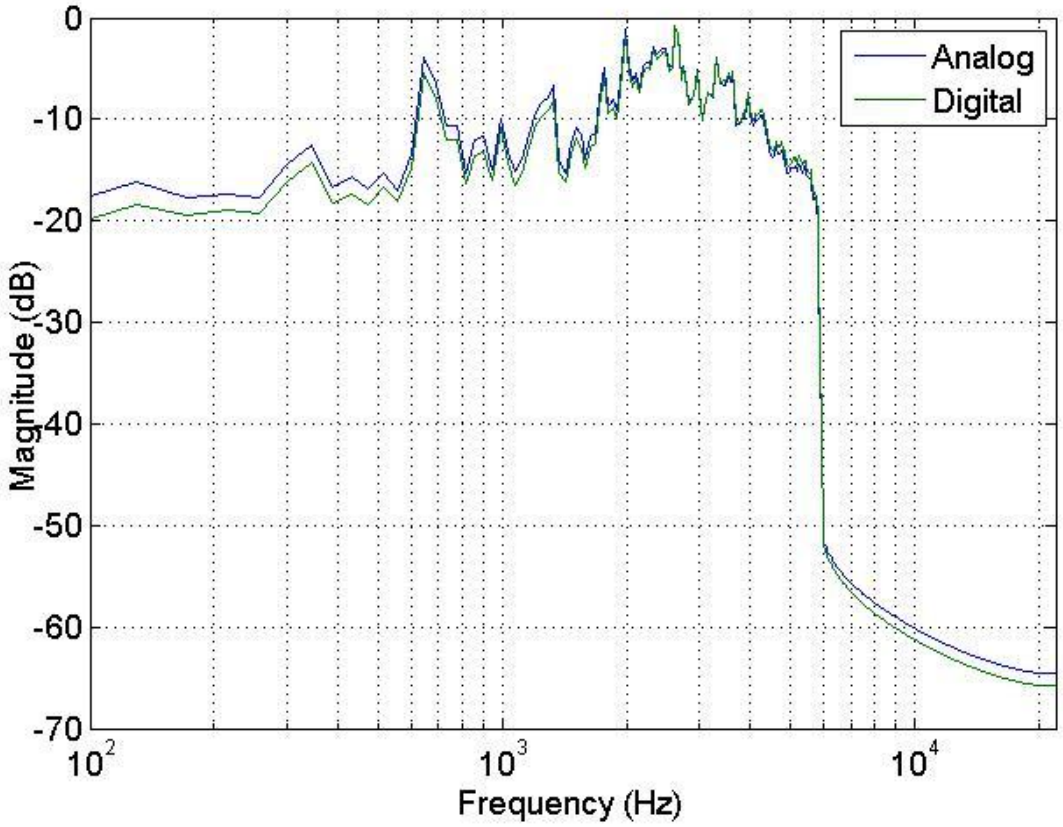



Figure 36: Magnitude Response of Band Pass Filter Audio after Low Pass Filtering

As can be seen in Figure 36, the two are now almost indistinguishable. Thus, if we are to construct a digital filter that is indistinguishable from the analog filter, we ought to in some way correct the high frequency disparity. The rest of this section will be concerned with just that.

First, other the zero order hold transformation method will be subjected to the same audio test. We have already compared the zero order hold and bilinear transform on the grounds of magnitude response and computational issues, so let us now compare them in terms of subjective sound quality.

To do this, the same digital filter topology was utilized, but the filter coefficients were discretized using the zero order hold method. The test setup was identical to that used in the tests of the analog and bilinear-digital filters. Listening to the results of this test versus both the analog filter

and the bilinear transform-discretized filters, it was found that no difference could be perceived between the ZOH-discretized filter and the analog filter, and the distinction between the ZOH and bilinear filters was the same as that of the analog and the bilinear filters. Observing the magnitude response of all three of these filters in Figure 37, we find that they are consistent with our aural expectations.

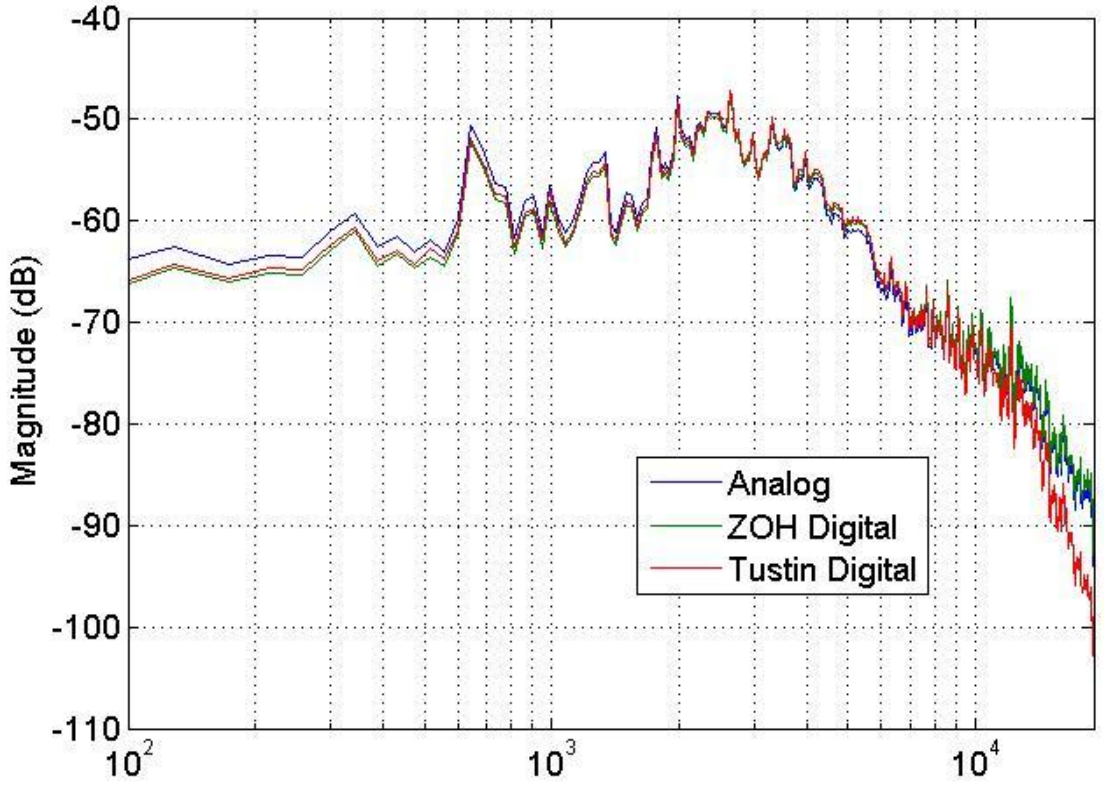



Figure 37: Audio Test: Bilinear vs Zero-Order Hold vs Audio Magnitude Response

Though it is difficult to detect, there are some minimal differences between the ZOH filter and the analog filter. The plot only extends to 19kHz, as there was essentially no frequency content above 19kHz, most likely due to the mp3 encoding algorithm used to encode this particular sound sample. It can be readily observed that there are substantial differences between the Bilinear digital filter and the other two. At 19kHz, we see a full *10dB* difference between the bilinear and the others. Given the

results of the tests involving the cutoff filter and now the ZOH-discretized filter, it can be concluded that the shape of the magnitude response near the Nyquist frequency in the bilinear case is responsible for audible separation from the analog filter.

It is only necessary to use the bilinear transform method of discretization in the case of *time-varying* filters. Given that a signal processing circuit will usually include both time-varying and time-invariant filters, it is unnecessary to exclusively use the bilinear method of discretization. In fact, given the results of this section, it can be concluded that it would be beneficial to use the ZOH method instead for all time-invariant filter sub circuits. Nevertheless, nearly all signal processors one might wish to model will have some time-varying aspect(s). It is clear now that filter sections transformed with the bilinear transform method are the "weak link". With this in mind, how might the shortcomings of bilinear-discretized filters be avoided?

The first, and most obvious response is to increase the sampling rate, thus pushing the negative effects outside of the bandwidth of interest, or even beyond the range of human hearing. Figure 38 below shows the band pass filter as sampled at 96kHz, more than twice the frequency that has been used.

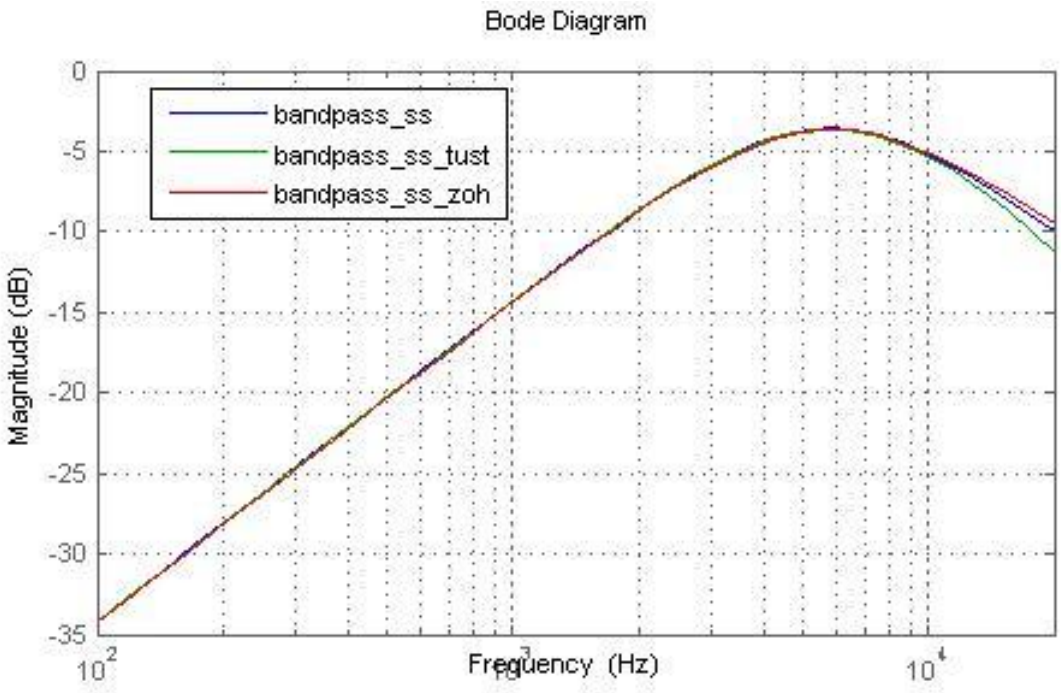



Figure 38: Sallen-Key Band Pass Filter Sampled at 44.1kHz, 96kHz

As can be observed, the magnitude difference between the bilinear-digital and the analog filter at nearly 20kHz is a mere 1.3dB. This is a much more desirable result, but it would be best to avoid reliance on oversampling to avoid the shortcomings of the bilinear method, as it is yet unknown how much computation will required for processing, and an increase in sampling rate means a decrease in the number of processor cycles available for processing in the ISR. Since scalability is a goal of this project, we will refrain from using oversampling to solve this problem. If this method is used for an application in which there is plenty of available computation time, oversampling may be used, but it would be best to make this an option as opposed to the solution.

Returning to some of the magnitude plots (Figs. 28, 29, 30, 32), the notable distinction is always the high rate of attenuation near the Nyquist frequency. The question is posed: what if prior to filtering by a bilinear-discretized, the signal was altered in such a way so as to negate its undesirable effects? If the signal were first passed through a filter that pronounced the frequencies lost in the bilinear

transformation, so as the result would be a magnitude response that matched that of the analog filter, the bilinear-transformed digital filter might not be subject to the shortcomings that have thus far been observed, and we would be able to develop accurate time-varying filters. In certain situations, this method might prove useful and if implemented correctly, could still be relatively efficient. For instance, observe the system diagram of Figure 39:

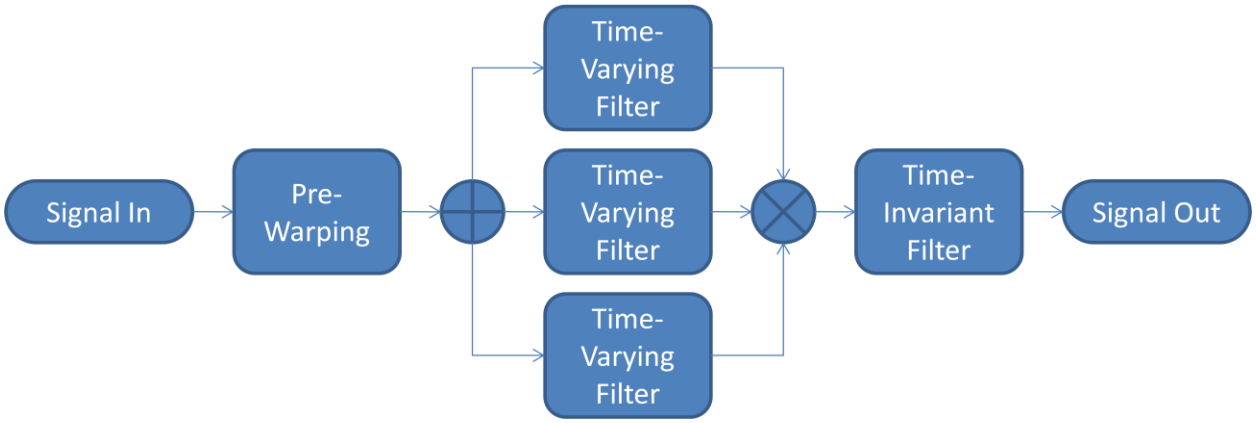



Figure 39: Block Diagram: Prewarping in Parallel Filter System

This diagram might represent the digitally implemented model of an analog graphic equalizer, a signal processing device that allows a user to boost or cut various fixed frequencies. In such an application, the system can be thought of as sending a signal through several parallel band pass filters with fixed center frequencies, and then summing the output of the various filters. In such a case with multiple same-order filters which have been discretized in the same manner, the signal may only need to be "pre-warped" once before being fed to all of the filters. This particular implementation would be particularly conducive to the pre-warping concept as it only necessitates that the pre-warping be performed once for the entire system. It would not however be as efficient in a system where there were several filters in series, as the signal would have to be pre-warped before each new filter. Such an implementation may be observed in Figure 40:

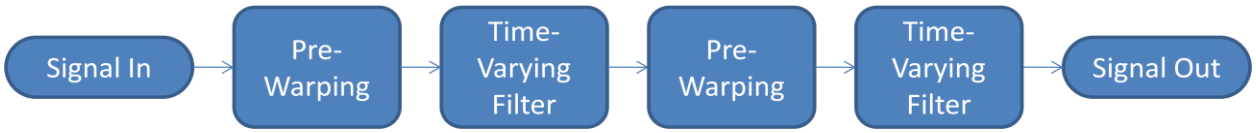



Figure 40: Block Diagram: Prewarping in Series Filter System

It can be extrapolated that for each time-varying filter in a series configuration, pre-warping must be performed. Luckily, in guitar signal processing applications, series configurations such as this are relatively rare.

More common still is a configuration where in there is one time-varying filter with multiple user-variable controls. This would be modeled as a single filter, and thus would only necessitate one instance of pre-warping. The most common of this is the guitar amplifier "tone stack", a frequency-shaping variable filter that may be found in almost all guitar amplifiers and many guitar effects pedals. Such a circuit may be observed in Figure 41:

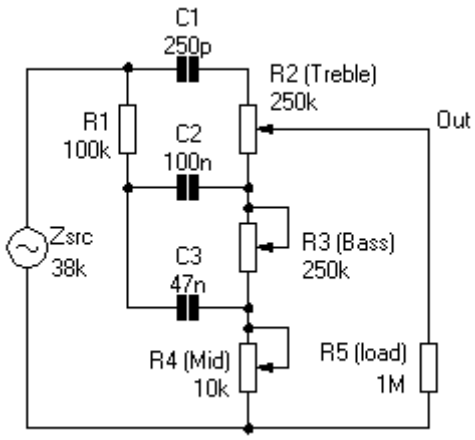



Figure 41: General Guitar Amplifier "Tone Stack"

This is commonly known as the "FMV" tone stack, which stands for Fender-Marshall-Vox, three of the most common guitar amplifier companies, who have all used variations on this configuration. Implementing a filter such as this would only involve one instance of pre-warping.

## 9. LU Decomposition

The LU decomposition algorithm decomposes a matrix into the product of two triangular matrices, one upper and one lower. The following is a 3x3 example.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Since

$$[A] = [L][U],$$

$$[A]^{-1} = [L]^{-1}[U]^{-1}$$

Using this principle, the inverse of [A] can be found column by column. Using the matrix inverse principle we know that

$$[A][A]^{-1} = [I]$$

So, using a temporary vector [z], we may solve for a column of $[A]^{-1}$ using [L] and [U]. We set this equal to the appropriate column of a properly-sized identity matrix and solve. For example, solving for the first column. Here, let $[A]^{-1}=[B]$

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

So, by first solving for [z] using the first equation and forward substitution, we may then solve for [B] using the second equation and back substitution. The result is the first column of $[A]^{-1}$. The second and third columns can subsequently be solved in the same manner.

LU decomposition makes several types of matrix math simpler and has the benefit of readily algorithmic. The function used is a well established one found in *Numerical Recipes In C*. It provides one function to decompose a matrix in place, and another to perform backsubstitution. Backsubstitution can be used to solve systems of linear equations or, as in this case, it can be used to calculate the inverse of a matrix. As noted earlier, the LU decomposition method of matrix inversion has the same operation count as the Jordan Gaussian method, unless the inverted matrix is subsequently multiplied by another matrix. In this case, an entire matrix multiplication is skipped. By backsubstituting the matrix to be multiplied by the inverted matrix, the operation may be combined into a single step.

## 9.1 Application of LU Decomposition to the Bilinear Transform

There is one inversion necessary for the computation of all discrete state space coefficients by the bilinear transform,

$$\left(\boldsymbol{I} - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}$$

So, prior to performing the inversion, the continuous time **A** matrix must be calculated based on the analog circuit. This **A** matrix must be multiplied by a scalar, (1/k), and then subtracted from a properly sized identity matrix. The result may then be inverted. It would be desirable to take advantage of the computational savings provided by this algorithm when a matrix multiplication is included in the solution. Reexamining the bilinear transform algorithm, it can be observed that there are five matrix multiplications involved. Two of these involve the multiplication of the inverted matrix by the **B** matrix. If the overall number of matrix multiplications can be reduced, this would be desirable, as matrix multiplication is a high computational complexity operation.

What must be determined is whether it is computationally more efficient to use the LU decomposition algorithm to perform the inversion and multiplication in one step, but multiple times depending on the multiplication to be made, or to perform the inversion just once and perform all multiplications with a standard matrix multiplication function.

In the meantime, there are several routines that need to be written that perform various functions necessary to perform the transform. These include: multiplication of a matrix by a scalar, multiplication of two matrices, addition of two matrices, subtraction of one matrix from another, creation of an identity matrix of a given order, and finally the matrix inversion. It is necessary to test all of these functions individually before stringing them all together into the ultimate function to calculate the bilinear transform.

With all the necessary subroutines constructed, it was possible to start testing. Debugging was performed one function at a time, using a parallel Matlab routine to verify the results. Following the order of the code, functions were tested in this order: matrix scalar multiplication, matrix subtraction, lu decomposition and inversion, matrix addition, and finally, matrix multiplication. After some careful debugging, the code was found to functional up for all routines.

The algorithm used is stated in the Bilinear Transform section, and is reprinted here for convenience:

$$\boldsymbol{A}_d = \left(I + \left(\frac{1}{k}\right)\boldsymbol{A}\right)\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}$$

$$\boldsymbol{B}_d = \frac{2k}{r}\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}\boldsymbol{B}$$

$$\boldsymbol{C_d} = r\boldsymbol{C}\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}$$

$$\boldsymbol{D}_d = \left(\frac{1}{k}\right)\boldsymbol{C}\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}\boldsymbol{B} + \boldsymbol{D}$$

Since it was known that all of the necessary functions were operating correctly, implementing the bilinear transform algorithm was not to be difficult. Upon coding the algorithm itself however, it was found that it returned values which were consistent with the Matlab results for the A and D matrices, but inconsistent for the B and C matrices. To investigate this, the variables were watched in the Code Composer Studio 'watch window' and the code was stepped through. It became apparent that the correct values for the B matrix were returned when the scalar multiplication of (2k/r) was omitted from the B section of the algorithm. Several matrices were checked in Matlab and in the code, and it

was found that they produced the same results without the scalar multiplication and inconsistent results when it was included.

When analyzing the discrepancy between the Matlab results for the C matrix and those of the code, it appeared that they were off by a scale factor, but otherwise consistent. Performing simple algebra, the scale factor was determined, and it was found to by 209.98. To determine the source of this scale factor, the Matlab code for computing the bilinear transform was examined. The Matlab code was carefully tested, operation by operation against the DSP code. It was determined that there is a typographical error in Matlab help document which states the algorithm. This accounts for the "scale factor". The actual code for computing the transform is as follows:

$$\boldsymbol{A}_d = \left(I + \left(\frac{1}{k}\right)\boldsymbol{A}\right)\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}$$

$$\boldsymbol{B}_d = \left(\frac{T}{r}\right)\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}\boldsymbol{B}$$

$$\boldsymbol{C_d} = r\boldsymbol{C}\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}$$

$$\boldsymbol{D}_d = \left(\frac{T}{2}\right)\boldsymbol{C}\left(I - \left(\frac{1}{k}\right)\boldsymbol{A}\right)^{-1}\boldsymbol{B} + \boldsymbol{D}$$

Where T is the sampling period and r = sqrt(T).

With this algorithm and the code written on the DSK, the system is now capable of converting continuous time state space model to discrete time using the bilinear transform.

It is desirable to know how long this code takes to run. In the development environment, Code Composer Studio, there is a profiling utility that allows the user to determine how many clock cycles it take to run a certain function or section of code. Using a second-order system, the code was profiled,

and it was found that the bilinear transform code took an average of 11075 cycles and a maximum of 12115 cycles to complete, without any compiler optimization. Given that the processor on the TMS320C6713 operates at 225MHz, this means that the routine takes at most 53.8µs to process a second order system. In other words, it can calculate the bilinear transform of a second order system at a rate of about 18.5kHz. This is excellent news, as the models really only need be updated at a rate of 1kHz or so. The next step will be to perform the same test on systems of various orders such that a trend can be determined. As the system order increases, not only the time to compute the bilinear transform but also the time to process and input and compute an output will increase. The upper limit of the system order is unknown, and maybe have to be determined qualitatively, as one of the major constraints is user perception of delay between input (adjusting a control) and output (filter changing).

# 10. Providing User Variability

One issue with the TMS320C6713 DSK is that aside from the stereo audio inputs and outputs and JTAG emulation port, communication is difficult. The board does provide expansion ports that allow access to memory, peripherals, and HPI (Host Port Interface), but without external hardware and extensive custom code, external control is rather daunting. Though the DSK does not natively support UART serial communication, it can be achieved by using the McBSP (multi-channel buffered serial port). Unfortunately, the McBSP is primarily use to communicate with the AIC23 analog to digital converter (ADC) and digital to analog converter (DAC). Texas Instruments provides documentation on how to implement external serial communication but even with this functionality, but even still, an external serial capable device would be necessary to provide control to the DSK.

Technical considerations aside, guitarists are most accustomed to rotary controls. In other words, they like knobs. The ideal control interface would be either potentiometers or rotary encoders directly interfaced with the DSK. With additional hardware, this is possible.

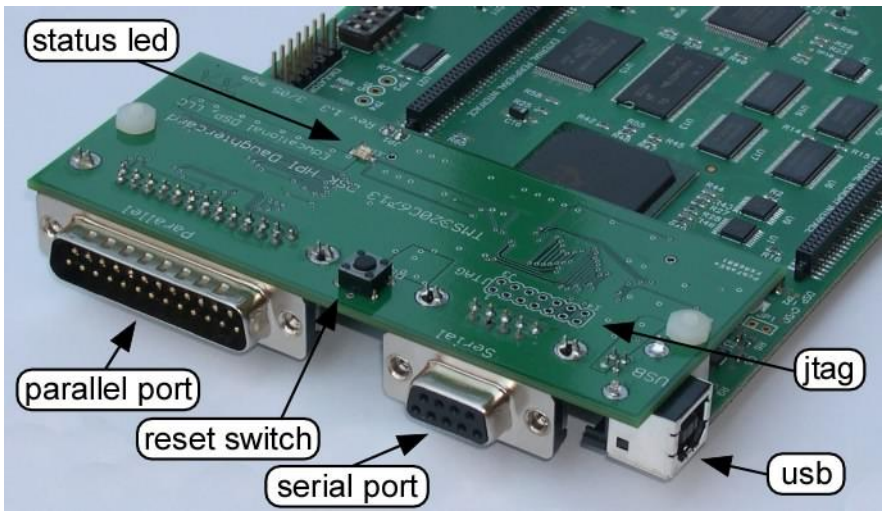



Figure 42: DSK6XXXHPI Daughterboard

Others have found the same inadequacies in the TMS320C6713 and developed a solution. The DSK6XXXHPI daughtercard for the TMS320C6713 can be seen in Figure 42. By interfacing with the Host Port Interface (HPI), the DSK6XXXHPI Daughtercard provides, USB, parallel, and serial (RS-232) communication ports. Furthermore, 8 of the inputs on the parallel port (pins 2 through 9) are capable of analog input with a dynamic range of 2.5V and 12-bit resolution (Morrow, Welch, & Wright). In additional to the added hardware functionality, code functionality is provided to read and write to the inputs, provide UART functionality, and allow direct access to the DSK from a computer via high speed USB communication and a PC interface GUI. Such additional functionality could be the beginning of a highly versatile, user modifiable musical effects platform. Physically interfacing with the daughter board is relatively simple. Figure 43 below shows the schematic representation of how an external control would be interfaced.

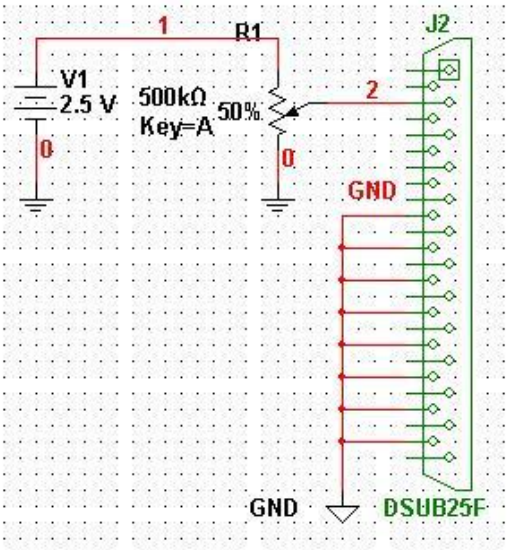



Figure 43: Schematic of Analog Control Input

With and external voltage connected across a potentiometer, the voltage across the wiper will be proportional to the resistance. Thus a voltage that can be varied between 0V and 2.5V can be supplied to the ADC of the daughter board, thus providing a use variable input in a familiar form factor.

The daughter board is provided with a number of C routines which allow for out of the box functionality. To achieve the DC coupled analog inputs, there are four functions which must be used. First, EnableAnalog() can pins D0-D7 of the daughterboard's parallel port as analog inputs. This function must be called before any of the others. Next, StartHpiServices() is a routine to initialize communication between the DSK and the daughterboard, and perform setup duties. Once this is run, the daughterboard can be used and accessed during runtime. Lastly, once the daughterboard has been initialized and configured, IsHpiDataNew() can be run to determine if any of the data on the daughterboard has changed. If the data has changed, that means that there is new input data to be fetched. ReadAnalog() fetches the value read by the ADC on the daughterboard.

# 11. Putting It All Together

With the ability to read DC analog inputs, compute the bilinear transform, and model discrete time implementations of continuous time filters, we now have all of the necessary components to implement a robust digital modeling platform. In the code, there are three main sections. In the main() function, the DSK must be configured and initialized before anything can be done. This entails the enabling of ADC interrupts, configuring the ADC to operate at the proper sampling frequency, enabling various inputs, declaring global variables, and defining common terms. Once initialization and configuration have been completed, the code enters the infinite while(1) loop in the main() function. The block diagram of this can be seen on the left hand side of Figure 44.

Within the while(1) loop, the code checks to see if there are new values at the input of the daughter card. If there are, they are read. Once they have been read, the values of the continuous time system can be recomputed. Once the continuous time system has been updated, it can be discretized

using the bilinear transform function. Once this routine has completed, a flag is set to notify the interrupt service routine that there are new state space values that should be used to compute the next output. A diagram of this can be seen in the center of Figure 44.

During the while(1) loop, interrupts from the AIC23 will occur. The ISR checks to see if there are new state space values. If there are not, it reads the new sample and proceeds to calculate the output using the established values. If the flag indicates that new values are indeed ready, it chooses the memory containing the new values, resets the flag, reads the new input sample, and computes the output using the new values. The state space values are set up in a "ping-pong" style memory management scheme. Using an array of two pointers, we can point to either one collection of values or another merely by changing a single designation, which is swapped when the ISR sees that new values are ready. Thus, one array is always being used and the other is either being updated or waiting to be updated. The diagram of the ISR can be found on the right hand side of Figure 44.

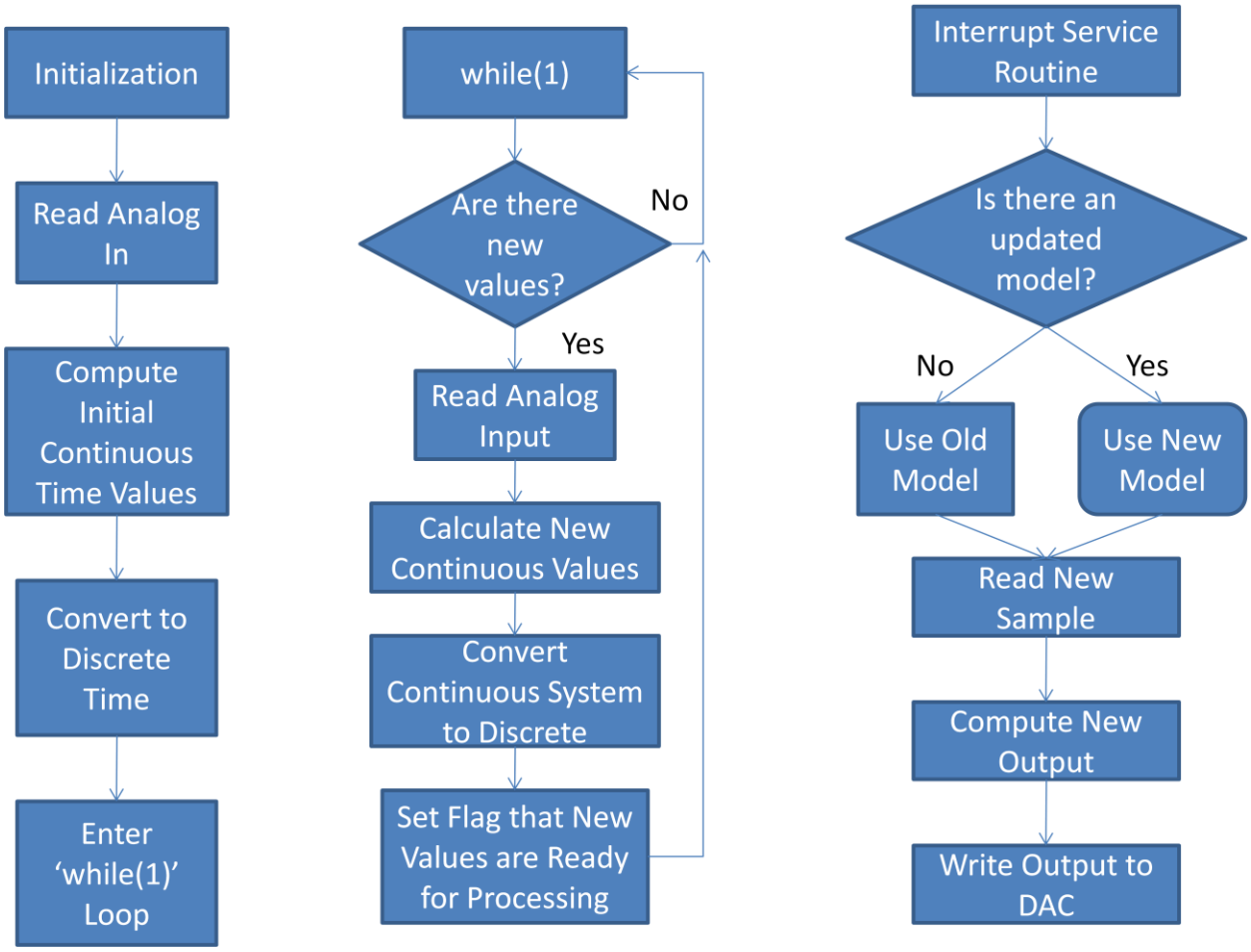



Figure 44: Code Flow Diagrams: Main Fuction, Infinite 'while' loop, ISR

# 12. Final Results

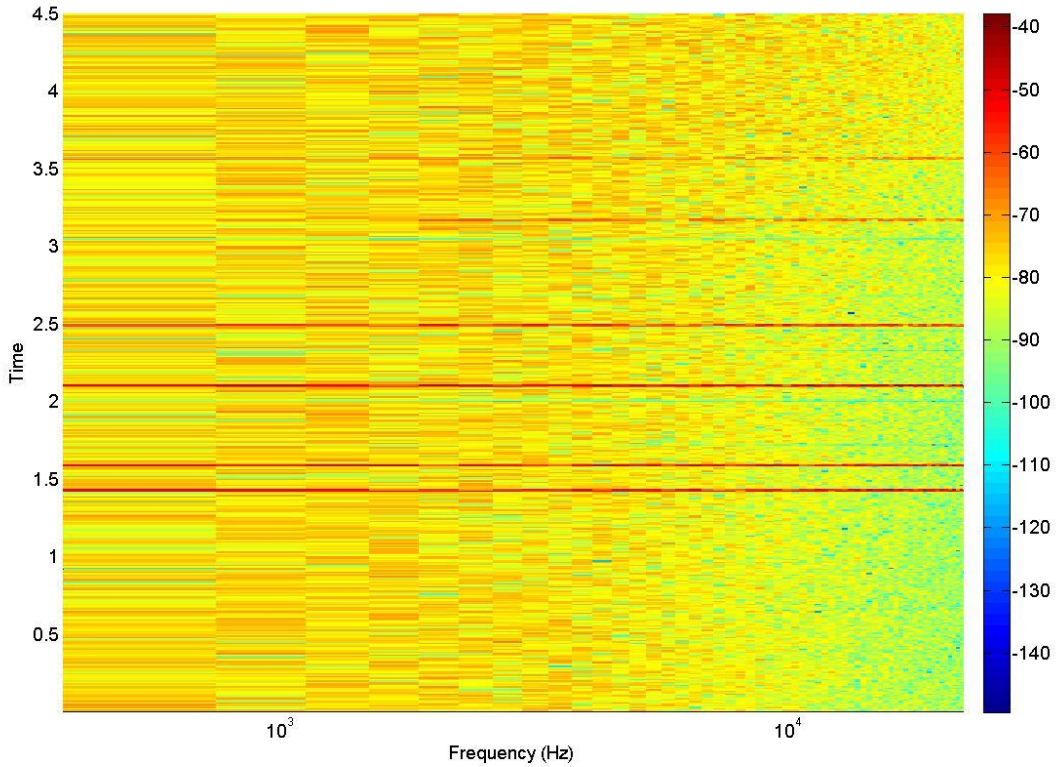



Figure 45: Spectrogram of Variable Sallen-Key Low Pass Model

Figure 45 shows a spectrogram of white noise as processed by the variable second order low pass model. The adjustment knob was turned from one extreme to the other over the course of the recording. A spectrogram shows frequency intensity over time. The color bar on the right is provided for reference of intensity. As can be observed, the intensity of frequencies above approximately 10kHz increases over the course of the recording. As can also be observed, there are several short bands of intense red. These correspond to the clicks discussed earlier as the frequency response is changed.

Despite the artifacts created by the variability, the system does function as desired and it is felt that these artifacts can be removed, though it is beyond the scope of this project to do so.

# 13. Conclusions

In the final iteration of this project, what was delivered is a DSP platform capable of accurately modeling variable analog filter circuits. Though there are undoubtedly other approaches to digital modeling, the intention of this project was to determine whether a circuit theory-informed approach could accurately and reliably model analog circuitry. Though there are some difficulties in this approach, it can be said with confidence that the general concept has been proven.

Ultimately, what the platform can do is to take a continuous state space model of an analog circuit, convert it to a discrete model, and digitally emulate the analog circuit in real time. Circuits such as variable filters are possible as well, due to the ability to recalculate the continuous-to-discrete conversion on demand. Though this is a complex computation, it can be performed in a small enough amount of time such that adjustment is perceived as smooth. Though the system is currently set up to handle single input single output systems, it readily adapts to multiple input multiple output systems as well.

Thus, the three major goals: accuracy, variability, and scalability have been met. In audio testing, the final digital models were imperceptibly different from the analog filters they aimed to replicate. Further testing should be performed to confirm this. Though there are minor differences between the magnitudes responses of the analog and digital filters, the difference is quite small and cannot be heard. The only limitation to circuit complexity is processing power. Though the system has not been tested beyond second order filters, it is built to handle N order systems. Thus, the code could be ported to a more powerful system if ever the hardware became a limitation. Through the use of a daughterboard with a DC-coupled ADC, the addition of analog controls was possible, allowing real-time user adjustment of the filter(s). Though the adjustment itself introduced new issues, it is felt that these are readily overcome.

# 14. Further Considerations and Future Improvements

The first improvement to be made would be to remove the “zipper” sounds that occur when the filter models are being modified. This may consist of an extra routine in the ISR which crossfades between the output as calculated by the previous model and the output as calculate by the new model. A method such as this might be sufficient to smooth the transition as the frequency response of the filter is suddenly changed.

This project is considered to be only a piece of a larger system. The ideal end product would consist of:

1. A graphical user interface (GUI) that would allow users to construct schematic such as in schematic capture programs
2. The ability for the program to algorithmically develop state space models based on a schematic and user defined input(s), output(s), and controls
3. The ability to generate C or assembly code for the DSP platform based upon that state space model
4. The ability to load said code onto a standalone DSP platform to be used untethered.
5. The ability to store and load various models from system memory
6. The ability to vary models on the fly based upon user defined controls

Using such a system, ideally any analog circuit could be easily and accurately modeled on this scalable and flexible platform.

## Works Cited


Jacko, J. A., & Sears, A. (2003). *The human-computer interaction handbook.* Mahwah, New Jersey, USA: Lawrence Earlbaum Associates, Inc., Publishers.

Karkaradov, B. (2004, Spring). *Ultra-Fast Matrix Multiplication: An Empirical Analysis of Highly Optimized Vector Algorithms.* Retrieved March 27, 2010, from Stanford University: http://cs.stanford.edu/people/boyko/pubs/MatrixMult_SURJ_2004.pdf

Lathi, B. (2005). *Linear Systems and Signals.* Oxford: Oxford University Press.

Lay, D. (2003). *Linear Algebra and Its Applications.* Boston: Addison Wesley.

Line 6. (n.d.). *Line 6 - M13 Stompbox Modeler - Guitar Multi Effects Pedal*. Retrieved April 18, 2010, from Line 6: http://line6.com/m13/sounds.html

Morrow, M. G., Welch, T. B., & Wright, C. H. (n.d.). *A Host Port Interface Board to Enhance the TMS320C6713 DSK.* Retrieved March 28, 2010, from IEEE: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01660506

National Digital Information Infrastructure & Preservation Program. (2008, February 19). *Sustainability of Digital Formats Planning for Library of Congress*. Retrieved March 21, 2010, from Linear Pulse Code Modulated Audio (LPCM): http://www.digitalpreservation.gov/formats/fdd/fdd000011.shtml

Olshausen, B. A. (2000, October 10). *Aliasing.* Retrieved March 21, 2010, from UC Berkeley: Redwood Center for Theoretical Neuroscience: https://redwood.berkeley.edu/bruno/npb261/aliasing.pdf

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1999). *Numerical Recipes in C.* Cambridge, UK: Cambridge University Press.

Robinson, S. (2005). Toward an Optimal Algorithm for Matrix Multiplication. *SIAM News , 38* (9).

Sankar, K. (2007, December 2). *Digital implementation of RC low pass filter*. Retrieved November 24, 2009, from dspLog: http://www.dsplog.com/2007/12/02/digital-implementation-of-rc-low-pass-filter/

Slump, C., C.G.M., v. A., Barels, J., Brunink, W., Drenth, F., Pol, J., et al. (n.d.). *Design and Implementation of a Linear-Phase Equalizer in Digital Audio Signal Processing.* Retrieved March 28, 2010, from University of Twente: http://doc.utwente.nl/16234/1/Design_Implementation_-_slump.pdf

Spectrum Digital Inc. (2010). *Spectrum Digital Inc. DSP Starter Kit (DSK) for the TMS320C6713*. Retrieved April 22, 2010, from Spectrum Digital Inc.: http://www.spectrumdigital.com/product_info.php?cPath=31_75&products_id=113&osCsid=8ec5ff0ea8dd123d8163a4bbc1506c4c

Texas Instruments. (2002, September). *Analysis of the Sallen-Key Architecture.* Retrieved January 15, 2010, from Texas Instruments: http://focus.ti.com/lit/an/sloa024b/sloa024b.pdf

The Mathworks. (2010). *Bilinear transformation method for analog-to-digital filter conversion*. Retrieved March 7, 2010, from The Mathworks: http://www.mathworks.com/access/helpdesk/help/toolbox/signal/bilinear.html

The MathWorks. (2010). *Convert from continuous to discrete-time models.* Retrieved February 16, 2010, from The MathWorks: http://www.mathworks.com/access/helpdesk/help/toolbox/control/ref/c2d.html

The MathWorks. (2010). *Converting Between Continuous- and Discrete-Time Representations*. Retrieved March 21, 2010, from The MathWorks: http://www.mathworks.com/access/helpdesk/help/toolbox/control/manipmod/f2-3161.html

Williams, C. S. (1986). *Designing Digital Filters.* Englewood Cliffs, New Jersey: Prentice-Hall.

# Appendix A: Variable Low Pass Filter (Final) C Code

```
#define CHIP_6713
#define TINY 1.0e-20;
#define N 2
#define IN 1
#define OUT 1
#define NR_END 1
#define FREE_ARG char*
#include <stdio.h>
#include <stdlib.h>
#include <c6x.h>
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include "dsk6713.h"
#include "dsk6713_aic23.h"
#include "nr.h"
#include "nrutil.h"
#include <math.h>
#include "winDSK6_def.h"
#include "hpi_services.h"
#include "hpi_services_funcs.h"
#define NUMBER_OF_HPI_INIT_RETRIES    100000  // retries on initialization

HPI_SVC_BLOCK HpiSvcBlock = {
       0, // flag
       HPI_SVC_BAUD115200 // default baud rate
};

DSK6713_AIC23_CodecHandle hCodec;
DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
// Codec configuration with default settings

/*****************FUNCTION PROTOTYPES*****************/
interrupt void serialPortRcvISR(void);
void hook_int();
void identity_builder(void);
void matrix_scalar_mult(float **a, int rows, int cols, float scalar);
void matrix_mult(float **out, float **m1, float **m2, int r1, int c1, int r2, int c2);
void matrix_add(float **a, float **b, int rows, int cols);
void matrix_sub(float **a, float **b, int rows, int cols);
void ludcmp(float **a, int n, int *indx, float *d);
void lubksb(float **a, int n, int *indx, float b[]);

const float R1 = 10000;
float R2 = 10000;
const float Q1 = 1.0e-9;
const float Q2 = 1.0e-9;
const float fs = 96000;
float t;

const int order = N; // system order, determines matrix sizes
int ins = IN;
int outs = OUT;

float **identity;
float **inverted;

float **Ac;
float **Bc;
float **Cc;
float **Dc;

float **Ad;
float **Bd;
float **Cd;
float **Dd;

float A[N+1][N+1];
float B[N+1][IN+1];
```

```
float C[IN+1][N+1];
float D[N+1][N+1];

float x[N+1] = {0.0,0.0,0.0};
float x_next[N+1] = {0.0,0.0,0.0};

int flag = 0;
int ludflag = 0;

void main()
{
        float **Atemp, **Apart, tmp1, tmp2;
        int i,j,*indx;
        float d,r,*col;
        unsigned short analog_in[2];

        DSK6713_init(); // Initialize the board support library
        hCodec = DSK6713_AIC23_openCodec(0, &config);
        MCBSP_FSETS(SPCR1, RINTM, FRM);
        MCBSP_FSETS(SPCR1, XINTM, FRM);
        MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
        MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
        DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_96KHZ);

        // daughterboard setup
        EnableAnalog(0xFF);
        StartHpiServices();

        hook_int();

        indx=ivector(1,N);
        col=vector(1,N);
        Ac = matrix(1,N,1,N);
        Bc = matrix(1,N,1,IN);
        Cc = matrix(1,IN,1,N);
        Dc = matrix(1,IN,1,IN);
        Ad = matrix(1,N,1,N);
        Bd = matrix(1,N,1,IN);
        Cd = matrix(1,IN,1,N);
        Dd = matrix(1,N,1,N);
        identity = matrix(1,N,1,N);
        inverted = matrix(1,N,1,N);
        Apart = matrix(1,N,1,N);
        Atemp = matrix(1,N,1,N);

        t = 1/fs;

        Ac[1][1] = -2.0e5;
        Ac[1][2] = 1.0e5;
        Ac[2][1] = 1.0e5;
        Ac[2][2] = -1.0e5;

        Bc[1][1] = 1.0e5;
        Bc[2][1] = 0.0;

        Cc[1][1] = 0.0;
        Cc[1][2] = 1;

        Dc[1][1] = 0.0;

        A[1][1] = -0.2496;
        A[1][2] = 0.3987;
        A[2][1] = 0.3987;
        A[2][2] = 0.1492;

        B[1][1] = 178.7;
        B[2][1] = 94.94;

        C[1][1] = 0.0009494;
        C[1][2] = 0.002736;
```

```
D[1][1] = 0.226;


r = sqrt(t);

tmp1 = 1/(R1*Q1);

identity_builder();    // Create an identity matrix with same dimensions as Ac

while(1)
{
        if(IsHpiDataNew()){
                analog_in[0] = ReadAnalog(0);

                // set noise threshold.  If analog value has not changed significantly,
                // do not recompute bilinear transform
                if((analog_in[0]-analog_in[1] >= 7)||(analog_in[1]-analog_in[0] >= 7))
                {
                        // Compute value of R2 based on analog reading
                        R2=((float)analog_in[0])/((float)HPI_SVC_ANALOG_MAX_VALUE)*50000;

                        if(R2<1000)
                                R2 = 1000;

                        tmp2 = 1/(Q1*R2);

                // compute new continuous A matrix values (B,C, and D do not change)
                        Ac[1][1] = -(tmp1 + tmp2);
                        Ac[1][2] = tmp2;
                        Ac[2][1] = tmp2;
                        Ac[2][2] = -tmp2;

// move continuous values into temp matrix.  They will be destroyed in the process
                        for(i=1;i<=order;i++){
                        for(j=1;j<=order;j++){
                                Atemp[i][j] = Ac[i][j];
                                }
                        }

                        // Compute (I-(1/k)A)^-1
                        // Perform (1/k)*Ac
                        matrix_scalar_mult(Atemp, N, N,(t/2));
                        // Perform I-(1/k)*Ac  Now ready for inversion!
                        matrix_sub(identity,Atemp,order,order);
                        ludcmp(Atemp,N,indx,&d);

                        // Was there an error in ludcmp?  If not, proceed with transform
                        if (ludflag != 1){
                                for(j=1;j<=N;j++){
                                        for(i=1;i<=N;i++) col[i] = 0.0;
                                        col[j] = 1.0;
                                        lubksb(Atemp,N,indx,col);
                                        for(i=1;i<=N;i++) inverted[i][j] = col[i];
                                }


                                // Compute Ad
                                for(i=1;i<=N;i++){
                                for(j=1;j<=N;j++){
                                        Apart[i][j] = Ac[i][j];
                                        }
                                }

                                // Perform (1/k)*Ac
                                matrix_scalar_mult(Apart, N, N, (t/2));
                                // Perform I+(1/k)*Ac
                                matrix_add(Apart, identity, order, order);
                                // Calculate Ad
                                matrix_mult(Ad,Apart,inverted,N,N,N,N);

                                // Compute Bd
```

```
                                        matrix_mult(Bd,inverted,Bc,N,IN,N,N);
                                        matrix_scalar_mult(Bd, N, IN, (t/r));

                                        // Compute Cd & Dd
                                        matrix_mult(Cd,Cc,inverted,IN,N,N,N);
                                        matrix_mult(Dd,Cd,Bc,IN,N,N,IN);
                                        matrix_scalar_mult(Cd,IN,N,r);
                                        matrix_scalar_mult(Dd,IN,IN,(t/2));
                                        matrix_add(Dd,Dc,IN,IN);

                                // Move analog value in buffer for comparison with next value
                                        analog_in[1] = analog_in[0];

                                        // Set flag that new values are ready
                                        flag = 1;
                                }
                        // if there was an error in ludcmp do not compute transform, reset ludflag
                                else
                                        ludflag = 0;
                        }
                }
        }

}

void nrerror(char error_text[])
// Numerical Recipes Standard Error Handler
// Source: Numerical Recipes in C (Press, et al)
{
        fprintf(stderr,"Numerical Recipies run-time error...\n");
        fprintf(stderr,"%s\n",error_text);
        fprintf(stderr,"...now exiting to system...\n");
        exit(1);
}

float **matrix(long nrl, long nrh, long ncl, long nch)
// Source: Numerical Recipes in C (Press, et al)
// Allocate a float matrix with subscript range m[nrl-nrh][ncl-nch]
{
        long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
        float **m;

        // allocate pointers to rows
        m=(float **) malloc((size_t)((nrow+NR_END)*sizeof(float*)));
        if (!m)
                nrerror("allocation failure 1 in matrix()");
        m += NR_END;
        m -= nrl;

        // allocate rows and set pointers to them
        m[nrl]=(float *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(float)));
        if (!m[nrl])
                nrerror("allocation failure 2 in matrix()");
        m[nrl] += NR_END;
        m[nrl] -= ncl;

        for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

        // return pointer to array of pointers to rows
        return m;

}

int *ivector(long nl, long nh)
// Source: Numerical Recipes in C (Press, et al)
// allocate an int vector with subscript range v[nl..nh]
{
        int *v;

        v=(int *)malloc((size_t)((nh-nl+1+NR_END)*sizeof(int)));
        if (!v)
```

```
                nrerror("allocation failure in ivector()");
        return v-nl+NR_END;
}

void identity_builder(void)
// Creates an identity matrix (square) of a given order
{
        int n,k;

        for(n=1;n<=order;n++){
                for(k=1;k<=order;k++){
                        if(n==k)
                                identity[n][k] = 1;
                        else
                                identity[n][k] = 0;
                }
        }
}

void matrix_scalar_mult(float **a, int rows, int cols, float scalar)
// Multiplies a matrix by a scalar value
{
        int n,k;

        for(n=1;n<=rows;n++){
                for(k=1;k<=cols;k++){
                        a[n][k] *= scalar;
                }
        }
}

void matrix_mult(float **out, float **m1, float **m2, int r1, int c1, int r2, int c2)
// Multiply two matrices together
{
        int i,j,k;      //for loop placeholders

        for(i=1;i<=r1;i++){
                for(j=1;j<=c2;j++){
                        out[i][j]=0;
                        for(k=1;k<=r2;k++){
                                out[i][j]+=m1[i][k]*m2[k][j];
                        }
                }
        }
}


void matrix_add(float **a, float **b, int rows, int cols)
// Add two matrices together
{
//  NOTE: MATRICES MUST BE OF IDENTICAL DIMENSIONS
//      The matrices A and B will be added together and the result returned in A

        int i,j;

        for(i=1;i<=rows;i++){
                for(j=1;j<=cols;j++){
                        a[i][j] += b[i][j];
                }
        }

}




void matrix_sub(float **a, float **b, int rows, int cols)
// Subtract one matrix from another
{
//  NOTE: MATRICES MUST BE OF IDENTICAL DIMENSIONS
//      The matrix A will be subtracted from B and the result returned in B
```

```
        int i,j;

        for(i=1;i<=rows;i++){
                for(j=1;j<=cols;j++){
                        b[i][j] = a[i][j] - b[i][j];
                }
        }

}


void ludcmp(float **a, int n, int *indx, float *d)
// Numerical Recipies lu decomposition routine
// destroys 'a', replaces with decomposed matrix
{
        int i,imax,j,k;
        float big,dum,sum,temp;
        float *vv;                                      //vv stores the implicit scaling of each row

        vv = vector(1,n);
        *d = 1.0;
        for(i=1;i<=n;i++){
                big = 0.0;
                for(j=1;j<=n;j++)
                        if((temp=fabs(a[i][j])) > big) big = temp;
                if(big == 0.0) {
                        ludflag = 1;
                        return;
                }
                vv[i] = 1.0/big;              // save scaling
        }
        for(j=1;j<=n;j++){
                for(i=1;i<j;i++){
                        sum = a[i][j];
                        for(k=1;k<i;k++) sum -= a[i][k]*a[k][j];
                        a[i][j] = sum;
                }
                big = 0.0;
                for(i=j;i<=n;i++){
                        sum = a[i][j];
                        for(k=1;k<j;k++)
                                sum -= a[i][k]*a[k][j];
                        a[i][j] = sum;
                        if((dum=vv[i]*fabs(sum)) >= big) {
                                big = dum;
                                imax = i;
                        }
                }
                if(j != imax){
                        for(k=1;k<=n;k++){
                                dum = a[imax][k];
                                a[imax][k] = a[j][k];
                                a[j][k] = dum;
                        }
                        *d = -(*d);
                        vv[imax] = vv[j];
                }
                indx[j] = imax;
                if(a[j][j] == 0.0)
                        a[j][j] = TINY;
                if(j != n){
                        dum = 1.0/(a[j][j]);
                        for(i=j+1;i<=n;i++) a[i][j] *= dum;
                }
        }
                free_vector(vv,1,n);
}

void lubksb(float **a, int n, int *indx, float b[])
// Source: Numerical Recipes in C (Press, et al)
```

```
// Perform backsubstitution to solve Ax=B
// We use this with an identity vector to solve the inverse
{
	int i,ii=0,ip,j;
	float sum;

	for(i=1;i<=n;i++){
		ip = indx[i];
		sum = b[ip];
		b[ip] = b[i];
		if(ii)
			for(j=ii;j<=i-1;j++) sum -= a[i][j] * b[j];
		else if (sum) ii = i;
		b[i] = sum;
	}
	for(i=n;i>=1;i--){
		sum = b[i];
		for(j=i+1;j<=n;j++) sum -= a[i][j] * b[j];
		b[i] = sum/(a[i][i]);
	}
}

float *vector(long nl, long nh)
// Source: Numerical Recipes in C (Press, et al)
// allocate a float vector with subscript range v[nl..nh]
{
	float *v;

	v = (float *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(float)));
	if (!v)
		v = (float *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(float)));
	return v-nl+NR_END;
}

void free_vector(float *v, long nl, long nh)
// Source: Numerical Recipes in C (Press, et al)
// free a float vector allocated with vector()
{
	free((FREE_ARG) (v+nl-NR_END));

}


void hook_int()
// Source: Professor Donald Richard Brown III
// set up global interrupts
{
	IRQ_globalDisable();		// Globally disables interrupts
	IRQ_nmiEnable();		// Enables the NMI interrupt
	IRQ_map(IRQ_EVT_RINT1,15);	// Maps an event to a physical interrupt
	IRQ_enable(IRQ_EVT_RINT1);	// Enables the event
	IRQ_globalEnable();		// Globally enables interrupts
}

int StartHpiServices()
// Source: EducationalDSP
// Set up daughtercard services via HPI
{
	unsigned int i;

//	HpiSvcBlock.ddir = 0xFFFF;			// all digital pins inputs
//	HpiSvcBlock.aen = 0;				// all analog disabled
	HpiSvcBlock.com_tx_head = 0;			// show all buffers empty
	HpiSvcBlock.com_rx_head = 0;
	HpiSvcBlock.usb_tx_head = 0;
	HpiSvcBlock.usb_rx_head = 0;
	HpiSvcBlock.com_tx_tail = 0;
	HpiSvcBlock.com_rx_tail = 0;
	HpiSvcBlock.usb_tx_tail = 0;
	HpiSvcBlock.usb_rx_tail = 0;
```

```
	*(unsigned int *)HPI_SVC_XFER_ADDRESS = (unsigned int)&HpiSvcBlock;	// store data
structure address
	HpiSvcBlock.flag = DSP_TO_HOST_MAGIC_NUMBER;	// store magic number

	// signal HPI daughtercard to start services by writing 1 to HINT bit in HPIC
	*(unsigned int *)0x01880000 = 0x00000004;

	for(i = 0;i < NUMBER_OF_HPI_INIT_RETRIES;i++)		// wait for daughtercard to
reply
		if(HpiSvcBlock.flag == HOST_TO_DSP_MAGIC_NUMBER)
			return 1;

	return 0;
}

int IsHpiDataNew()
// Source: EducationalDSP
// Check if there is new data to be read from the daughtercard
{
	static unsigned short last_flag = 0;

	if(last_flag != HpiSvcBlock.flag) {
		last_flag = HpiSvcBlock.flag;
		return 1;
	}
	return 0;
}

void EnableAnalog(unsigned short number)
// Source: EducationalDSP
// enable analog inputs on the parallel port
{
	HpiSvcBlock.aen = number;
}

unsigned short ReadAnalog(unsigned short pin)
// Source: EducationalDSP
// read specified analog pin
{
	switch(pin) {
	case 0:
		return HpiSvcBlock.an0;
	case 1:
		return HpiSvcBlock.an1;
	case 2:
		return HpiSvcBlock.an2;
	case 3:
		return HpiSvcBlock.an3;
	case 4:
		return HpiSvcBlock.an4;
	case 5:
		return HpiSvcBlock.an5;
	case 6:
		return HpiSvcBlock.an6;
	case 7:
		return HpiSvcBlock.an7;
	}
	return 0; // out of bounds
}

interrupt void serialPortRcvISR()
// Interrupt service routine
// read new ADC inputs, write new DAC outputs
{
	int n=0;
	int k=0;
	short u = 0;
	float output;
	union {Uint32 combo; short channel[2];} temp;

	if(flag==1){
```

```
		for(n=1;n<=N;n++){
			for(k=1;k<=N;k++){
				A[n][k] = Ad[n][k];
				B[n][1] = Bd[n][1];
				C[1][k] = Cd[1][k];
				x[n] = 0.0;
			}
		}
		D[1][1] = Dd[1][1];

	flag = 0;

	}

	temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
	u = temp.channel[0];			//write new sample to input var

	output = 0.0;

	for(n=1;n<=N;n++){			//loop to compute output, mac'ing C matrix and x's
		output+=C[1][n] * x[n];
	}

	output += (float)u * D[1][1];

	temp.channel[0] = (short)output;	//cast output as short, write to MCBSP buffer

	MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo); //write output

	for(k=1;k<=N;k++){				// choose which matrix row
		x_next[k] = B[k][1] * (float)u;		// do B matrix multiplication
		for(n=1;n<=N;n++){			// choose which matrix column
			x_next[k] += A[k][n] * x[n];	// do A matrix multiplication
		}
	}

	for(n=1;n<=N;n++){				//move in new x values
		x[n] = x_next[n];
	}
}
```

# Appendix B: RC Filter C Code

```
#define CHIP_6713
#define N 1
#include <stdio.h>
#include <c6x.h>
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include "dsk6713.h"
#include "dsk6713_aic23.h"

DSK6713_AIC23_CodecHandle hCodec;
DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;

// Codec configuration with default settings
interrupt void serialPortRcvISR(void);
void hook_int();

float u = 0.0;
float output = 0.0;

float x = 0.0;
float x_next = 0.0;
float A;
float B;
float out;                                      // SS C Matrix

//const float R = 10500.0;
//const float C = 0.0000000048;
//const float Fs = 44100.0;
//double k = 0.0;

void main()
{
        DSK6713_init(); // Initialize the board support library
        hCodec = DSK6713_AIC23_openCodec(0, &config);
        MCBSP_FSETS(SPCR1, RINTM, FRM);
        MCBSP_FSETS(SPCR1, XINTM, FRM);
        MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
        MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
        DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_44KHZ);
        hook_int();

        A = 6.173;

        B = 1.000;

        out = 3.827;

                while(1)
                        {
                        }
}

void hook_int()
// Source: Professor Donald Richard Brown III
{
        IRQ_globalDisable();          // Globally disables interrupts
        IRQ_nmiEnable();                      // Enables the NMI interrupt
        IRQ_map(IRQ_EVT_RINT1,15);    // Maps an event to a physical interrupt
        IRQ_enable(IRQ_EVT_RINT1);    // Enables the event
        IRQ_globalEnable();           // Globally enables interrupts
}

interrupt void serialPortRcvISR()
{
        union {Uint32 combo; short channel[2];} temp;
        temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
        u = temp.channel[0];                  //write new sample to input var
```

```
        x_next = (A * x) + (B * u);                          // do B matrix multiplication
        output = out * x_next;
        temp.channel[0] = (short)output;      //cast output as short, write to MCBSP buffer
        MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo); //write output

        x = x_next;
        output = 0.0;

}
```

# Appendix C: State Space Sallen-Key Filter Code

```
#define CHIP_6713
#define N 2
#include <stdio.h>
#include <c6x.h>
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include "dsk6713.h"
#include "dsk6713_aic23.h"

DSK6713_AIC23_CodecHandle hCodec;
DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;

// Codec configuration with default settings
interrupt void serialPortRcvISR(void);
void hook_int();

float u = 0.0;
float output = 0.0;

float x[N] = {0.0, 0.0};
float x_next[N] = {0.0, 0.0};
float A[N][N];
float B[N];
float out[N];
float D;


void main()
{
        DSK6713_init(); // Initialize the board support library
        hCodec = DSK6713_AIC23_openCodec(0, &config);
        MCBSP_FSETS(SPCR1, RINTM, FRM);
        MCBSP_FSETS(SPCR1, XINTM, FRM);
        MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
        MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
        DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_44KHZ);
        hook_int();

        /* BILINEAR TRANSFORM COEFFS
        A[0][0] = 0.06015;
        A[0][1] = -16110;
        A[1][0] = .00001202;
        A[1][1] = 0.8173;

        B[0] = 0.5301;
        B[1] = 0.00000601;

        out[0] = .5015;
        out[1] = -7620;

        D = 0.2507;*/

        /* ZERO ORDER HOLD COEFFS */
        A[0][0] = 0.1071;
        A[0][1] = -14440;
        A[1][0] = .00001077;
        A[1][1] = 0.7857;

        B[0] = 0.00001077;
        B[1] = 0.0000000001599;

        out[0] = 41720;
        out[1] = 0.0;

        D = 0.0;

                while(1)
                        {
```

```
                        }
}

void hook_int()
// Source: Professor Donald Richard Brown III
{
        IRQ_globalDisable();          // Globally disables interrupts
        IRQ_nmiEnable();              // Enables the NMI interrupt
        IRQ_map(IRQ_EVT_RINT1,15);    // Maps an event to a physical interrupt
        IRQ_enable(IRQ_EVT_RINT1);    // Enables the event
        IRQ_globalEnable();           // Globally enables interrupts
}


interrupt void serialPortRcvISR()
{
        int n=0;
        int k=0;

        union {Uint32 combo; short channel[2];} temp;
        temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
        u = temp.channel[1];                  //write new sample to input var

        output = 0.0;

        for(n=0;n<N;n++){                     //move in new x values
                x[n] = x_next[n];
        }

        for(n=0;n<N;n++){                     //loop to compute output, mac'ing C matrix and x's
                output+=out[n] * x[n];
        }

        output += u * D;

        temp.channel[1] = (short)output;      //cast output as short, write to MCBSP buffer

        MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo); //write output

        for(k=0;k<N;k++){                                          // choose which matrix row
                x_next[k] = B[k] * u;                           // do B matrix multiplication
                for(n=0;n<N;n++){                                  // choose which matrix column
                        x_next[k] += A[k][n] * x[n];            // do A matrix multiplication
                }
        }
}
```

# Appendix D: Bilinear Transform Code

```
#define CHIP_6713
#define TINY 1.0e-20;
#define N 2
#define IN 1
#define OUT 1
#define NR_END 1
#define FREE_ARG char*
#include <stdio.h>
#include <stdlib.h>
#include <c6x.h>
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include "dsk6713.h"
#include "dsk6713_aic23.h"
#include "nr.h"
#include "nrutil.h"
#include <math.h>

DSK6713_AIC23_CodecHandle hCodec;
DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
// Codec configuration with default settings

/*****************FUNCTION PROTOTYPES*****************/
interrupt void serialPortRcvISR(void);
void hook_int();
void identity_builder(void);
void matrix_scalar_mult(float **a, int rows, int cols, float scalar);
void matrix_mult(float **out, float **m1, float **m2, int r1, int c1, int r2, int c2);
void matrix_add(float **a, float **b, int rows, int cols);
void matrix_sub(float **a, float **b, int rows, int cols);
void ludcmp(float **a, int n, int *indx, float *d);
void lubksb(float **a, int n, int *indx, float b[]);

const float R1 = 10000;
const float R2 = 10000;
const float Q1 = 1e-9;
const float Q2 = 1e-9;
const float fs = 44100.0;

const int order = N; // system order, determines matrix sizes (# of energy storage devices)
int ins = IN;
int outs = OUT;

float **identity;
float **inverted;

float **Ac;
float **Bc;
float **Cc;
float **Dc;

float **Ad;
float **Bd;
float **Cd;
float **Dd;

float x[N] = {0.0,0.0};
float x_next[N] = {0.0,0.0};

void main()
{
        float **Atemp, **Apart;
        int i,j,*indx;
        float d,k,r,*col;

        DSK6713_init(); // Initialize the board support library
        hCodec = DSK6713_AIC23_openCodec(0, &config);
        MCBSP_FSETS(SPCR1, RINTM, FRM);
```

```
        MCBSP_FSETS(SPCR1, XINTM, FRM);
        MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
        MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
        DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_44KHZ);
        hook_int();

        indx=ivector(1,N);
        col=vector(1,N);
        Ac = matrix(1,N,1,N);
        Bc = matrix(1,N,1,IN);
        Cc = matrix(1,IN,1,N);
        Dc = matrix(1,IN,1,IN);
        Ad = matrix(1,N,1,N);
        Bd = matrix(1,N,1,IN);
        Cd = matrix(1,IN,1,N);
        Dd = matrix(1,N,1,N);
        identity = matrix(1,N,1,N);
        inverted = matrix(1,N,1,N);
        Apart = matrix(1,N,1,N);
        Atemp = matrix(1,N,1,N);

        Ac[1][1] = -63000;
        Ac[1][2] = -1340000000;
        Ac[2][1] = 1.0;
        Ac[2][2] = 0.0;

        Bc[1][1] = 1.0;
        Bc[2][1] = 0.0;

        Cc[1][1] = 41720;
        Cc[1][2] = 0.0;

        Dc[1][1] = 0.0;

        k = 2*fs;
        r = sqrt(2/k);

        identity_builder();           // Create an identity matrix with same dimensions as Ac

        for(i=1;i<=order;i++){
        for(j=1;j<=order;j++){
                Atemp[i][j] = Ac[i][j];
                }
        }

        matrix_scalar_mult(Atemp, N, N,(1/k));                      // Perform (1/k)*Ac

        matrix_sub(identity,Atemp,order,order);// Perform I-(1/k)*Ac  Now ready for inversion!

        ludcmp(Atemp,N,indx,&d);
        for(j=1;j<=N;j++){
                for(i=1;i<=N;i++) col[i] = 0.0;
                col[j] = 1.0;
                lubksb(Atemp,N,indx,col);
                for(i=1;i<=N;i++) inverted[i][j] = col[i];
        }


        // Ad
        for(i=1;i<=N;i++){
        for(j=1;j<=N;j++){
                Apart[i][j] = Ac[i][j];
                }
        }
    matrix_scalar_mult(Apart, N, N, (1/k));                                                 //
Perform (1/k)*Ac
       matrix_add(Apart, identity, order, order);                                  // Perform
I+(1/k)*Ac
    matrix_mult(Ad,Apart,inverted,N,N,N,N);                                        // Calculate
Ad
```

```
        // Bd
        matrix_mult(Bd,inverted,Bc,N,IN,N,N);
        //matrix_scalar_mult(Bd, N, IN, (2*k)/r);

        // Cd & Dd
        matrix_mult(Cd,Cc,inverted,IN,N,N,N);
        matrix_mult(Dd,Cd,Bc,IN,N,N,IN);
        matrix_scalar_mult(Cd,IN,N,r/209.985);
        matrix_scalar_mult(Dd,IN,IN,(1/k));
        matrix_add(Dd,Dc,IN,IN);

        while(1)
        {
        }

}

void nrerror(char error_text[])
// Numerical Recipes Standard Error Handler
{
        fprintf(stderr,"Numerical Recipies run-time error...\n");
        fprintf(stderr,"%s\n",error_text);
        fprintf(stderr,"...now exiting to system...\n");
        exit(1);
}

float **matrix(long nrl, long nrh, long ncl, long nch)
// Source: Numerical Recipes in C (Press, et al)
// Allocate a float matrix with subscript range m[nrl-nrh][ncl-nch]
{
        long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
        float **m;

        // allocate pointers to rows
        m=(float **) malloc((size_t)((nrow+NR_END)*sizeof(float*)));
        if (!m) nrerror("allocation failure 1 in matrix()");
        m += NR_END;
        m -= nrl;

        // allocate rows and set pointers to them
        m[nrl]=(float *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(float)));
        if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
        m[nrl] += NR_END;
        m[nrl] -= ncl;

        for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

        // return pointer to array of pointers to rows
        return m;

}

int *ivector(long nl, long nh)
// allocate an int vector with subscript range v[nl..nh]
// Source: Numerical Recipes in C (Press, et al)
{
        int *v;

        v=(int *)malloc((size_t)((nh-nl+1+NR_END)*sizeof(int)));
        if (!v) nrerror("allocation failure in ivector()");
        return v-nl+NR_END;
}

// Creates an identity matrix (square) of a given order
void identity_builder(void)
{
        int n,k;

        for(n=1;n<=order;n++){
                for(k=1;k<=order;k++){
                        if(n==k)
```

```
                                identity[n][k] = 1;
                        else
                                identity[n][k] = 0;
                }
        }
}

// Multiplies a matrix by a scalar value
void matrix_scalar_mult(float **a, int rows, int cols, float scalar)
{
        int n,k;

        for(n=1;n<=rows;n++){
                for(k=1;k<=cols;k++){
                        a[n][k] *= scalar;
                }
        }
}

// Multiply two matrices together
void matrix_mult(float **out, float **m1, float **m2, int r1, int c1, int r2, int c2)
{
        int i,j,k;      //for loop placeholders

        for(i=1;i<=r1;i++){
                for(j=1;j<=c2;j++){
                        out[i][j]=0;
                        for(k=1;k<=r2;k++){
                                out[i][j]+=m1[i][k]*m2[k][j];
                                //ie:
mult[0][0]=m1[0][0]*m2[0][0]+m1[0][1]*m2[1][0]+m1[0][2]*m2[2][0];
                        }
                }
        }
}

// Add two matrices together
void matrix_add(float **a, float **b, int rows, int cols)
{
//  NOTE: MATRICES MUST BE OF IDENTICAL DIMENSIONS
//      The matrices A and B will be added together and the result returned in A

        int i,j;

        for(i=1;i<=rows;i++){
                for(j=1;j<=cols;j++){
                        a[i][j] += b[i][j];
                }
        }

}



// Subtract one matrix from another
void matrix_sub(float **a, float **b, int rows, int cols)
{
//  NOTE: MATRICES MUST BE OF IDENTICAL DIMENSIONS
//      The matrix A will be subtracted from B and the result returned in B

        int i,j;

        for(i=1;i<=rows;i++){
                for(j=1;j<=cols;j++){
                        b[i][j] = a[i][j] - b[i][j];
                }
        }

}
```

```
void ludcmp(float **a, int n, int *indx, float *d)
// Source: Numerical Recipes in C (Press, et al)
{
	int i,imax,j,k;
	float big,dum,sum,temp;
	float *vv;                                  //vv stores the implicit scaling of each row

	vv = vector(1,n);
	*d = 1.0;
	for(i=1;i<=n;i++){
		big = 0.0;
		for(j=1;j<=n;j++)
			if((temp=fabs(a[i][j])) > big) big = temp;
		if(big == 0.0) nrerror("Singular matrix in routine ludcmp");
		vv[i] = 1.0/big;              // save scaling
	}
	for(j=1;j<=n;j++){
		for(i=1;i<j;i++){
			sum = a[i][j];
			for(k=1;k<i;k++) sum -= a[i][k]*a[k][j];
			a[i][j] = sum;
		}
		big = 0.0;
		for(i=j;i<=n;i++){
			sum = a[i][j];
			for(k=1;k<j;k++)
				sum -= a[i][k]*a[k][j];
			a[i][j] = sum;
			if((dum=vv[i]*fabs(sum)) >= big) {
				big = dum;
				imax = i;
			}
		}
		if(j != imax){
			for(k=1;k<=n;k++){
				dum = a[imax][k];
				a[imax][k] = a[j][k];
				a[j][k] = dum;
			}
			*d = -(*d);
			vv[imax] = vv[j];
		}
		indx[j] = imax;
		if(a[j][j] == 0.0)
			a[j][j] = TINY;
		if(j != n){
			dum = 1.0/(a[j][j]);
			for(i=j+1;i<=n;i++) a[i][j] *= dum;
		}
	}
		free_vector(vv,1,n);
}

void lubksb(float **a, int n, int *indx, float b[])
// Source: Numerical Recipes in C (Press, et al)
{
	int i,ii=0,ip,j;
	float sum;

	for(i=1;i<=n;i++){
		ip = indx[i];
		sum = b[ip];
		b[ip] = b[i];
		if(ii)
			for(j=ii;j<=i-1;j++) sum -= a[i][j] * b[j];
		else if (sum) ii = i;
		b[i] = sum;
	}
	for(i=n;i>=1;i--){
		sum = b[i];
		for(j=i+1;j<=n;j++) sum -= a[i][j] * b[j];
```

```
                b[i] = sum/(a[i][i]);
        }
}

float *vector(long nl, long nh)
// Source: Numerical Recipes in C (Press, et al)
// allocate a float vector with subscript range v[nl..nh]
{
        float *v;

        v = (float *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(float)));
        if (!v) nrerror("allocation failure in vector()");
        return v-nl+NR_END;
}

void free_vector(float *v, long nl, long nh)
// Source: Numerical Recipes in C (Press, et al)
// free a float vector allocated with vector()
{
        free((FREE_ARG) (v+nl-NR_END));

}


void hook_int()
// Source: Professor Donald Richard Brown III
{
        IRQ_globalDisable();          // Globally disables interrupts
        IRQ_nmiEnable();              // Enables the NMI interrupt
        IRQ_map(IRQ_EVT_RINT1,15);    // Maps an event to a physical interrupt
        IRQ_enable(IRQ_EVT_RINT1);    // Enables the event
        IRQ_globalEnable();           // Globally enables interrupts
}


interrupt void serialPortRcvISR()
{
        int n=0;
        int k=0;
        float u = 0.0, output;


        union {Uint32 combo; short channel[2];} temp;
        temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
        u = temp.channel[0];                  //write new sample to input var

        for(k=1;k<=N;k++){                                          // choose which matrix row
                x_next[k] = Bd[k][1] * u;                           // do B matrix multiplication
                for(n=1;n<=N;n++){                                  // choose which matrix column
                        x_next[k] += Ad[k][n] * x[n];        // do A matrix multiplication
                }
        }

        output = 0.0;

        for(n=1;n<=N;n++){                    //loop to compute output, mac'ing C matrix and x's
                output+=Cd[1][n] * x[n];
        }

        output += u * Dd[1][1];

        temp.channel[0] = (short)output;      //cast output as short, write to MCBSP buffer

        MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo); //write output

        for(n=1;n<=N;n++){                            //move in new x values
                x[n] = x_next[n];
        }
}
```

# Appendix E: Daughterboard Test Code

```
#define CHIP_6713
#include <stdio.h>
#include <c6x.h>
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include "dsk6713.h"
#include "dsk6713_aic23.h"
#include "winDSK6_def.h"
#include "hpi_services.h"
#include "hpi_services_funcs.h"
#define NUMBER_OF_HPI_INIT_RETRIES    100000                // retries on initialization

HPI_SVC_BLOCK HpiSvcBlock = {
        0, // flag
        HPI_SVC_BAUD115200 // default baud rate
};

DSK6713_AIC23_CodecHandle hCodec;
DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;

// Codec configuration with default settings
interrupt void serialPortRcvISR(void);
void hook_int();

double input = 0.0;
double output = 0.0;
double next_output = 0.0;

float gain = 0.0;

const float R = 9980.0;
const float C = 0.0000000048;
const float Fs = 44100.0;
double k = 0.0;

void main()
{
        unsigned short analog_in;

        DSK6713_init(); // Initialize the board support library
        hCodec = DSK6713_AIC23_openCodec(0, &config);
        MCBSP_FSETS(SPCR1, RINTM, FRM);
        MCBSP_FSETS(SPCR1, XINTM, FRM);
        MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
        MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
        DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_44KHZ);
        // daughterboard setup
        EnableAnalog(0xFF);
        StartHpiServices();
        hook_int();

        while(1)
        {
                if(IsHpiDataNew()){
                        analog_in = ReadAnalog(0);
                        gain = ((float)analog_in)/((float)HPI_SVC_ANALOG_MAX_VALUE);
                }
        }
}

void hook_int()
// Source: Professor Donald Richard Brown III
{
        IRQ_globalDisable();          // Globally disables interrupts
        IRQ_nmiEnable();                      // Enables the NMI interrupt
        IRQ_map(IRQ_EVT_RINT1,15);    // Maps an event to a physical interrupt
        IRQ_enable(IRQ_EVT_RINT1);    // Enables the event
        IRQ_globalEnable();           // Globally enables interrupts
```

```
}

int StartHpiServices()
// Source: EducationalDSP
{
	unsigned int i;

//	HpiSvcBlock.ddir = 0xFFFF;					// all digital pins inputs
//	HpiSvcBlock.aen = 0;						// all analog disabled
	HpiSvcBlock.com_tx_head = 0;					// show all buffers empty
	HpiSvcBlock.com_rx_head = 0;
	HpiSvcBlock.usb_tx_head = 0;
	HpiSvcBlock.usb_rx_head = 0;
	HpiSvcBlock.com_tx_tail = 0;
	HpiSvcBlock.com_rx_tail = 0;
	HpiSvcBlock.usb_tx_tail = 0;
	HpiSvcBlock.usb_rx_tail = 0;

	*(unsigned int *)HPI_SVC_XFER_ADDRESS = (unsigned int)&HpiSvcBlock;	// store data
structure address
	HpiSvcBlock.flag = DSP_TO_HOST_MAGIC_NUMBER;			// store magic number

	// signal HPI daughtercard to start services by writing 1 to HINT bit in HPIC
	*(unsigned int *)0x01880000 = 0x00000004;

	for(i = 0;i < NUMBER_OF_HPI_INIT_RETRIES;i++)			// wait for daughtercard to
reply
		if(HpiSvcBlock.flag == HOST_TO_DSP_MAGIC_NUMBER)
			return 1;

	return 0;
}

int IsHpiDataNew()
// Source: EducationalDSP
{
	static unsigned short last_flag = 0;

	if(last_flag != HpiSvcBlock.flag) {
		last_flag = HpiSvcBlock.flag;
		return 1;
	}
	return 0;
}

void EnableAnalog(unsigned short number)
// Source: EducationalDSP
{
	HpiSvcBlock.aen = number;
}

unsigned short ReadAnalog(unsigned short pin)
// Source: EducationalDSP
{
	switch(pin) {
	case 0:
		return HpiSvcBlock.an0;
	case 1:
		return HpiSvcBlock.an1;
	case 2:
		return HpiSvcBlock.an2;
	case 3:
		return HpiSvcBlock.an3;
	case 4:
		return HpiSvcBlock.an4;
	case 5:
		return HpiSvcBlock.an5;
	case 6:
		return HpiSvcBlock.an6;
	case 7:
		return HpiSvcBlock.an7;
```

```
        }
        return 0; // out of bounds
}

interrupt void serialPortRcvISR()
{
        union {Uint32 combo; short channel[2];} temp;

        temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);

        input = temp.channel[0];

        next_output = gain*(.3827*input + .6173*output);

        temp.channel[0] = (short)next_output;

        MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo);

        output = next_output;

}
```

## Appendix F: Bilinear Algorithm Verification (MATLAB)

```
clear all
clc

fs = 96000;
T = 1/fs;

R1 = 10000; R2 = 22412.2; C1 = 1.0E-9; C2 = 1.0E-9;

k = 2*fs;
r = sqrt(T);

A = [ -((1/(C1*R1))+(1/(C1*R2))) 1/(C1*R2) ;  1/(C2*R2) -1/(C2*R2)];

%A = [ -138084.6, 38084.61; 38084.61, -38084.61];

B = [ 1/(C1*R1) ; 0];
C = [0 1];
D = 0;

I = [1,0;0,1];

invert = (I-(1/k)*A);

t1 = eye(size(A)) + A*T/2;
t2 = eye(size(A)) - A*T/2;

Ad = t2\t1
Bd = T/r*(t2\B)
Cd = r*C/t2
Dd = C/t2*B*T/2 + D

[Ad1,Bd1,Cd1,Dd1] = bilinear(A,B,C,D,fs)

Z = I-A*T/2;

inverted = inv(invert);

Ad3 = t1*inverted
Bd3 = T/r*(inverted*B)
Cd3 = r*C*inverted
Dd3 = C*inverted*B*T/2 + D

system = ss(Ad1,Bd1,Cd1,Dd1,T);
system2 = ss(Ad,Bd,Cd,Dd,T);
cont = ss(A,B,C,D);
disc = c2d(cont,T,'tust');
impinv = c2d(cont,T,'imp');

bode(system,system2,cont,disc)
```

## Appendix G: Sampling Frequency Analysis (MATLAB)

```
clear all
clc

fs1 = 44100;
T1 = 1/fs1;

fs2 = 48000;
T2 = 1/fs2

fs3 = 96000;
T3 = 1/fs3;

fs4 = 192000;
T4 = 1/fs4;

R1 = 10000; R2 = 10000; C1 = 1.0E-9; C2 = 1.0E-9;

A = [ -((1/(C1*R1))+(1/(C1*R2))) 1/(C1*R2) ;  1/(C2*R2) -1/(C2*R2)];
B = [ 1/(C1*R1) ; 0];
C = [0 1];
D = 0;

[Ad1,Bd1,Cd1,Dd1] = bilinear(A,B,C,D,fs1);
[Ad2,Bd2,Cd2,Dd2] = bilinear(A,B,C,D,fs2);
[Ad3,Bd3,Cd3,Dd3] = bilinear(A,B,C,D,fs3);
[Ad4,Bd4,Cd4,Dd4] = bilinear(A,B,C,D,fs4);

Discrete1 = ss(Ad1,Bd1,Cd1,Dd1,T1);
Discrete2 = ss(Ad2,Bd2,Cd2,Dd2,T2);
Discrete3 = ss(Ad3,Bd3,Cd3,Dd3,T3);
Discrete4 = ss(Ad4,Bd4,Cd4,Dd4,T4);
Continuous = ss(A,B,C,D);

bode(Discrete1,Discrete2,Discrete3,Discrete4,Continuous)
```

## Appendix H: Sallen-Key Band Pass Verification and Analysis (MATLAB)

```
r1 = audiorecorder(44100,16,2);
record(r1); % start recording
pause(10); % pause for 10 seconds
stop(r1); % stop recording
y1 = getaudiodata(r1); % extract the audio data
max(abs(y1)) % make sure no clipping
pause
r2 = audiorecorder(44100,16,2);
record(r2); % start recording
pause(10); % pause for 10 seconds
stop(r2); % stop recording
y2 = getaudiodata(r2); % extract the audio data
max(abs(y2)) % make sure no clipping


fs = 44100;
Ts = 1/fs;

[Py1,f] = pwelch(y1(:,1),1024,512,1024,fs); % estimate psd of left channel
[Py2,f] = pwelch(y1(:,2),1024,512,1024,fs); % estimate psd of right channel
[Py3,f] = pwelch(y2(:,1),1024,512,1024,fs); % estimate psd of left channel
[Py4,f] = pwelch(y2(:,2),1024,512,1024,fs); % estimate psd of right channel

C1 = 4.7e-9; C2 = 4.7e-9; R1 = 5100; R2 = 10000; Rf = 10000;

b = [1/(R1*C1) 0];
a = [1 ((1/(R1*C1))+(1/(R2*C1))+(1/(R2*C2))-(1/(Rf*C1)))
((R1+Rf)/(R1*Rf*R2*C1*C2))];

T = 10; % duration
x = randn(fs*T,1); % stereo white noise
x = 0.99*x/max(max(abs(x))); % normalize

bandpass = tf(b,a);
bandpass_tust = c2d(bandpass,Ts,'tustin');

[A,B,C,D] = tf2ss(b,a);
bandpass_ss = ss(A,B,C,D);
bandpass_ss_tust = c2d(bandpass_ss,Ts,'tustin');
bandpass_ss_zoh = c2d(bandpass_ss,Ts,'zoh');
bandpass_ss_foh = c2d(bandpass_ss,Ts,'foh');

filtered = filter([.2507 0 -.2507],[1 -.8775 .2428],x);
filtered_cont = filter(b,a,x);

[Py5,f] = pwelch(filtered,1024,512,1024,fs);
[Py6,f] = pwelch(filtered_cont,1024,512,1024,fs);

figure(1);
semilogx(f,10*log10(Py1)+67.2,f,10*log10(Py4)+71.5,f,10*log10(Py5)+57.5); %
plot psd in dB
grid on
legend('Buffered Analog Filter','DSP Filter','Discrete SS');
```

```
xlabel('Frequency (Hz)');
ylabel('Magnitude Response (dB)');

figure(2)
bode(bandpass,bandpass_ss_disc);
```

## Appendix I: Discretization Method Analysis (MATLAB)

```
R = 10000;
C = 4.7e-9;
Fs = 44100;
Ts = 1/Fs;
k = 1/(R*C*Fs);
RC = tf([0 k],[1 -(1-k)],Ts);

RC_cont = tf([1],[R*C 1]);

[A0,B0,C0,D0] = tf2ss([1],[R*C 1]);
RC_ss = ss(A0,B0,C0,D0);

fs = 44100; Ts = 1/fs;

RC_zoh = c2d(RC_ss,Ts,'zoh');
RC_foh = c2d(RC_ss,Ts,'foh');
RC_imp = c2d(RC_ss,Ts,'imp');
RC_tust = c2d(RC_ss,Ts,'tustin');
RC_pre = c2d(RC_ss,Ts,'prewarp',21200);
RC_matched = c2d(RC_ss,Ts,'matched');

figure(1)
bode(RC_ss,RC_zoh,RC_foh,RC_imp,RC_matched,RC_pre,RC_tust)
legend('Continuous','zoh','foh','imp','matched','pre','tust')

figure(2)
bode(RC_ss,RC_matched,RC_pre,RC_tust)
legend('Continuous','Matched Pole-Zero','Tustin with Frequency
Prewarping','Tustin (bilinear)')

[A1,B1,C1,D1] = tf2ss([1],[R*R*C*C (R*C + R*C) 1]);
SK_ss = ss(A1,B1,C1,D1);
SK_zoh = c2d(SK_ss,Ts,'zoh');
SK_foh = c2d(SK_ss,Ts,'foh');
SK_tust = c2d(SK_ss,Ts,'tust');

figure(3);
bode(SK_ss,SK_zoh,SK_foh,SK_tust)
legend('cont','zoh','foh','tust')

Q1 = 4.7e-9; Q2 = 4.7e-9; R1 = 5100; R2 = 10000; Rf = 10000;

b = [1/(R1*Q1) 0];
a = [1 ((1/(R1*Q1))+(1/(R2*Q1))+(1/(R2*Q2))-(1/(Rf*Q1)))
((R1+Rf)/(R1*Rf*R2*Q1*Q2))];
[A2,B2,C2,D2] = tf2ss(b,a);
BP_ss = ss(A2,B2,C2,D2);
BP_zoh = c2d(BP_ss,Ts,'zoh');
BP_tust = c2d(BP_ss,Ts,'tust');

figure(4);
bode(BP_ss,BP_zoh,BP_tust);
legend('cont','zoh','tust')
```