

VCO-based $\Sigma\Delta$ ADC Design Examples

Matthew Straayer

May, 2008

Copyright © 2008 by Matthew Straayer
All rights reserved.

Table of Contents

Setup	2
Introduction.....	5
VCO-Based Quantizer Background.....	6
Design Goals for an Example VCO-Based Quantizer.....	7
Basic VCO-Based ADC Simulations Using Sue2 and CppSimView	7
A. Opening Sue2 Schematics.....	7
B. Running the CppSim Simulation	10
C. Plotting the Time Domain Results.....	12
Post-Processing the Simulation Results Using a MATLAB Script.....	14
Design of a 4 th Order Loop Filter.....	15
A. Ideal Noise Transfer Function Design	15
B. Including Excess Loop Delay	17
C. Convert from Discrete to Continuous Time.....	20
D. Calculating Loop Filter Values.....	21
Simulating the 4 th Order Loop Filter.....	21
Conclusion	24

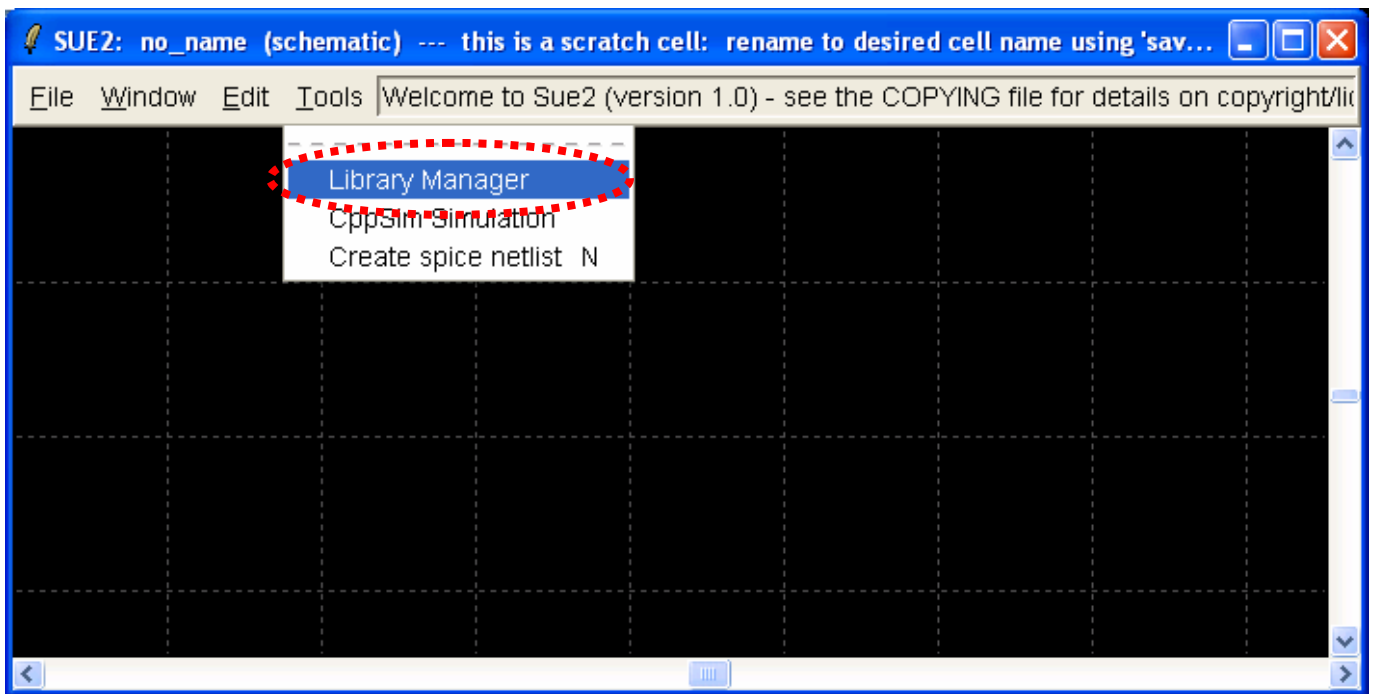
Setup

Download and install the CppSim Version 3 package (i.e., download and run the self-extracting file named **setup_cppsim3.exe**) located at:

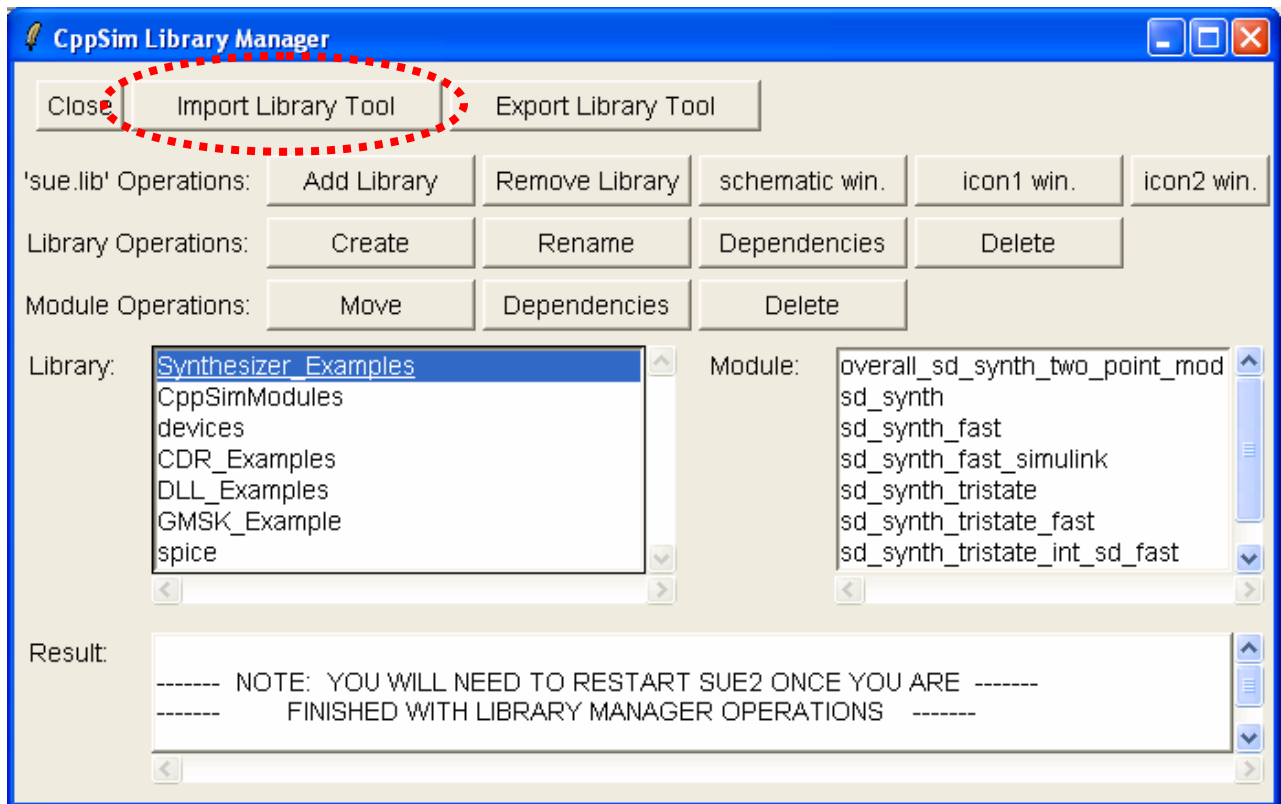
<http://www.cppsim.com>

Upon completion of the installation, you will see icons on the Windows desktop corresponding to the PLL Design Assistant, CppSimView, and Sue2. Please read the “**CppSim (Version 3) Primer**” document, which is also at the same web address, to become acquainted with CppSim and its various components.

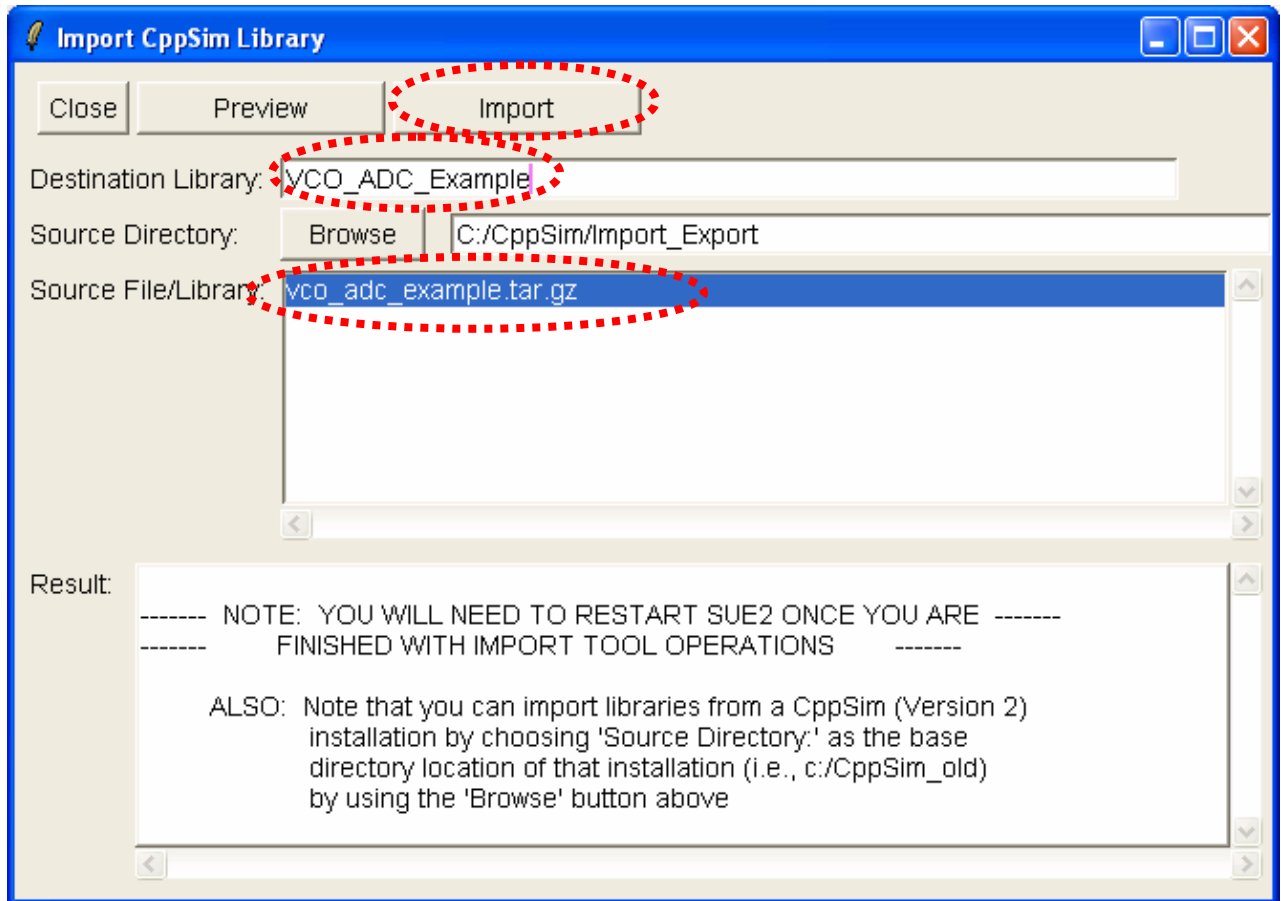
To run this tutorial, you will also need to download the file **vco_adc_example.tar.gz** available at <http://www.cppsim.com>, and place it in the **Import_Export** directory of CppSim (assumed to be **c:/CppSim/Import_Export**). Once you do so, start up **Sue2** by clicking on its icon, and then click on **Tools->Library Manager** as shown in the figure below.



In the **CppSim Library Manager** window that appears, click on the **Import Library Tool** button as shown in the figure below.



In the **Import CppSim Library** window that appears, change the **Destination Library** to **VCO_ADC_Example**, click on the **Source File/Library** labeled as **vco_adc_example.tar.gz**, and then press the **Import** button as shown in the figure below. Note that if **vco_adc_example.tar.gz** does not appear as an option in the **Source File/Library** selection listbox, then you need to place this file (downloaded from <http://www.cppsim.com>) in the **c:/CppSim/Import_Export** directory.



Once you have completed the above steps, restart **Sue2** as directed in the above figure.

Introduction

Advanced CMOS processes offer very fast switching speed and high transistor density that can be utilized to implement analog signal processing functions in interesting and unconventional ways, for example by leveraging time as a signal domain. In this context, voltage controlled ring oscillators are circuit elements that are not only very attractive due to their highly digital implementation which takes advantage of scaling, but also due to their ability to amplify or integrate conventional voltage signals into the time domain.

In [1], the use of VCO-based quantization within continuous-time (CT) $\Sigma\Delta$ analog-to-digital converter (ADC) structures is explored, with a custom prototype in 0.13 μm CMOS showing measured performance of 86/72 dB SNR/SNDR with 10 MHz bandwidth while consuming 40mW from a 1.2V supply and occupying an active area of 640 μm x 660 μm . A key element of the ADC structure is a 5-bit VCO-based quantizer clocked at 950MHz, which achieves first-order noise shaping of its quantization noise. The quantizer structure allows the second-order CT $\Sigma\Delta$ ADC topology to achieve third-order noise shaping, and direct connection of the VCO-based quantizer to the internal DACs of the ADC provides *intrinsic* dynamic element matching of the DAC elements.

This tutorial will focus on demonstrating the advantages of feedback to improve the performance of the VCO-based quantizer, comparing the stand-alone quantizer to loops with 2nd and 4th order dynamics, which results in 1st, 3rd, and 5th order quantization noise-shaping, respectively. We recommend first reading [1] to understand the VCO-based quantizer architecture, as well as the tradeoffs for the quantizer compared to a more traditional FLASH-based quantizer. For a point of comparison, the 2nd order loop filter in this tutorial is intended to approximate the implementation reported in [1].

Although MATLAB is not required for CppSim simulations, we provide scripts to post process the simulation results, and additionally we provide a MATLAB script for calculating continuous time 4th order loop filter coefficients in the presence of excess loop delay. This second script makes use of the Delta Sigma Toolbox [2], a well-known tool that is currently available as a free download from the MATLAB Central File Exchange.

[1] M. Straayer, and M.H. Perrott, "A 12-Bit, 10-MHz Bandwidth, Continuous-Time $\Sigma\Delta$ ADC With a 5-Bit, 950-MS/s VCO-Based Quantizer," IEEE JSSC, Vol. 43, No. 4, April 2008, pp. 805-814

[2] R. Schreier, Delta Sigma Toolbox [online].
<http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=19&objectType=file>

VCO-Based Quantizer Background

In Fig. 1, we show a simple VCO-based quantizer implementation that requires a set of standard registers, XOR gates, and a final adder stage. We see that the relative simplicity of this circuit allows high speed operation with small latency, which are important characteristics when placing the VCO-based quantizer within a CT ADC structure. Note that in this structure, the proposed quantizer expands the number of delay elements such that the number of transitions per clock period is smaller than that number of elements.

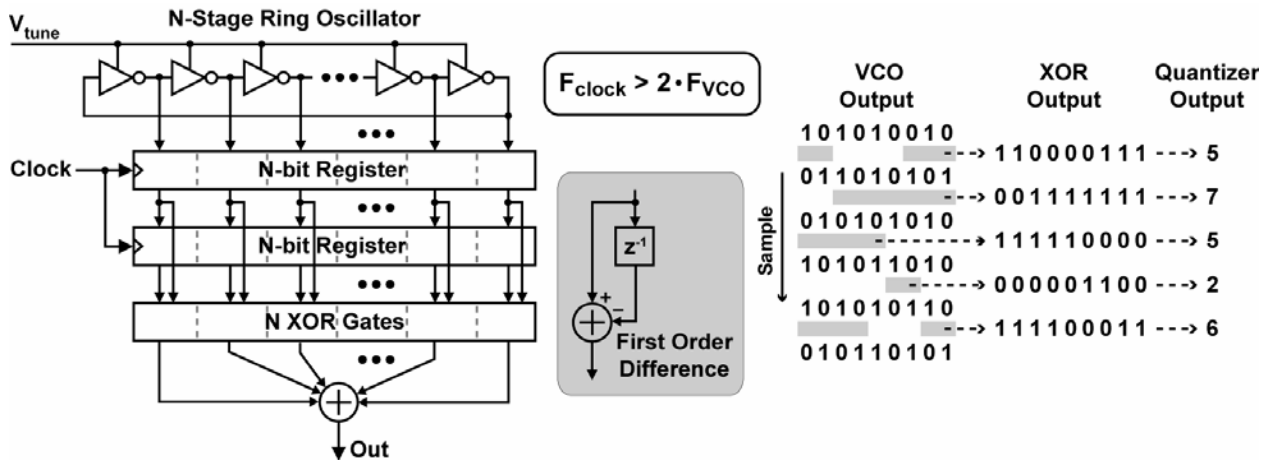


Figure 1: High-speed implementation of a VCO-based quantizer.

The binary sequences shown in Fig. 1 are useful for understanding the operation of the high speed quantizer structure. The key idea is to observe whether a given VCO delay cell undergoes a transition within a given clock period by comparing samples of its current and previous states with an XOR operation. The *number* of VCO delay cells that undergo a transition within a given clock period is a function of the delay through each stage as set by the V_{tune} voltage, which yields the quantized value of the voltage V_{tune} that we seek.

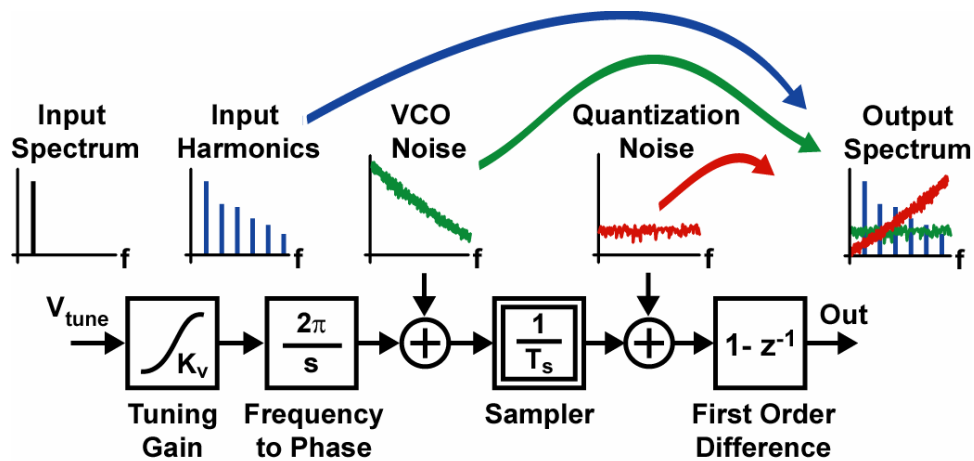


Figure 2: Block diagram of the VCO-based quantizer's behavioral model.

A key observation offered by Fig. 1 is that the quantization noise is first-order noise-shaped by virtue of the first-order difference operation shown in the figure. In Fig. 2, we also see that the VCO phase noise is shaped as well, but the result of such shaping is a flat spectrum due to the 20 dB/dec slope of the original phase-noise signal. In reality, the shaped VCO phase noise will also include $1/f$ noise, but this is ignored here for the sake of modeling simplicity. Also shown in Fig. 2 is that the voltage-to-frequency tuning curve of a VCO is quite nonlinear in practice. The impact of such nonlinearity is to introduce harmonic distortion which can significantly degrade the SNDR performance of the quantizer.

Design Goals for an Example VCO-Based Quantizer

As an initial target for the VCO-based quantizer, we set the following specifications:

- 950MHz sampling rate (F_s)
- Nominal delay per stage of twice the minimum inverter delay of 30-35ps, or 65ps
- K_v near 750MHz/V
- 5-10% tuning non-linearity
- -100dBc/Hz VCO phase noise at 1MHz offset frequency
- 20MHz analog input bandwidth

If we assume that the nominal oscillator frequency, F_{VCO} , is half of its maximum value (such that half of the elements transition for zero input), then we are left with requiring a sampling rate that is four times the nominal VCO frequency. Thus, we have the number of stages, N , to be

$$N = 2/T_{\text{delay}}/F_s = 31,$$

which is equivalent to 5-bits. The K_v near 750MHz/V then restricts the maximum input signal to be $\pm 300\text{mV}$.

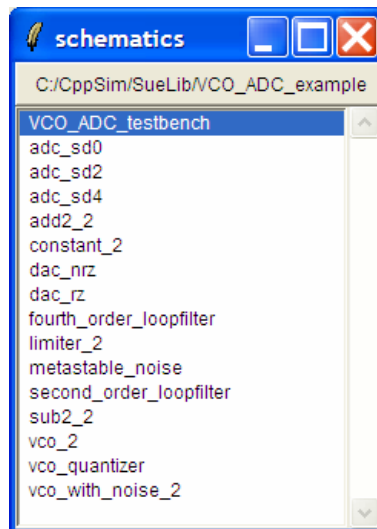
With the high-level parameters of the quantizer now in place, we are ready to begin simulating a stand-alone VCO-based quantizer in order to evaluate its performance.

Basic VCO-Based ADC Simulations Using Sue2 and CppSimView

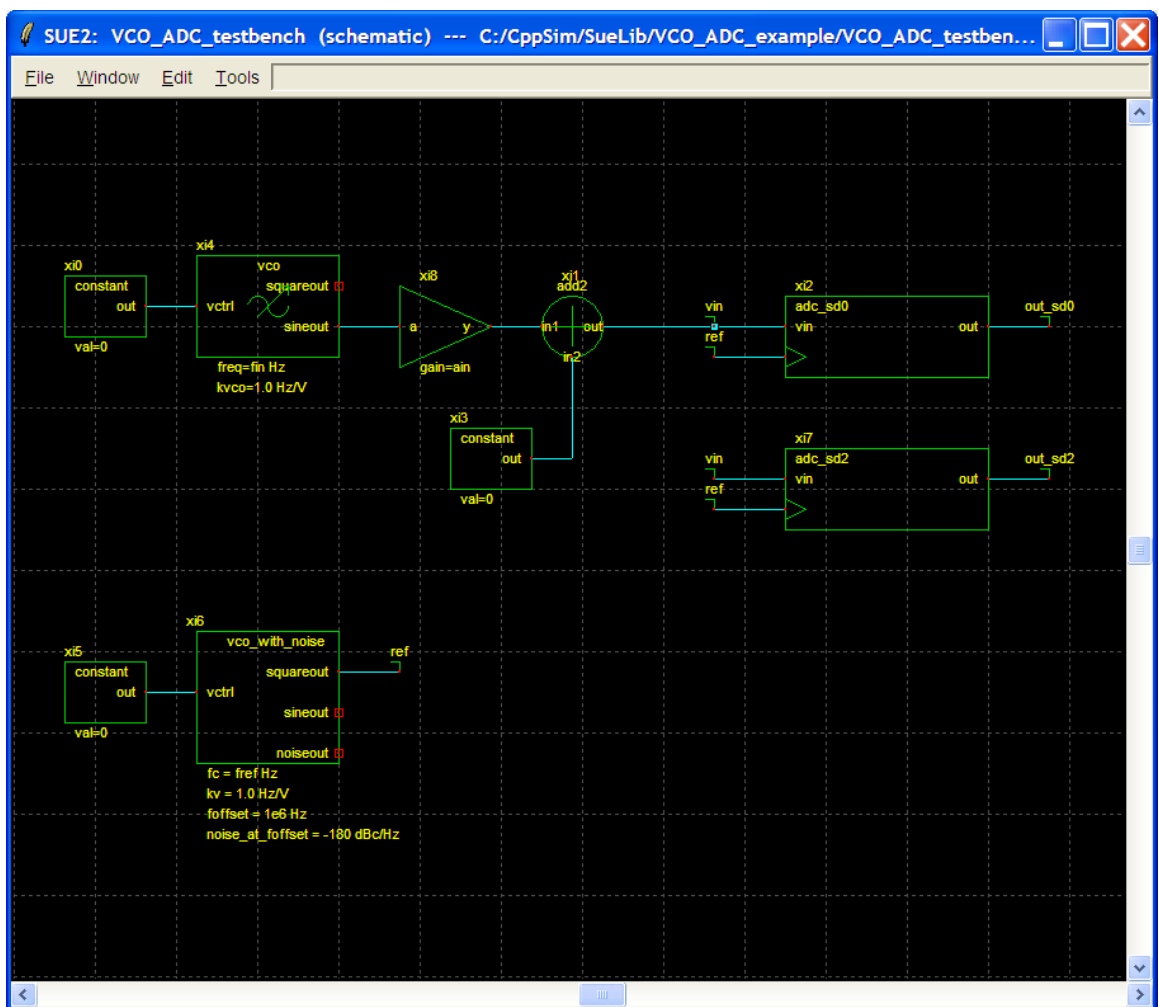
In this section, the user will be guided through basic tasks such as opening the ADC example within the Sue2 schematic editor and running basic CppSim simulations.

A. Opening Sue2 Schematics

- Click on the Sue2 icon to start Sue2, and then select the **VCO_ADC_example** library from the **schematic listbox**. The **schematic listbox** should now look as follows:

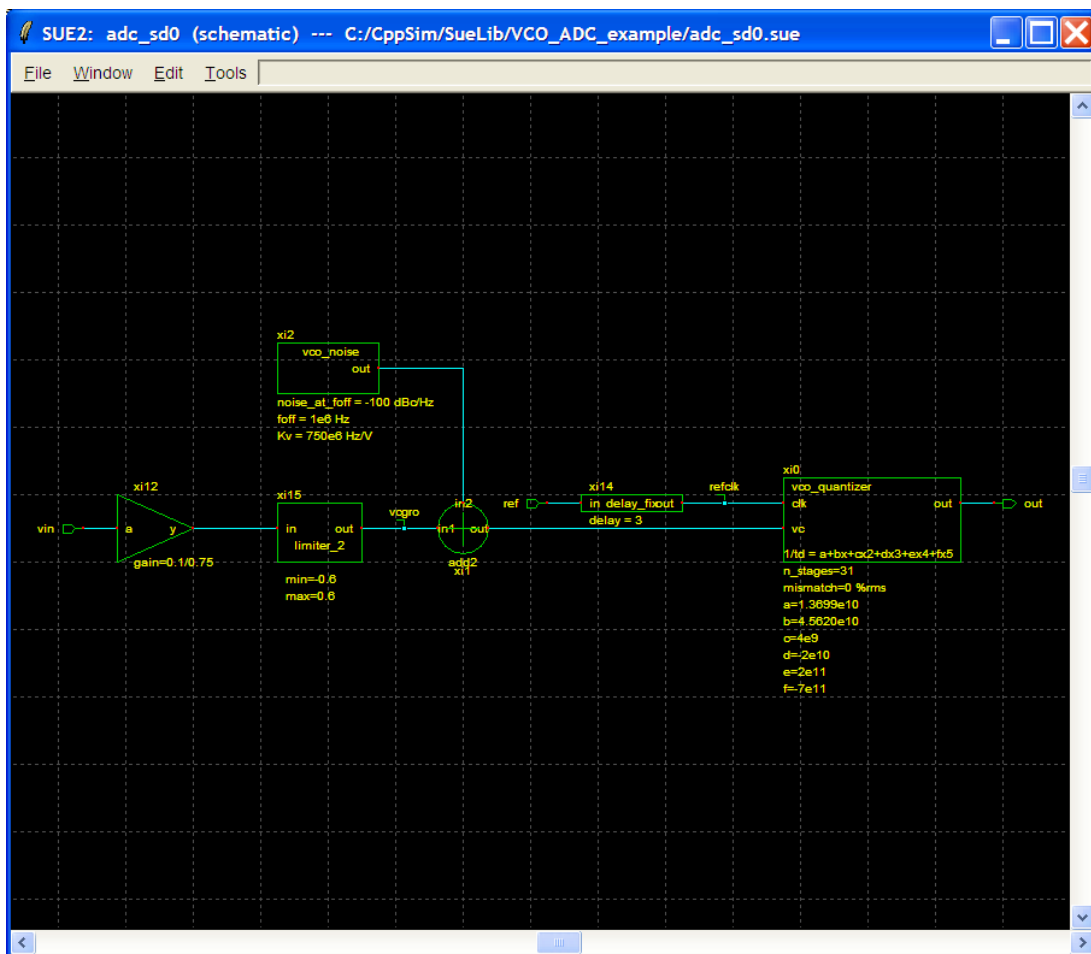


- Click on the **VCO_ADC_testbench** schematic. The schematic window should now display the input signal blocks as well as two ADC blocks as shown below:

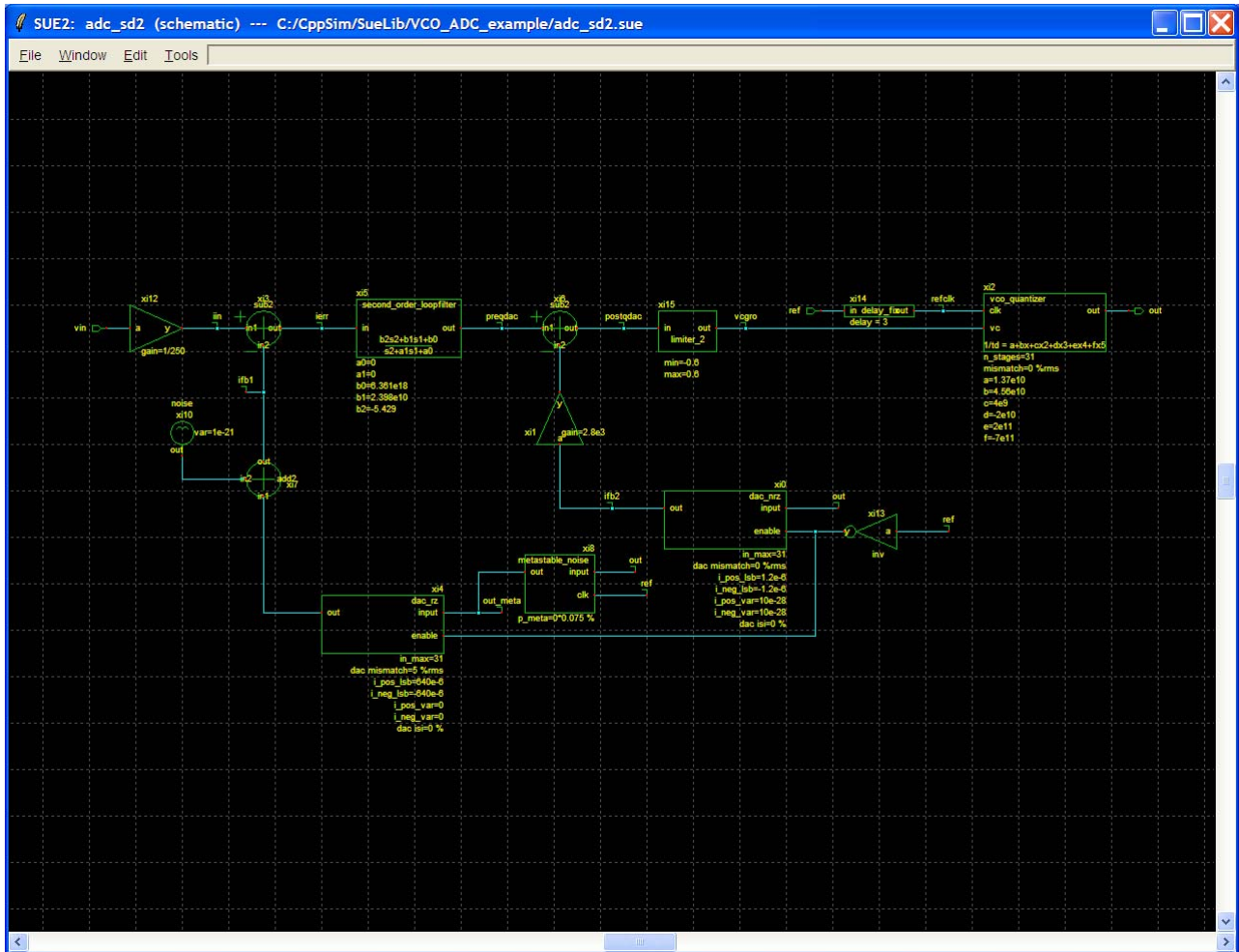


- The key blocks in the testbench are the stand-alone VCO-based quantizer ADC, **adc_sd0**, and the second-order $\Sigma\Delta$ ADC, **adc_sd2**. The “0” suffix indicates that there is no loop filter preceding the VCO-based quantizer, in contrast to a “2” suffix for an ADC with second-order dynamics, and later a “4” for a fourth-order loop filter.
- After clicking on **adc_sd0**, press **e** to descend into its schematic. You will see the schematic depicted below, which under normal conditions includes gain, noise, and the **vco_quantizer** block. The **vco_quantizer** block is a module with coded functionality, which can be viewed by double-clicking on the cell and then selecting the **Edit CppSim Code** button.

After closing any the **vco_quantizer** cell window, you may now press **Ctrl+e** to return to the testbench top level.



- Descend into **adc_sd2** (press **e**). Now in addition to the **vco_quantizer** cell, as shown below there are two DAC cells, **dac_rz** and **dac nrz**, a 2nd order loop filter, **second_order_loopfilter**, and some additional scale factors and noise contributions. These cells are all coded with behavioral models directly, and include many fundamental as well as non-ideal models for inter-symbol interference, noise, and mismatch.



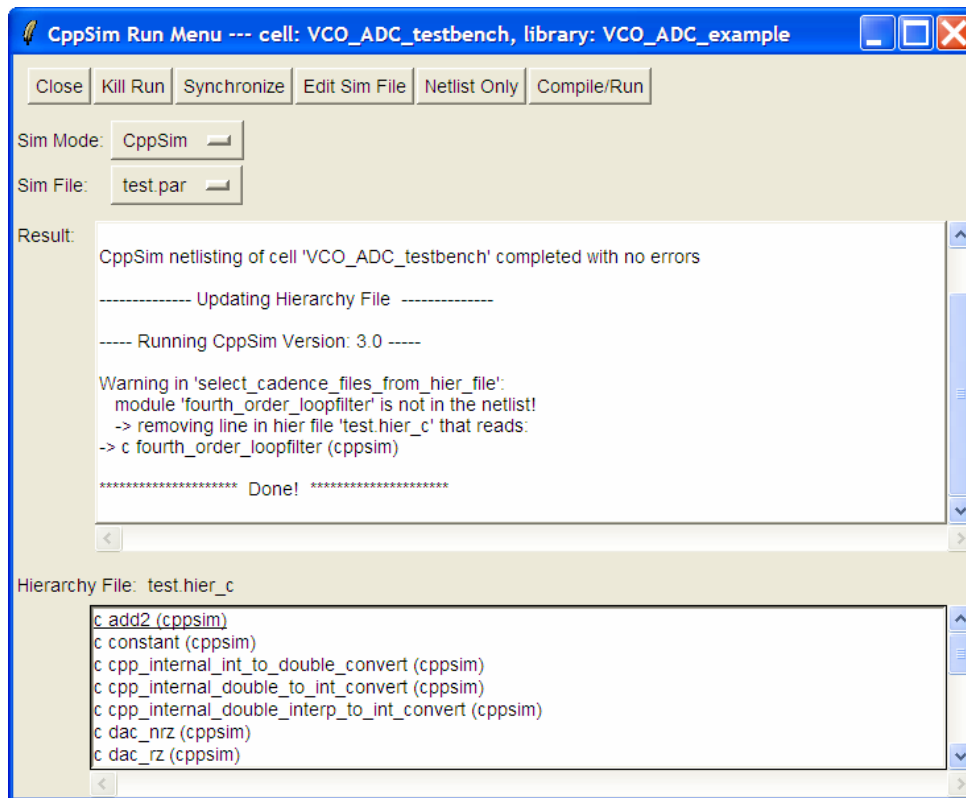
You may now press **Ctrl+e** to return to the testbench top level.

- The key signals present in the testbench are clearly:
 - **ref**: The sampling clock, nominally at 950MHz.
 - **vin**: This is the input signal to the ADC, although its amplitude is modified within **adc_sd0** for reasons that will be more clear later in the tutorial.
 - **out_sd0**: This is the digitized output of the converter.

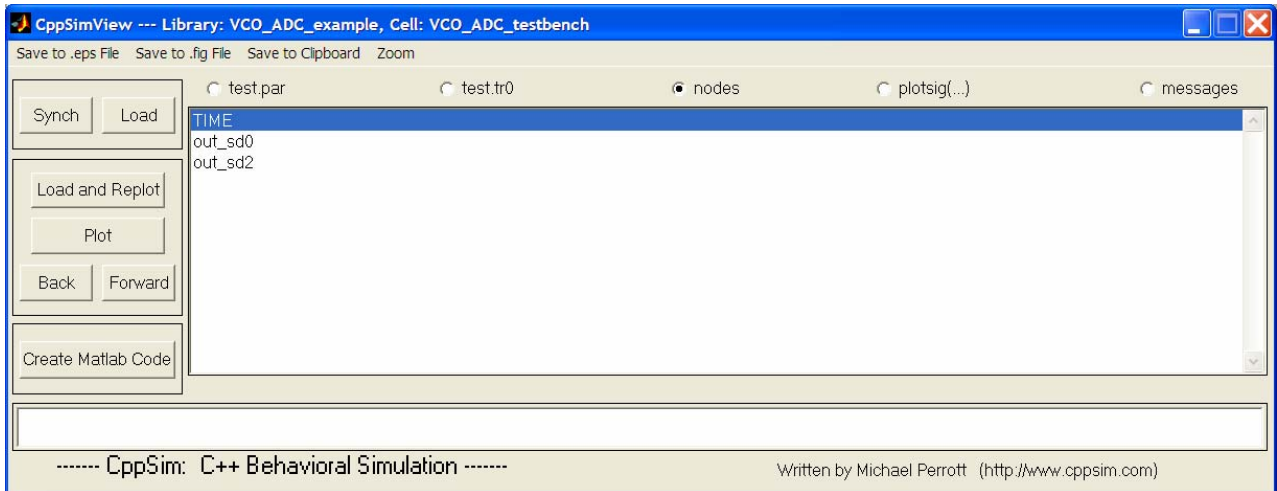
The model is setup in a way that you can bring the three parameters **fin**, **ain**, and **fref** easily from the simulation file to control the input signals.

B. Running the CppSim Simulation

- Within Sue2, Click on **Tools** and then **CppSim Simulation**. The **CppSim Simulation Window** should appear as shown below.



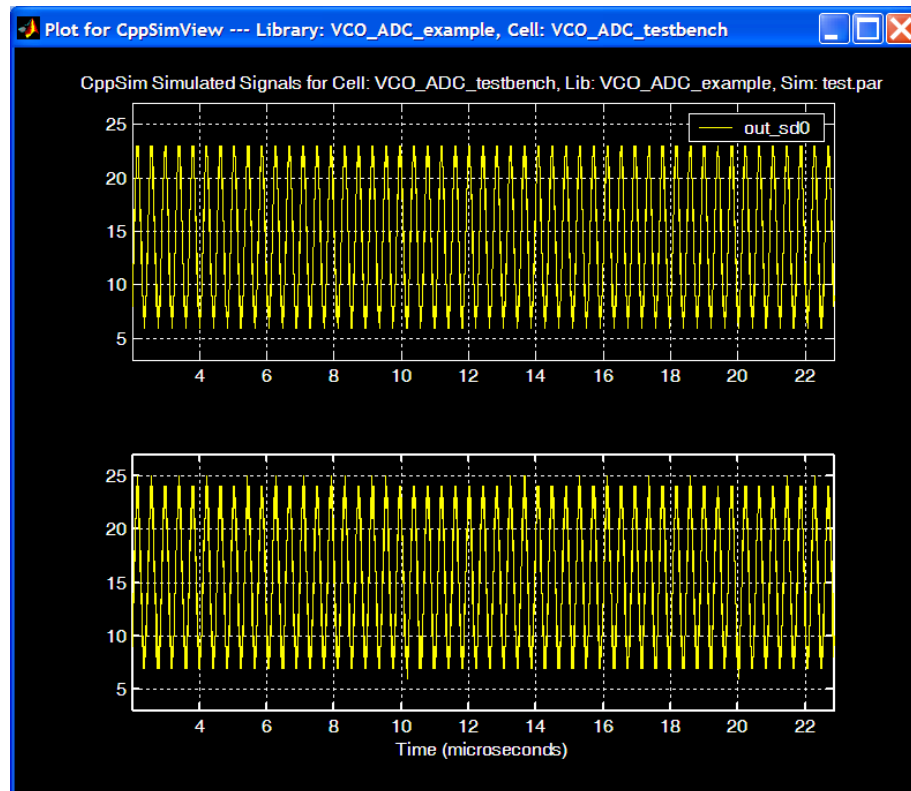
- Double click on the **Edit Sim File** button. An emacs window should appear that indicates that the number of simulation steps, **num_sim_steps**, is set to $8e6$ and the timestep, **Ts**, is set to $1/350e9$. Also note that the output files will be triggered by **ref** and begin at $2e-6$. You can close the emacs window if you like.
- Click on the **Compile/Run** button to launch the simulation.
- Click on the **CppSimView** icon on your desktop to start the CppSim viewer. (You don't have to wait until the simulation is completed!)
- Click on the **No Output File** radio button and select test.tr0 as the output file.
- Click on the **No Nodes** radio button to load in the simulated signals. CppSimView should now appear as shown below.



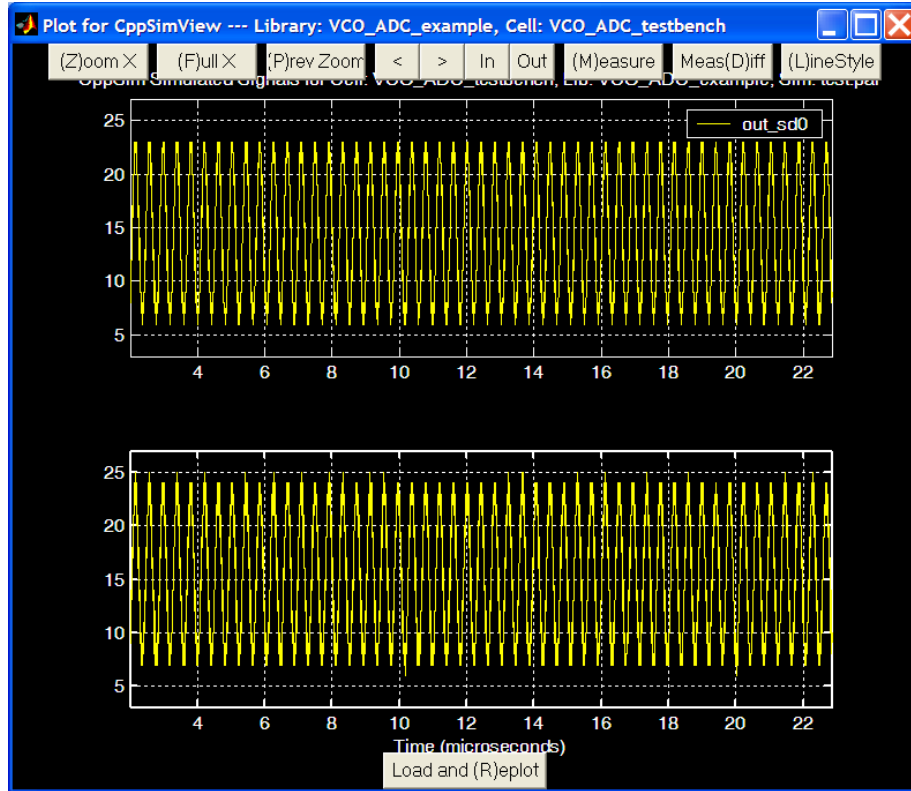
C. Plotting the Time Domain Results

Given the above simulation, we will now take a look at the output signals.

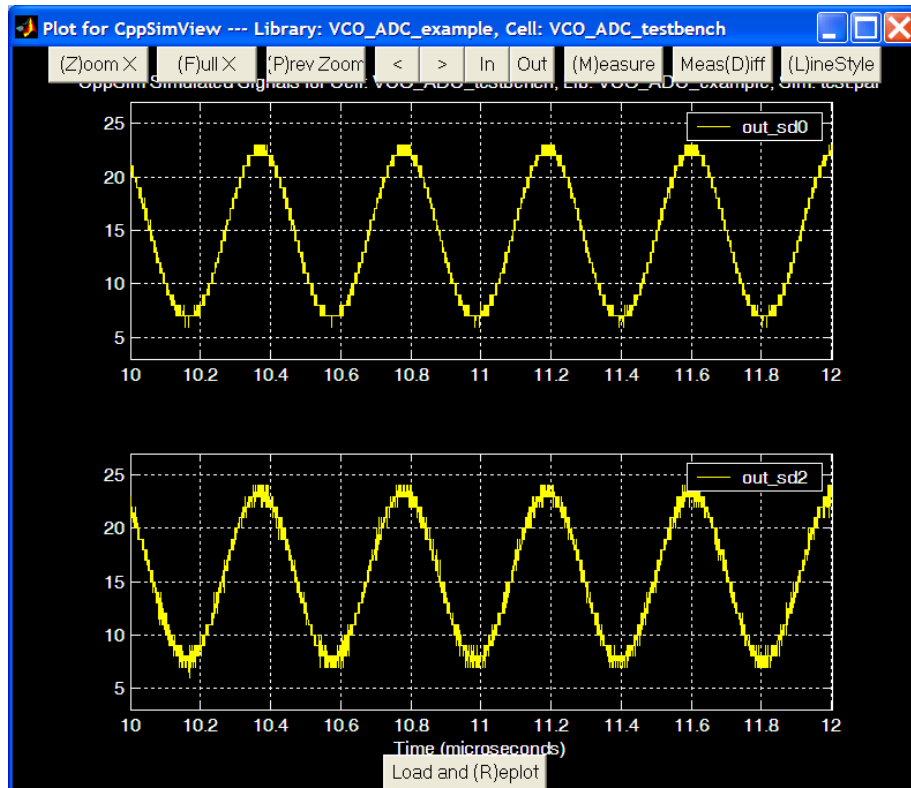
- In the CppSimView window, double-click on signals **out_sd0** and **out_sd0**. You should see the waveforms shown below.



- In the CppSimView window, click on **Zoom**. The waveform window will include buttons as shown below.



- Click on the **Zoom** button in the waveform window, and then zoom in several times in the 10 μ s to 12 μ s range until you observe the waveform below:



- Notice that both outputs appear to have the same amplitude and phase, and it is difficult to judge the relative performance of the two converters. To see the effect of noise-shaping and non-linearity suppression, we need to view the outputs in the frequency domain.

Post-Processing the Simulation Results Using a MATLAB Script

While CppSim has a few built-in functions for frequency domain analysis, in this case we find it convenient to use a customized script in MATLAB. The provided script, **post.m**, also can calculate SNR and SNDR, or other metrics as required by the designer.

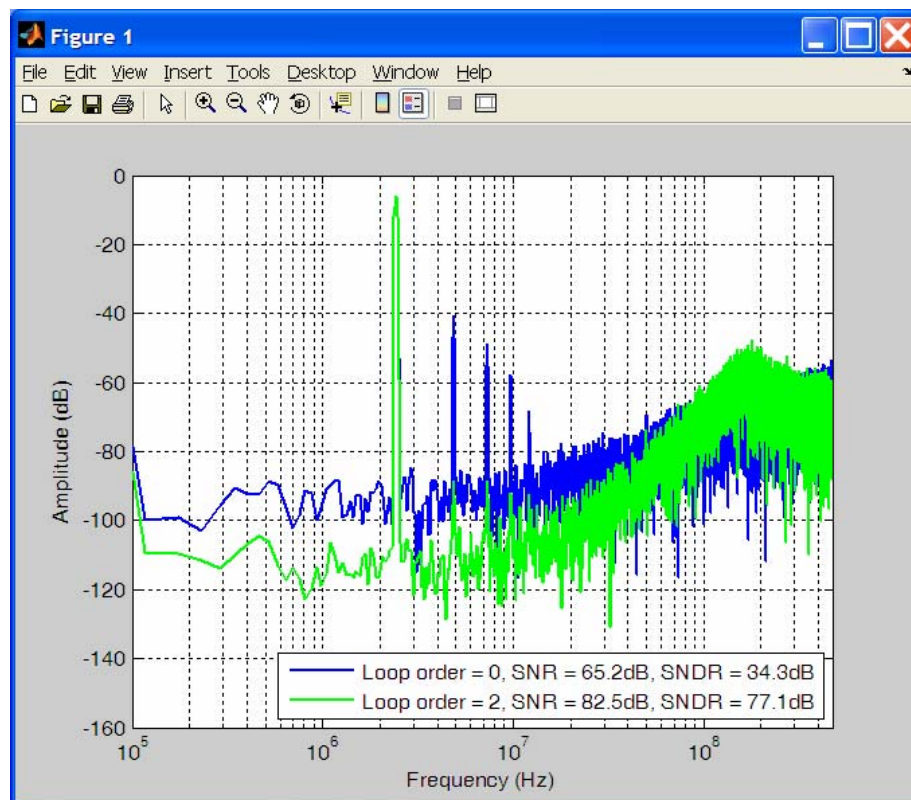
- Open MATLAB and set the working directory to the simulation results folder. For example:

```
>> cd C:\CppSim\SimRuns\VCO_ADC_example\VCO_ADC_testbench
```

- Run the post-processing script post.m by typing post into the command window

```
>> post
```

- After MATLAB completes processing the data, you should see the ADC output spectrum for each of the converters as shown below:



- We observe that the spectrum of the stand-alone VCO-based quantizer does indeed have 1st order noise-shaping, and that the floor is filled in with thermal noise. Additionally, the distortion of the VCO-based quantizer is very high, and limits the SNDR in this case to only 34dB.
- In the frequency domain, we can now see that the 2nd order $\Sigma\Delta$ ADC output is much cleaner than that of the VCO-based quantizer, despite the very similar appearance of the outputs in the time domain. The non-linearity of the VCO-based quantizer has been suppressed here by over 40dB, and the third-order quantization noise-shaping is also evident.

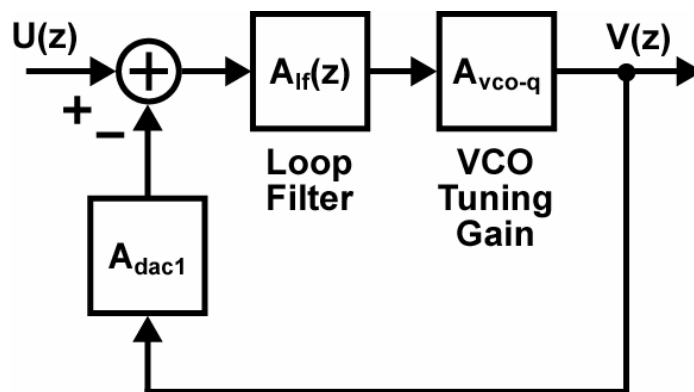
Although the VCO-based quantizer non-linearity has been improved, it is still the dominant source of distortion in the converter. To improve this situation, its linearity can be improved either with circuit design techniques or by increasing the gain of the loop filter. To illustrate the latter option, we now examine a 4th order loop filter, but keep the rest of the ADC the same, including the VCO-based quantizer and feedback DACs.

Design of a 4th Order Loop Filter

In this part of the tutorial, we will explain the design a 4th order loop filter by following along with a MATLAB script. As mentioned in the Introduction, the script uses the Delta Sigma Toolbox which you should download and familiarize yourself in order to fully benefit from this section. The script filename is `loop_filter_calc_params_sd4.m`, and it should be located in the same SimRuns directory as `post.m`. You may open this file now, as we will be referring to it throughout this section.

A. Ideal Noise Transfer Function Design

A simple discrete-time model without excess loop delay is shown below for the VCO-based ADC.



In this figure, the overall open loop gain can be expressed by the product of the three blocks as

$$A(z) = A_{\text{dac1}}A_{\text{lf}}(z)A_{\text{vco-q}}$$

and the noise transfer function (NTF), $H(z)$, can be expressed as

$$H(z) = 1/(1+A(z)).$$

The signal transfer function (STF), $G(z)$, is then

$$G(z) = A_{lf}(z)A_{vco-q}/(1+A(z)).$$

Assuming no excess loop delay, the discrete-time loop filter can easily be designed using the Delta Sigma Toolbox, and the results can be plotted with the following MATLAB commands:

```
% Setup parameters
Fs = 950e6;
BW_analog = 20e6;
R = 25;

% Synthesize ideal NTF given input parameters
H = synthesizENTF(4,R,1); % Fourth order optimized for 20MHz
A = 1/H-1; % Calculate ideal open loop TF for chosen NTF

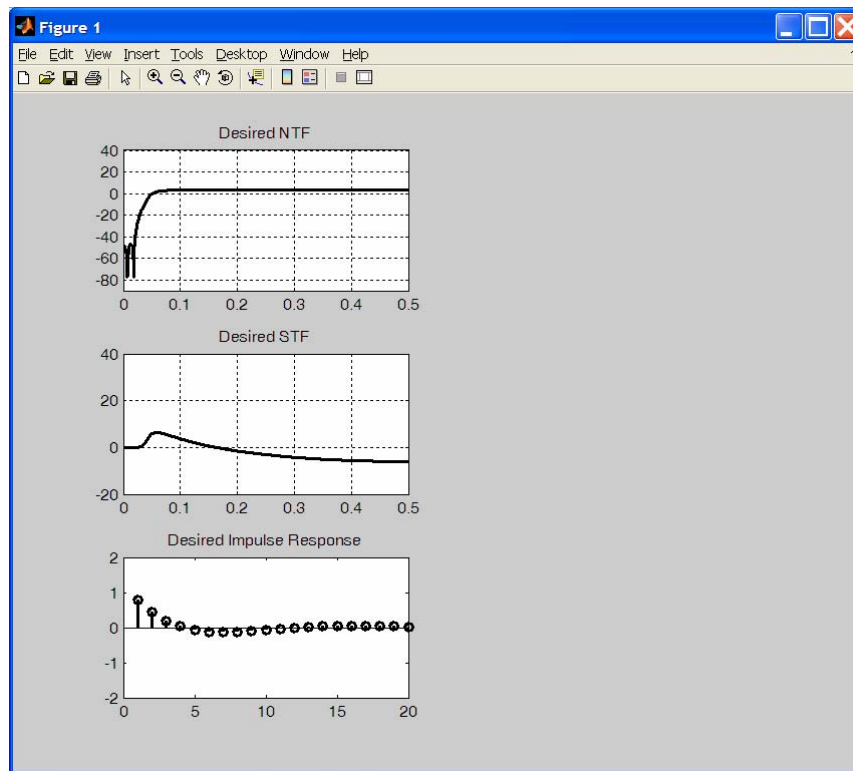
% Plot ideal NTF
f = [linspace(0,0.75/R,100) linspace(0.75/R,1,500)];
s = j*2*pi*f;
z = exp(s);
z_d = filt([0 1],1,1);

figure(1); clf;
subplot(321);
plot(f,dbv(evalTF(H,z)), 'k', 'Linewidth', 2);
axis([0 0.5 -90 40]);
hold on;
grid on;
title('Desired NTF');

% Plot ideal STF in black
subplot(323);
plot(f,dbv(evalTF(A/(1+A),z)), 'k', 'Linewidth', 2);
axis([0 0.5 -20 40]);
hold on;
grid on;
title('Desired STF');

% Plot ideal impulse response in black
subplot(325);
imp_l = 20;
y = impulse(A/(1+A),1:imp_l);
stem(1:imp_l,y, 'k', 'Linewidth', 2);
hold on;
axis([0 imp_l -2 2]);
title('Desired Impulse Response');
```


You can either run the entire script, cut and paste the above text into the MATLAB command window, or if Cell Mode is enabled, press “Shift + Ctrl + Enter” when the cursor is at the top of the file. At this point the plot should look appear like the figure below

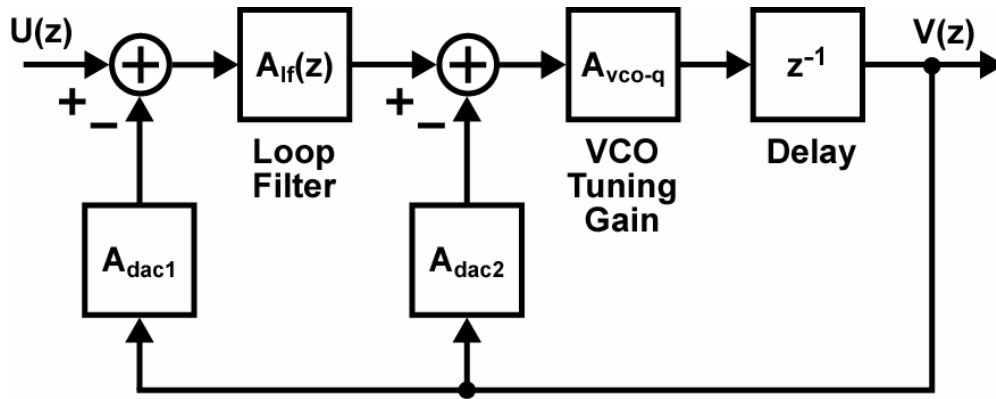


B. Including Excess Loop Delay

In practice, a continuous-time fourth-order loop must take into consideration the excess loop delay, which is found in [1] to typically be about 1.25 clock cycles. For simplicity, here we approximate the loop delay as simply equal to 1 clock period without significant error. To compensate for this delay, a minor compensation loop can be included around the quantizer (as seen in the 2nd order CppSim cell `adc_sd2`). In this case, the additional hardware required is a DAC [3] whose gain can be adjusted appropriately.

[3] S. Yan and E. Sanchez-Sinencio, “A Continuous-Time $\Sigma\Delta$ Modulator With 88-dB Dynamic Range and 1.1-MHz Signal Bandwidth,” IEEE JSSC, Vol. 39, pp. 75–86, January 2004.

Thus, a more complete model for the VCO-based ADC is shown below that includes a single clock delay, as well as the second DAC structure.



Although the *desired* NTF is still the same, we have a new expression for $H(z)$ now given by

$$H(z) = 1/[1 + A_{vco-q}(A_{dac1}A_{lf}(z) + A_{dac2})z^{-1}].$$

The expression for the STF, $G(z)$, is also modified to be

$$G(z) = A_{lf}(z)A_{vco-q}z^{-1}/[1 + A_{vco-q}(A_{dac1}A_{lf}(z) + A_{dac2})z^{-1}].$$

Therefore, we need a way to calculate the new values for $A_{lf}(z)$ and A_{dac2} in order to match both the NTF and STF of the original loop. To do this, we first note that

$$H(z) = 1/[1 + (A(z) + K)z^{-1}],$$

Where $K = A_{dac2}A_{vco-q}$, and $A(z)$ is the open-loop gain as defined earlier. For a given value of K , we can then choose a new $A'(z)$ that results in the desired $H(z)$, as in

$$\begin{aligned} A(z) &= (A'(z) + K)z^{-1}, \\ A'(z) &= A(z)z - K. \end{aligned}$$

The MATLAB script illustrates the effect of K on the resulting signal transfer function by stepping K from 0 to 1 by increments of 0.25.

```
% Now include the excess loop delay, vary the gain of the minor
% loop DAC to compensate for the delay, and look at the effects
% on the STF and the impulse response
k = 0:0.25:1;
k_1 = length(k);
cmap = colormap(jet(k_1));
for i = 1:k_1,
    % Pick a DAC gain
    K = filt(k(i),1,1);

    % Calculate the new discrete-time open loop TF "Ad" that has
    % the same NTF as the ideal "A"
    Ad = A*tf([1 0],1,1)-K;

    % Plot the effect on the NTF
    subplot(321);
```

```

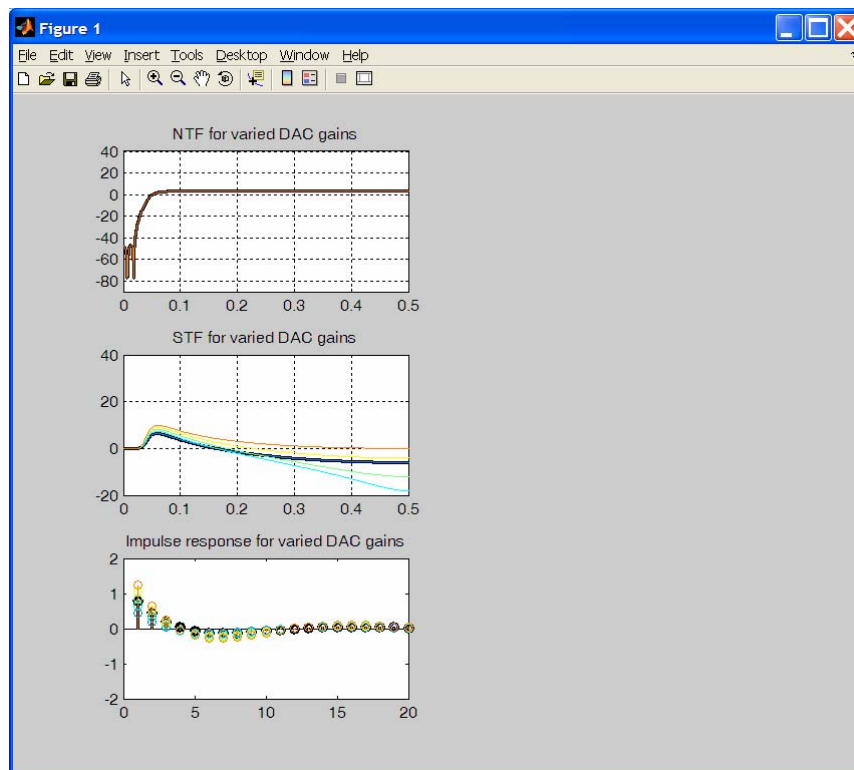
plot(f,dbv(evalTF(1/(1+(Ad+K)*z_d),z)), 'Color',cmap(i,:));
hold on;
grid on;
title('NTF for varied DAC gains');

% Plot the effect on the STF
subplot(323);
plot(f,dbv(evalTF(Ad/(1+(Ad+K)*z_d),z)), 'Color',cmap(i,:));
hold on;
grid on;
title('STF for varied DAC gains');

% and plot the effect on the impulse response
subplot(325);
y = impulse(Ad/(1+(Ad+K)*z_d),1:imp_l);
stem(1:imp_l,y, 'Color',cmap(i,:));
hold on;
title('Impulse response for varied DAC gains');
end

```

Again run the above code using your preferred method, and you should then see the plot now look like



Notice that for each value of K , the NTF is unchanged. However, the STF is clearly altered, as seen both in the frequency domain as well as in the time-domain impulse response. Thankfully, by picking K to be equal to 0.5, the STF is almost identical to the original response.

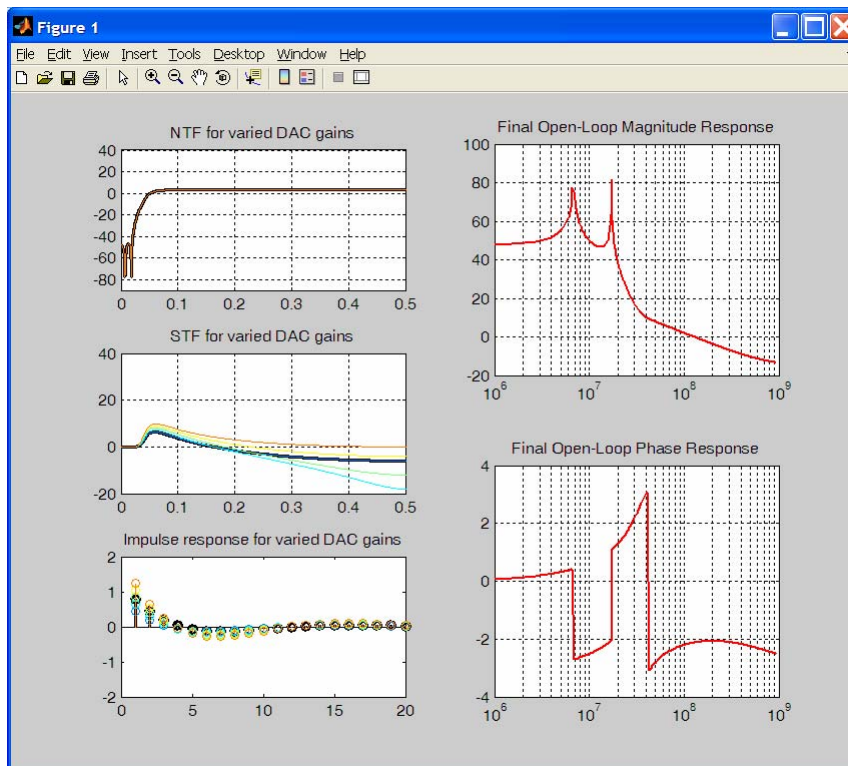
C. Convert from Discrete to Continuous Time

Now we will convert the discrete-time loop filter transfer function into a continuous-time filter, and plot the resulting magnitude and phase response of the open-loop gain. You can run the following section of MATLAB code now.

```
% Based on these results, calculate the final CT open loop TF
k = 0.5;
K = filt(k,1,1);
Ad = A*tf([1 0],1,1)-K; % for making NTF equal to ideal
Ad.Ts = 1/Fs;
Ac = d2c(Ad,'tustin');

% Plot the open and closed loop magnitude response
f = Fs*[linspace(0,0.75/R,100) linspace(0.75/R,1,500)];
subplot(222);
semilogx(f,dbv(squeeze(freqresp(Ac,2*pi*f))), 'r', 'LineWidth',1.5);
grid on;
hold on;
axis([1e6 1e9 -20 100]);
title('Final Open-Loop Magnitude Response');

% Plot the open and closed loop phase response
subplot(224);
semilogx(f,angle(squeeze(freqresp(Ac,2*pi*f))), 'r', 'LineWidth',1.5);
grid on;
hold on;
axis([1e6 1e9 -4 4]);
title('Final Open-Loop Phase Response');
```



D. Calculating Loop Filter Values

Once the continuous-time open-loop transfer function has been established, we can now calculate the parameters for the loop filter based on the assumptions for K_v , DAC gains, etc. Run the last section of MATLAB code now.

```
%% Use the other loop parameters to design the optimal 4th order filter
Kv = 7.35e8; % Hz/V
Avco = 2*pi*Kv/Fs; % rad/sample/V
Adac1 = 640e-6*31/pi; % A/rad/sample
Adac2 = k/Avco; % V/rad/sample
Alf_optimal = Ac/Avco/Adac1;

Alf_optimal = tf(Alf_optimal)
```

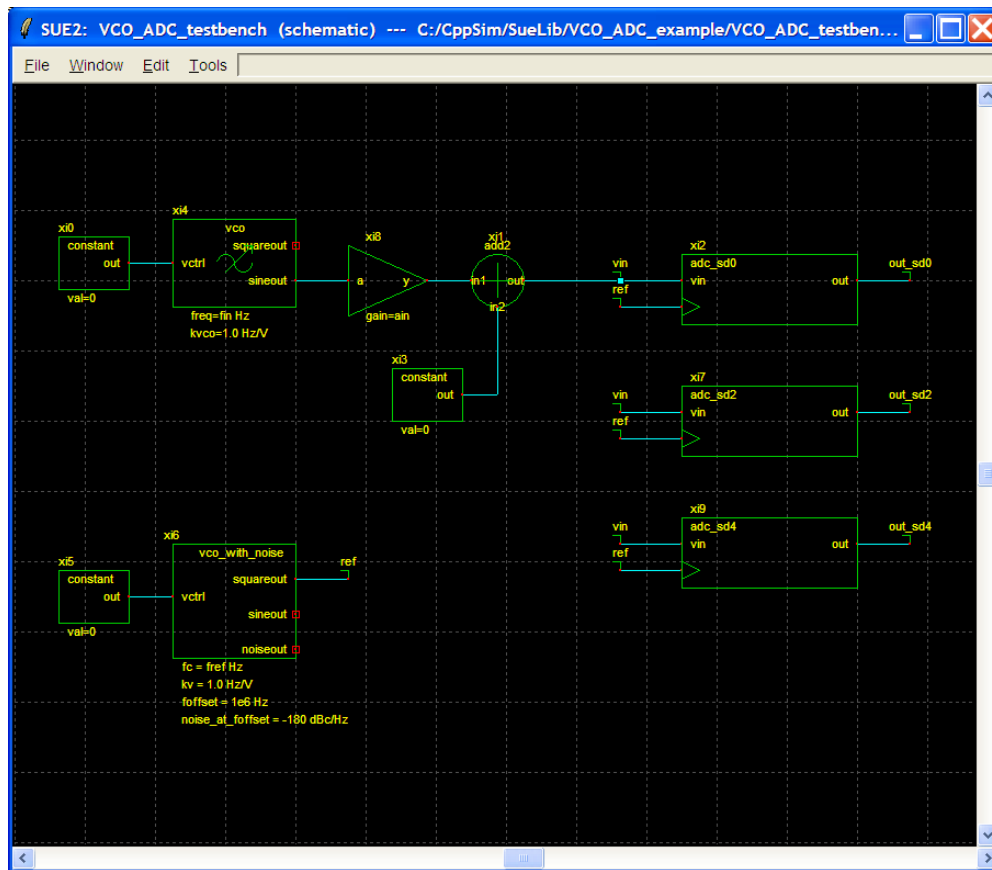
On the MATLAB command window you should see the calculated loop filter values to be

```
Transfer function:
-5.429 s^4 + 2.433e010 s^3 + 7.673e018 s^2 + 1.678e027 s + 1.676e035
-----
s^4 - 2.328e-009 s^3 + 1.224e016 s^2 + 3.751e008 s + 1.745e031
```

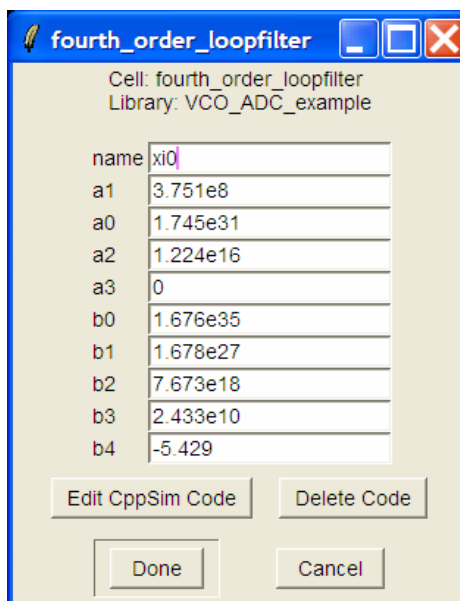
Simulating the 4th Order Loop Filter

After calculating the loop filter coefficients for 4th order dynamics, with NTF zeros optimized for a 20MHz bandwidth, we now proceed to simulate a new ADC with components otherwise identical to the $\Sigma\Delta$ ADC **adc_sd2** simulated earlier.

- Open Sue2 again to the same **VCO_ADC_testbench** that was simulated earlier. We will now place a new 4th order $\Sigma\Delta$ ADC cell into this testbench for comparison against the other two ADC.
- From the **icons1** window, select the **VCO_ADC_example** library from the pulldown menu, and then place **adc_sd4** into the **VCO_ADC_testbench**.
- Define the inputs and output of **adc_sd4** following the convention of the other ADC in the testbench. The output should be named **out_sd4**, the input should be **vin** and the clock should be **ref**. If your schematic is functionally equivalent to the schematic shown below, you should then save the testbench.



- Descend into **adc_sd4** by selecting the cell and pressing “e”. We will now edit the parameters of the loop filter to correspond to the values calculated by the MATLAB script earlier.
- Double-click on the **fourth_order_loopfilter** block and enter in the respective coefficients. The window should look like this before clicking the Done button:



- Save the **adc_sd4** schematic and return to the testbench by pressing “Ctrl + e”. We are now ready to simulate the new ADC.
- Either go to the **CppSim Run Menu** and click on the **Compile/Run** button, or in MATLAB type

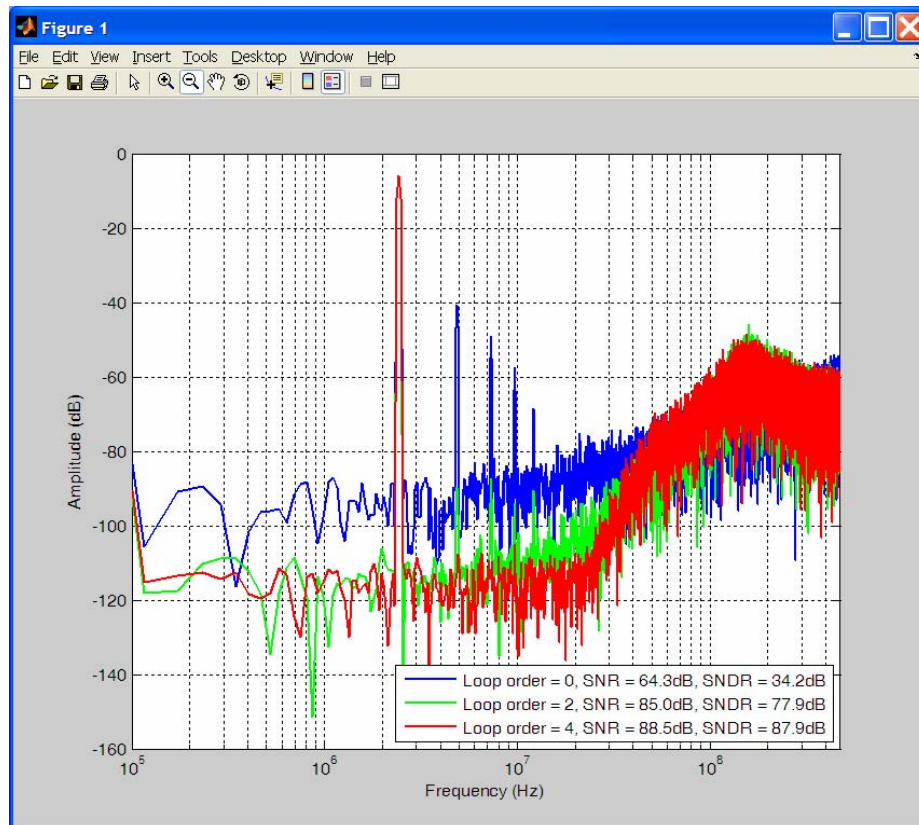
```
>> cppsim
```

from the “SimRuns/VCO_ADC_example/VCO_ADC_testbench/” simulation directory. The simulator should then start, and in this case you should wait a few minutes for the simulation to finish.

- From the MATLAB “SimRuns/VCO_ADC_example/VCO_ADC_testbench/” directory, run `post.m` by typing into the command window:

```
>> post
```

- The post-processing results should generate a new MATLAB figure that includes the simulation results from all three ADC in the testbench. The figure should appear as shown below:



Here, we see that the 4th order loop filter further suppresses the VCO-based quantizer non-linearity, and the SNR/SNDR performance numbers are now very similar.

Conclusion

In this tutorial, we have presented the simulation of a VCO-based quantizer, a 2nd order, and a 4th order $\Sigma\Delta$ ADC. In addition, we have shown that simulating within the MATLAB environment can be very efficient and allow for custom post-processing of CppSim outputs. Finally, we have described an example design flow for calculating the coefficients of a 4th order loop filter using MATLAB scripts.

Although we have not explored issues such as mismatch and metastability, this capability is included in the simulation testbench, and the user is encouraged to play with these parameters to observe the effect on the overall performance.