

# Behavioral Simulation of a Basic GMSK Transceiver using the CppSim Program

Michael H. Perrott  
<http://www.cppsim.com>

July 2008

Copyright © 2004-2008 by Michael H. Perrott  
All rights reserved.

---

## Table of Contents

Setup.....	2
Introduction.....	2
A. Transmitter Architecture .....	2
B. Receiver Architecture .....	4
Performing Basic Operations within Sue2 and CppSimView .....	4
A. Opening Sue2 Schematics .....	4
B. Running the CppSim Simulation .....	6
Plotting Time-Domain Results.....	8
A. I/Q Signal Plots .....	8
B. I/Q Eye Diagrams .....	9
Plotting Frequency Domain Results.....	11
A. Transmitter Output Spectrum (Modulated).....	11
B. Transmitter Output Spectrum (Unmodulated).....	13
Examining Nonidealities .....	16
A. Intersymbol Interference (ISI).....	16
B. Phase Synchronization.....	19
C. Transmitter and Receiver Noise .....	22
Design Overview of the PLL Transmitter.....	23
A. Assumed Parameter Settings .....	24
B. Modeling.....	24
C. Loop Filter Design using the PLL Design Assistant .....	26
Noise Analysis of the Unmodulated PLL using the PLL Design Assistant.....	27
The Need for the gmsk_limitamp Module .....	29
Overview of the I/Q Generator.....	31
A. Implementing Clocking Within Modules.....	32
B. Creating the Gaussian FIR Filter .....	33
Conclusion.....	35

## 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. Please read the manual “**PLL Design Using the PLL Design Assistant Program**”, which is located at <http://www.cppsim.com>, to obtain more information about the PLL Design Assistant.

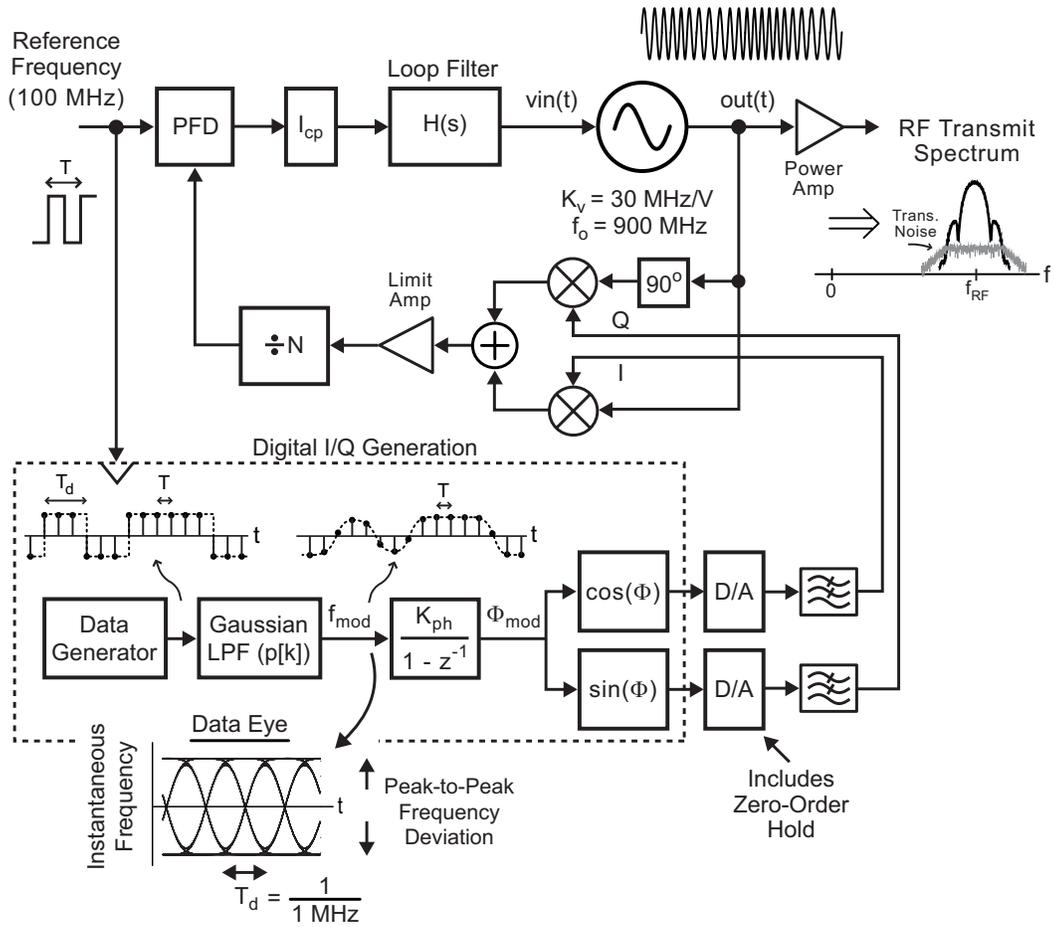
## Introduction

This document will explore the implementation of a system level representation of a GMSK (Gaussian Minimum Shift Keying) transceiver using the CppSim behavioral simulator. GMSK, which is widely used in GSM cell phones, is a modulation method that encodes data information by phase modulating a constant envelope sine wave signal. By constraining the transmitter output to have constant envelope, a nonlinear power amplifier can be used that broadcasts the signal with high efficiency (usually around 50% efficiency, as compared to about 10% efficiency encountered with linear power amps).

In this document, we will also briefly mention the similarities of GSMK modulation to GFSK (Gaussian Frequency Shift Keying) modulation. This latter standard is often employed in cordless phones according to specifications such as DECT.

### **A. Transmitter Architecture**

To explore GMSK modulation, we will focus on the transmitter structure shown in Figure 1, which employs a PLL with an I/Q mixer in its feedback loop. The function of the I/Q mixer is to vary the phase at the transmitter output according to I/Q signals generated by an I/Q generator block. The I/Q generator creates this signals by first smoothing a binary data stream with a Gaussian transmit filter and then accumulating the resulting frequency modulation signal,  $f_{mod}$ , to form a modulated phase signal,  $\Phi_{mod}$ . The modulated phase signal is then used as the argument for sine and cosine functions in order to create the I/Q signals.



**Figure 1: A GMSK modulator implemented as an offset PLL.**

Smoothing by the Gaussian transmit filter has the effect of reducing the spectral content of the transmitted signal. Given a sample period of  $T$  and symbol period of  $T_d$ , as shown in the figure, the discrete-time impulse response of the transmit filter,  $p[k]$ , is formulated as

$$p[k] = \frac{h}{2} \frac{1}{\sqrt{\pi\sigma}} e^{-\left(\frac{k}{\sigma}\right)^2}$$

where

$$\sigma = \frac{1.178 T_d}{BT_d 2\pi T}$$

The above impulse response is influenced by two key parameters:

- $BT_d$ : the ratio of the bandwidth of the Gaussian filter,  $B$ , to the symbol rate,  $1/T_d$ 
  - For GMSK systems,  $BT_d$  is often set to 0.3
  - For GFSK systems,  $BT_d$  is often set to 0.5
- $h$ : the modulation index, which is defined as the ratio of peak-to-peak frequency deviation to the symbol rate
  - For GMSK systems,  $h$  must be precisely set to 0.5
  - For GFSK systems,  $h$  can be loosely set to 0.5 +/- 0.05

GMSK systems achieve much better spectral efficiency than their GFSK counterparts due to their lower  $BT_d$  product, but have the disadvantage of requiring very accurate modulation control in order to achieve a precise setting of  $h$ .

In this document, we will explore the GMSK transmitter structure by examining its implementation in CppSim, running simulations, and examining signals in the form of eye diagrams and power spectra.

## B. Receiver Architecture

Figure 2 displays a direct conversion receiver that will be used to demodulate the desired GMSK RF signal. For simplicity, we will ignore the many practical difficulties encountered with this structure, such as high sensitivity to DC offsets, antennae impedance variations, and local oscillator feed-through. The noise source in the figure represents the overall input-referred noise of the receiver, which can be used to examine the influence of receiver noise on the overall signal-to-noise ratio of the receiver.

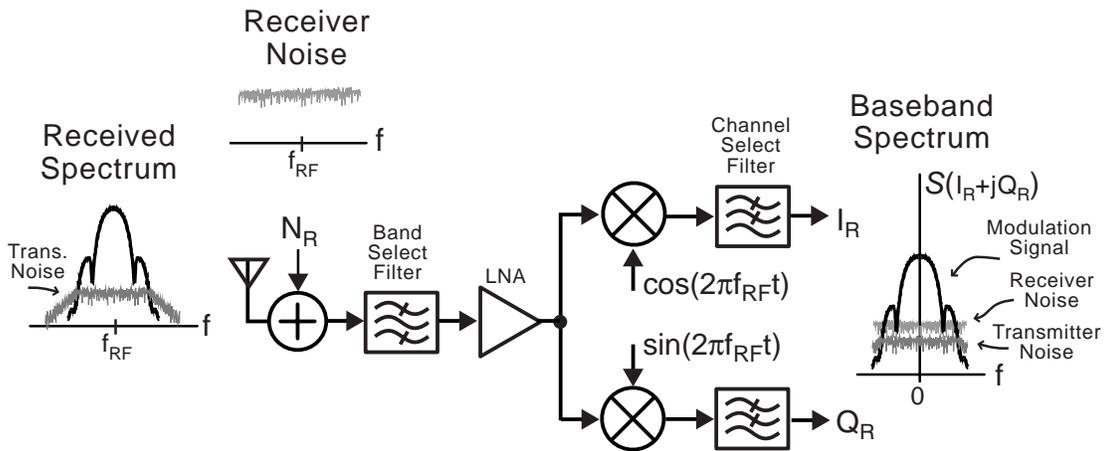


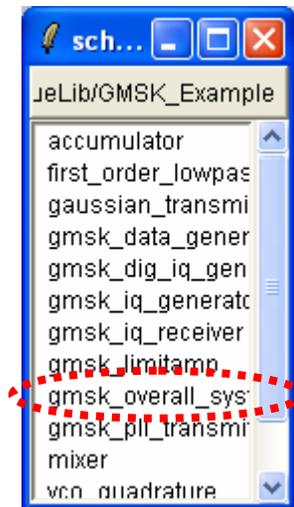
Figure 2: A direct conversion receiver for GMSK detection.

## Performing Basic Operations within Sue2 and CppSimView

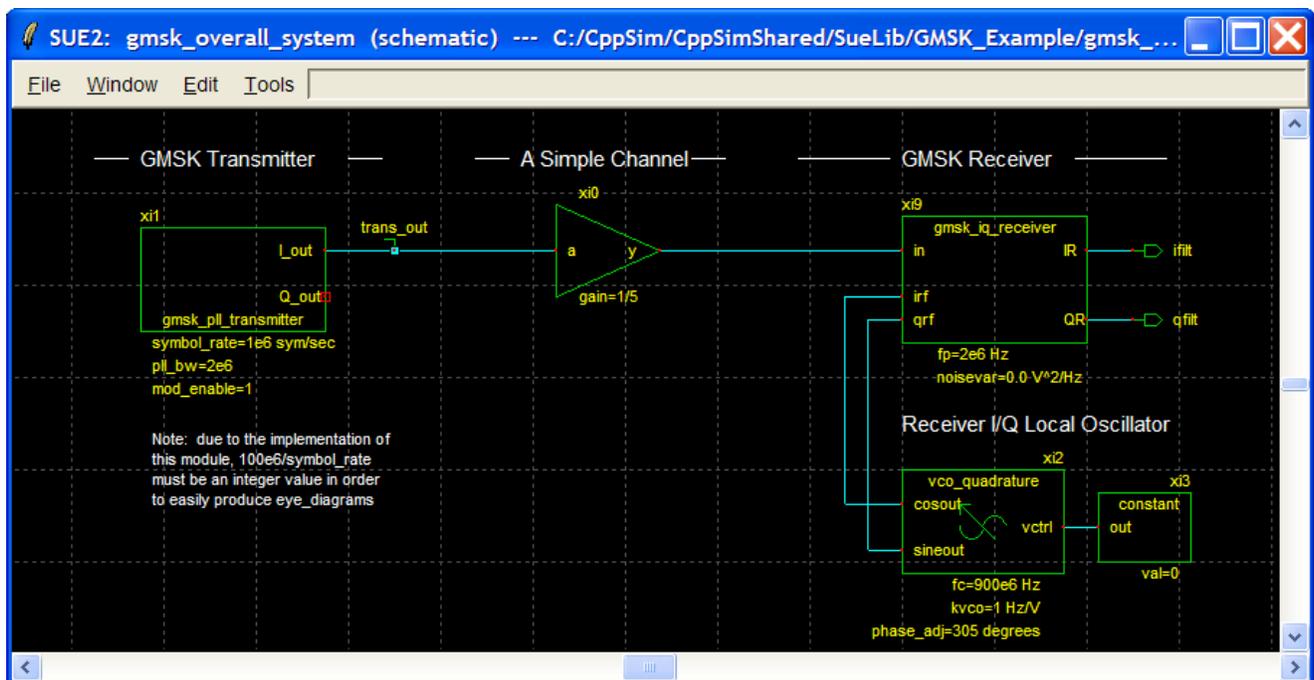
In this section, the user will be guided through basic tasks such as opening the GMSK example system within the Sue2 schematic editor and running CppSim simulations.

### A. Opening Sue2 Schematics

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

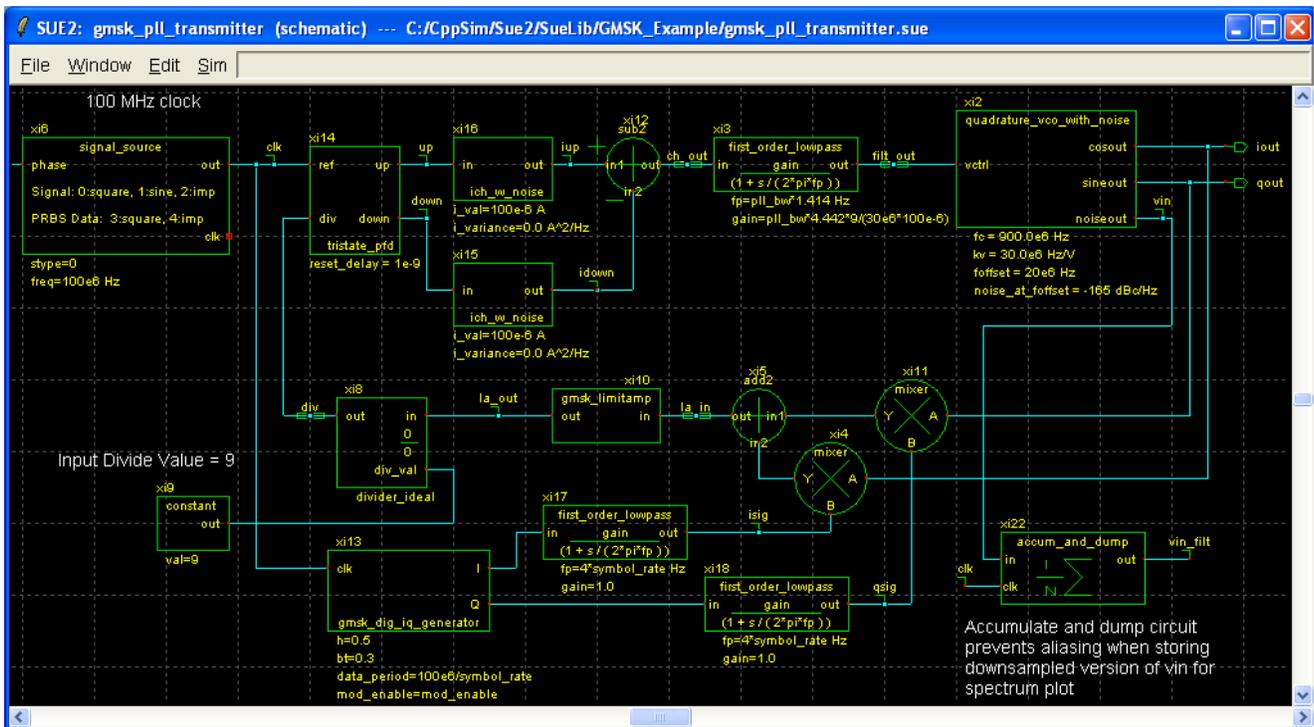


- Select the **gmsk\_overall\_system** cell from the above **schematic listbox**. The Sue2 schematic window should now appear as shown below. Key signals and parameters for this schematic include
  - **ifilt** and **qfilt**: correspond to the received I/Q signals which will be used to generate eye diagrams
  - **phase\_adj**: corresponds to the empirically derived optimal phase value for demodulating the I/Q signals. We will later explore the impact of changing this phase value.



- Select the **gmsk\_pll\_transmitter** icon within the above schematic, and then press **e** to descend down into its associated schematic. You should now see the schematic shown below. Some key signals in the transmitter schematic include
  - **vin\_filt**: corresponds to the input voltage of the VCO scaled by  $K_v$  (the VCO gain), which will be used to plot the output spectrum of the transmitter

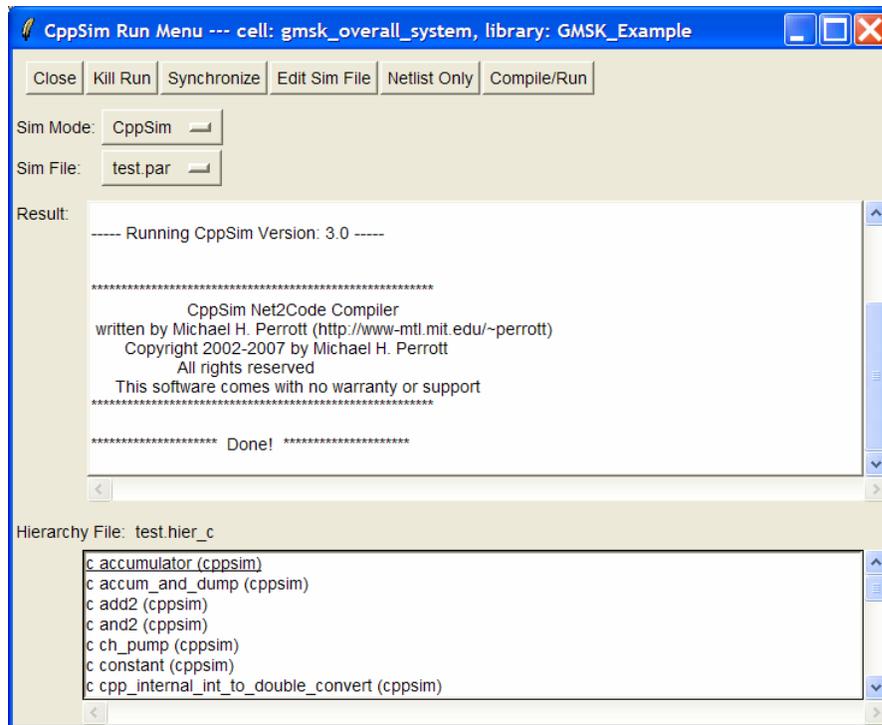
- **isig** and **qsig**: correspond to the ideal I/Q signals that we desire to transmit, which will be used to create ideal I/Q eye diagrams to allow comparison to the received ones within the **gmsk\_overall\_system** cell.



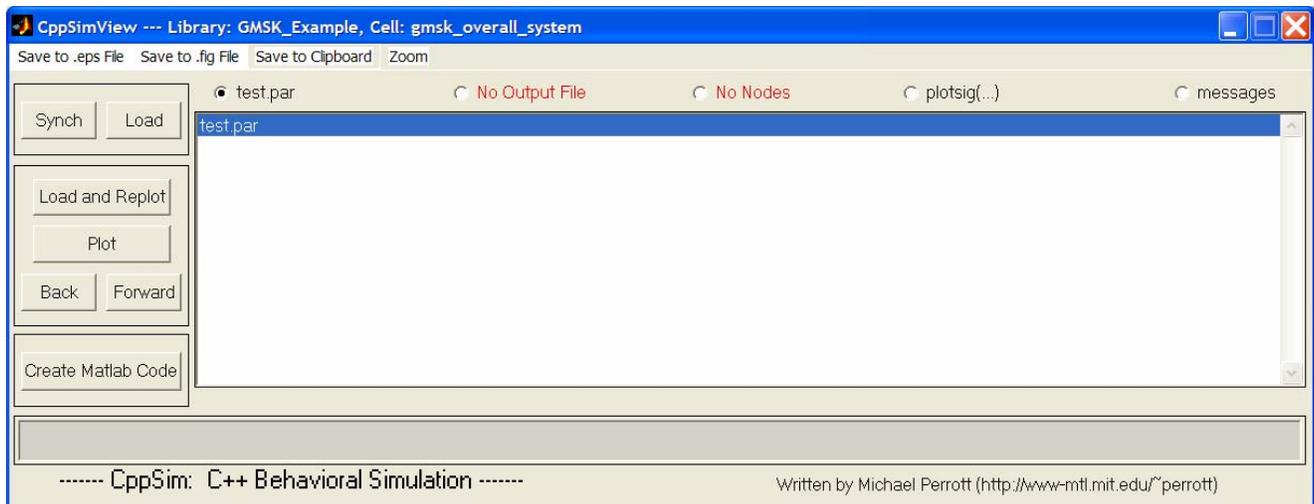
- Press **Ctrl-e** to return to the **gmsk\_overall\_system** cellview.

## B. Running the CppSim Simulation

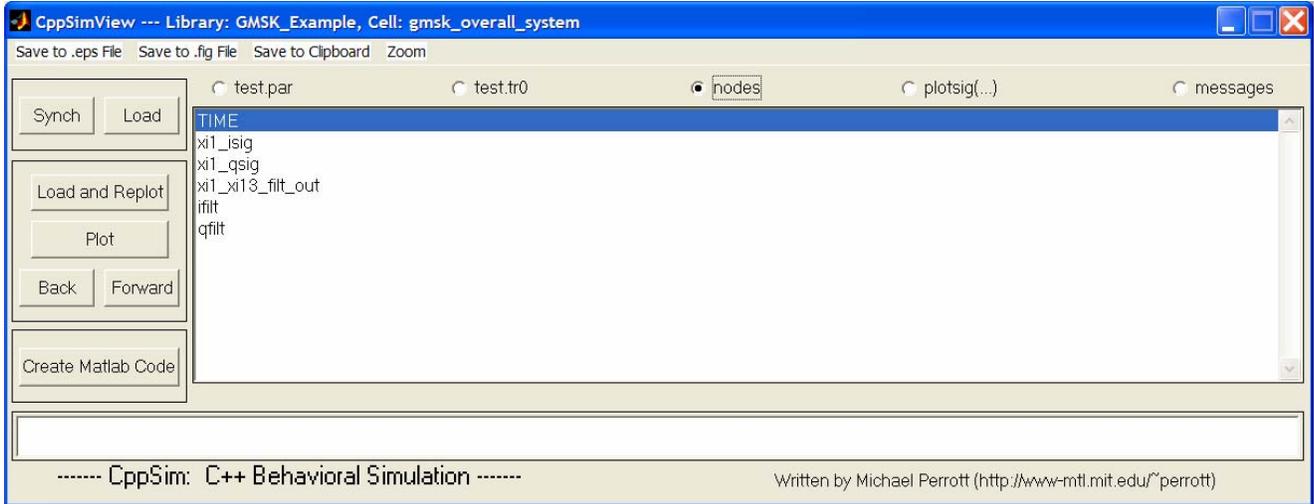
- Within the Sue2 window, select **Tools** → **CppSim Simulation**. You should see the Cppsim Run Menu that pops up. Left click on the **Edit Sim File** button, and an Emacs window should pop up.



- Now open CppSimView and make sure it synchronizes to your new schematic. If not, left-click the **Synch** button on the CppSimView window to do it.



- Go back to the CppSim Run Menu. Double-click on Edit Sim File – an Emacs window should appear that indicates that **num\_sim\_steps** is set to 1e6 and **Ts** is set to 1/9e9. You may then close the Emacs window if you like.
- Click on the **Compile/Run** button to run the simulation.
- 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

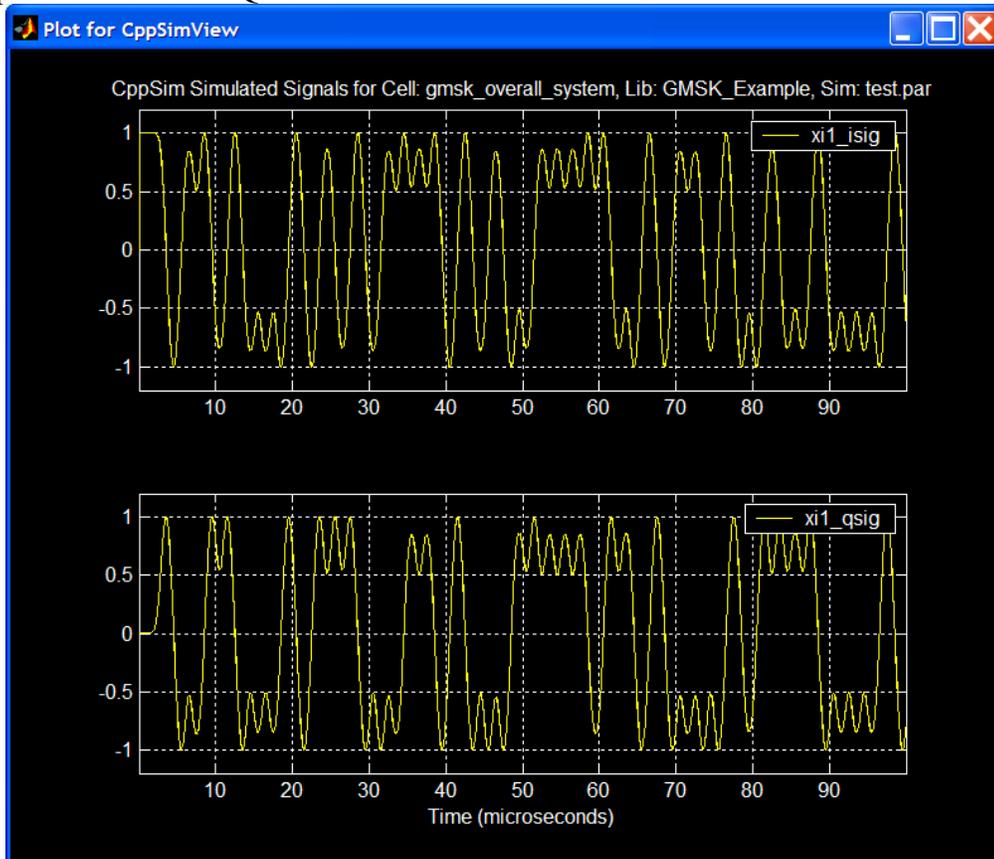


## Plotting Time-Domain Results

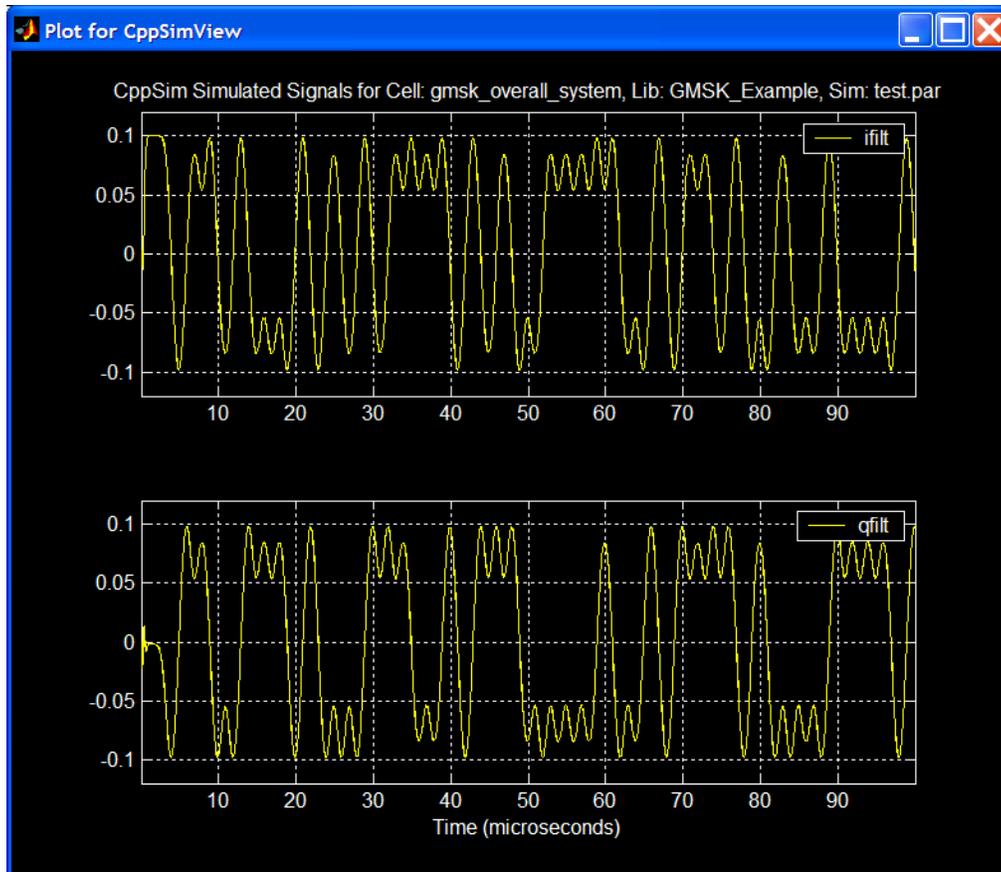
Given the completion of the above simulation, let us now examine the resulting I/Q signals in the time-domain in the form of signal plots and eye diagrams.

### A. I/Q Signal Plots

- In the CppSimView window, double-click on signals **xi1\_ising** and then **xi1\_qsig**. You should see plots of the ideal I/Q waveforms as shown below



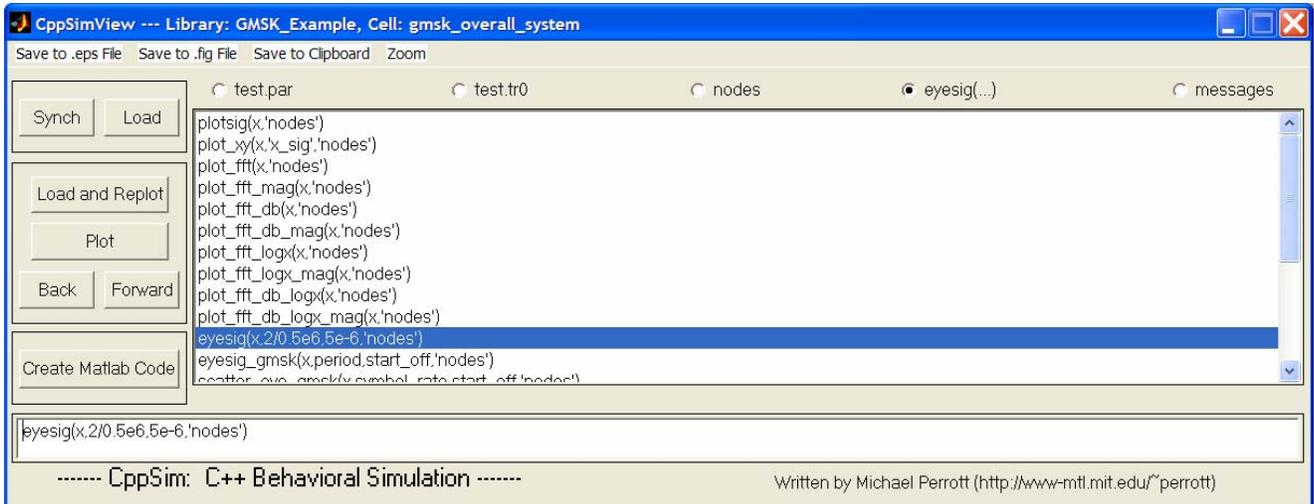
- Now click on the **Reset Node List** button, and then double-click on signals **ifilt** and **qfilt**. You should see plots of the received I/Q signals as shown below.



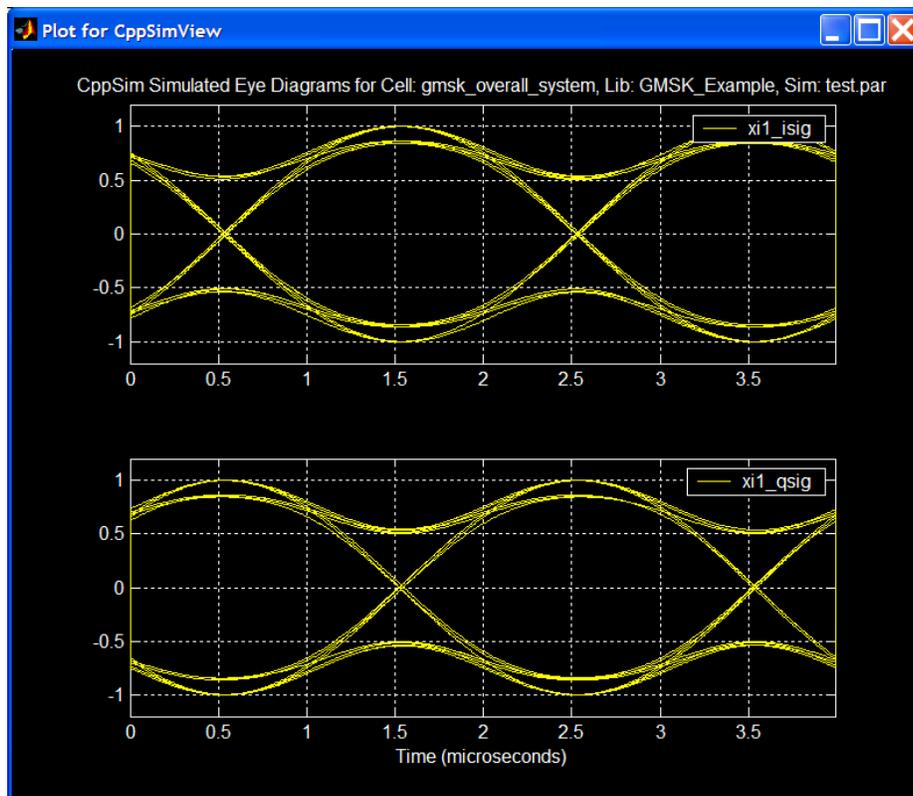
We see that there is pretty good agreement between the ideal and received waveforms, the only real difference being their relative magnitude. The received I/Q waveforms are a factor of 10 lower in magnitude due to the fact that the transmitter output was scaled by 1/5 in the ‘simple channel block’ within the **gmsk\_overall\_system** cellview and that demodulation by unit amplitude cosine/sine waveforms leads to another factor of 1/2 in scaling of the received signal.

## B. I/Q Eye Diagrams

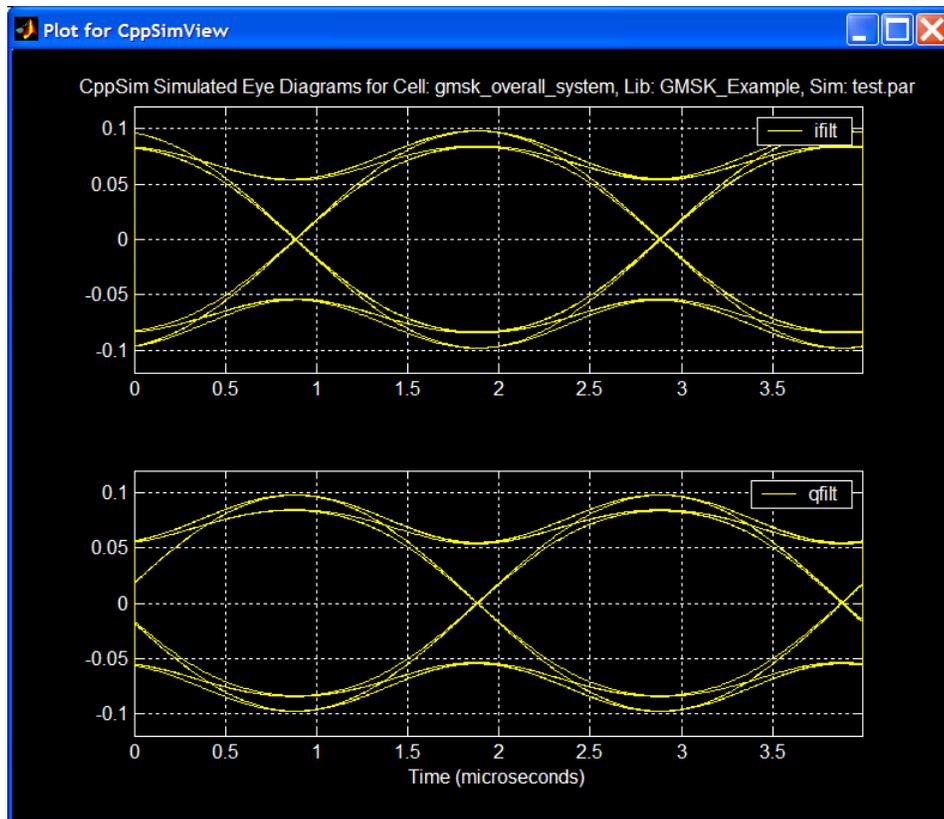
- Click on the **plotsig(...)** radio button in CppSimView and then select the **eyesig(...)** function. Set **period** to be 2/0.5e6 (i.e., two symbols long for the I/Q waveforms – note that the I/Q waveforms have twice the symbol period as the instantaneous frequency waveform) and **start\_off** to be 5e-6 (i.e., 5 microseconds, which is long enough for the transmitter PLL to reasonably settle). Hit **Return** to enter the function into the CppSimView function list. CppSimView should now appear as shown below.



- Click on the **nodes** radio button, and then double-click on signal **xi1\_isig** and then on signal **xi1\_qsig**. The eye diagrams of the ideal I/Q signals should appear as shown below.



- Now click on the **Reset Node List** button, and then double-click on signals **ifilt** and **qfilt**. The received I/Q eye diagrams should appear as shown below.



Given the above eye diagrams, we see that the received waveforms look very close to ideal!

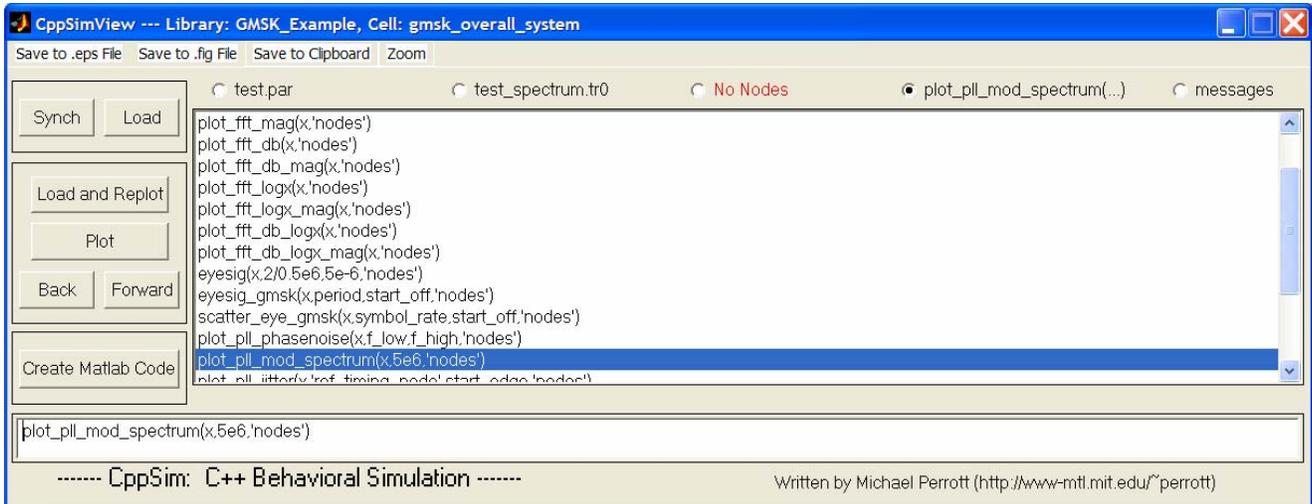
## Plotting Frequency Domain Results

We will now examine both the modulated and unmodulated output spectrum of the transmitter.

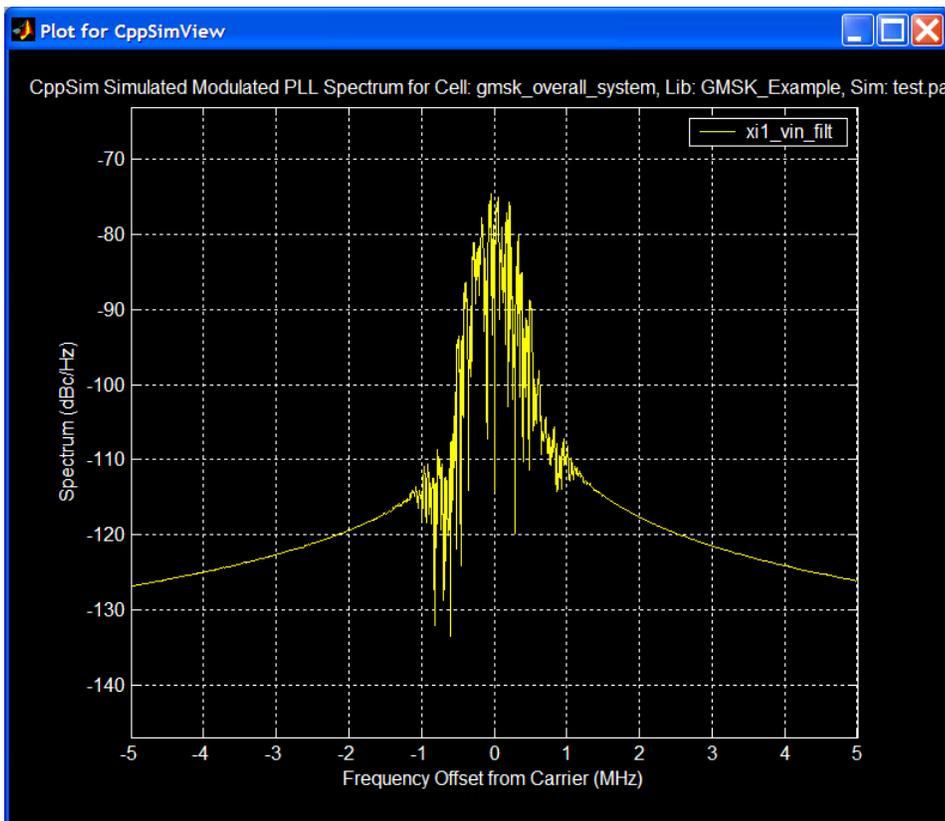
### **A. Transmitter Output Spectrum (Modulated)**

Now perform the following operations in CppSimView:

- Click on the **test.tr0** radio button and then choose the output file to be **test\_spectrum.tr0**.
- Click on the **eyesig(...)** radio button and then choose the plotting function to be **plot\_pll\_mod\_spectrum(...)**. Set the **fspan** parameter to  $5e6$  (i.e., to examine the output spectrum  $\pm 5$  MHz about the carrier frequency), and then hit **Return**. CppSimView should appear as shown below.

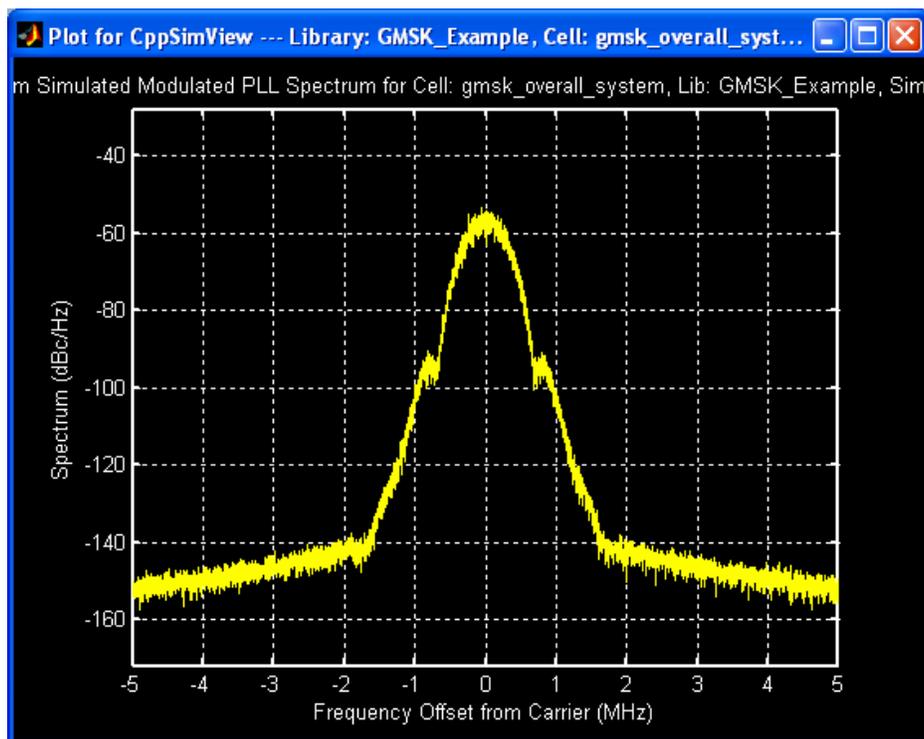


- Click on the **No Nodes** radio button, and then double-click on **xi1\_vin\_filt**. The resulting modulated output spectrum should appear as shown below.
  - Unfortunately, this spectrum does not resemble what we would expect to see for a GMSK modulator. The problem is that we have too few simulation points.



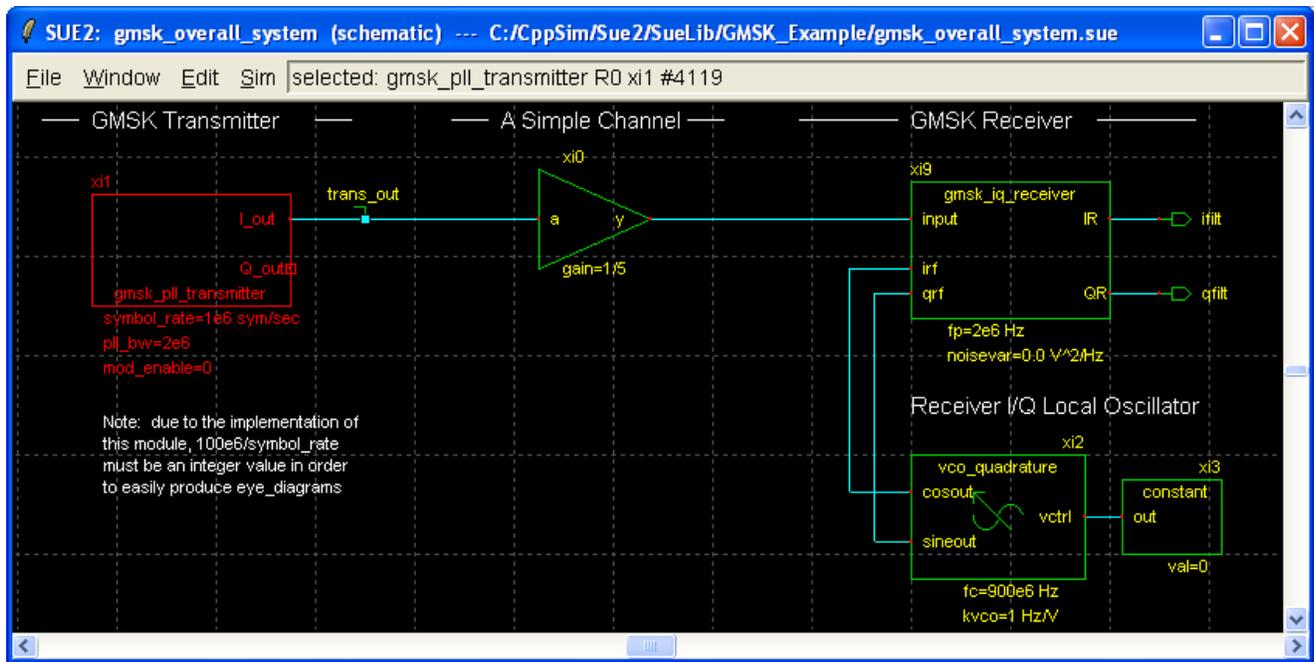
- To get a better spectrum plot, we need to increase the number of simulation points. To do so, click on the **Edit Sim File** button in CppSim Run Menu, change the **num\_sim\_steps** parameter from 1e6 to 50e6 within the Emacs window that pops up, and then save the file. You may close Emacs when you complete this task.

- Click on the **Compile/Run** button in CppSimView Run Menu to start the CppSim simulation. Note that this simulation takes about 10 minutes on a 2.6 GHz Pentium 4 machine with 1 Gig of DRAM, and about the same on a 1.3 GHz Pentium M laptop with 1 Gig of DRAM. If the simulation appears to be taking too long on your machine, you can end it by hitting **Ctrl-Alt-Delete**, selecting the **Processes** tab, and then ending the **test.exe** process. An alternative way to stop the simulation is to click on the **Kill Run** button in CppSimView Run Menu.
  - Note that you can also descend down into the **gmsk\_pll\_transmitter** icon within Sue2, click on the **Tools->CppSim Simulation**, and then click on the **Compile/Run CppSim** button within CppSimView Run Menu. By running the **gmsk\_pll\_transmitter** on its own (i.e., without the receiver), the simulation will run nearly twice as fast. Once the run is completed, you would again select the **plot\_pll\_mod\_spectrum(...)** plotting function and then select signal **vin\_filt** as its argument.
- Assuming that you ran the above simulation within cell **gmsk\_overall\_system**, click on the **No Nodes** button in CppSimView and then double-click on the **xi1\_vin\_filt** signal. You should see the plot shown below.

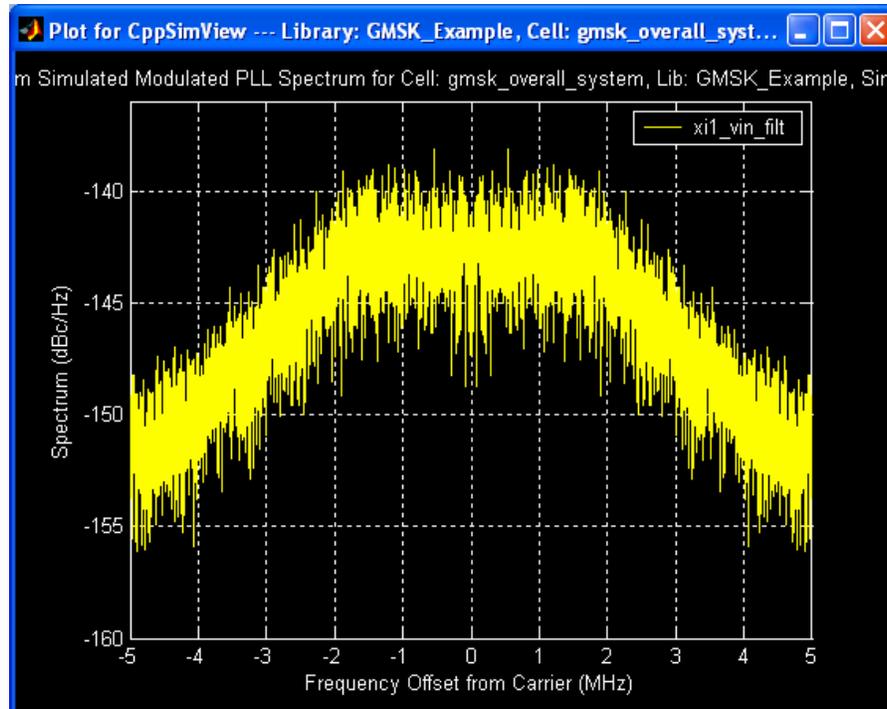


## B. Transmitter Output Spectrum (Unmodulated)

- Within the **gmsk\_overall\_system** schematic of Sue2, double-click on the **gmsk\_pll\_transmitter** icon and then change the **mod\_enable** parameter from 1 to 0. Save the schematic. The Sue2 schematic window should now appear as follows.



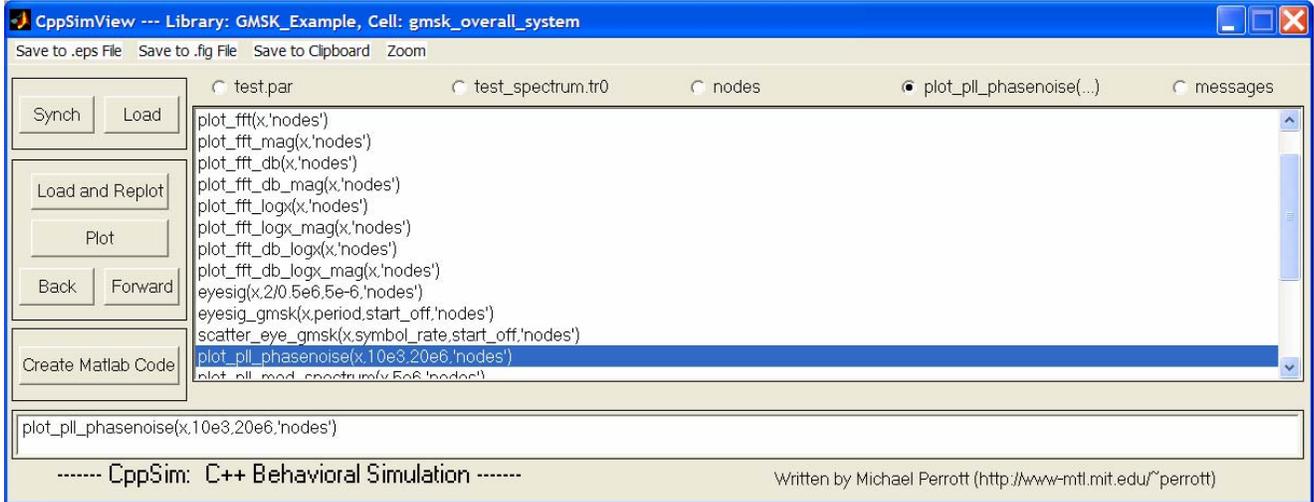
- Click on the **Compile/Run** button within CppSimView Run Menu to run the simulation. Note that this will again take some time to run.
- Click on the **No Nodes** button in CppSimView, and then double-click on signal `xi1_vin_filt`. You should see the output spectrum plot as show below.



For the case of an unmodulated sine wave, we often look at  $L(f)$  (which is the spectrum of the *phase* of the sine wave rather than the sine wave itself) instead of the output spectrum of the sine wave. It turns out that  $L(f)$  and the output spectrum are essentially the same when the sine wave is unmodulated (i.e., when the phase deviations are small). However, the CppSimView function that plots  $L(f)$  has a more

convenient display format for looking at phase noise, so we'll now proceed with plotting  $L(f)$  rather than the output spectrum show above.

- CppSimView, click on the **plot\_pll\_mod\_spectrum(...)** radio button, and then select the **plot\_pll\_phasenoise(...)** function. Change **f\_low** to 10e3 and **f\_high** to 20e6 and then push **Return** to save these values within the function list. CppSimView should appear as shown below.



- Now click on the **nodes** radio button within CppSimView, and then double-click on signal **xi1\_vin\_filt**. You should see a plot of  $L(f)$  as shown below.

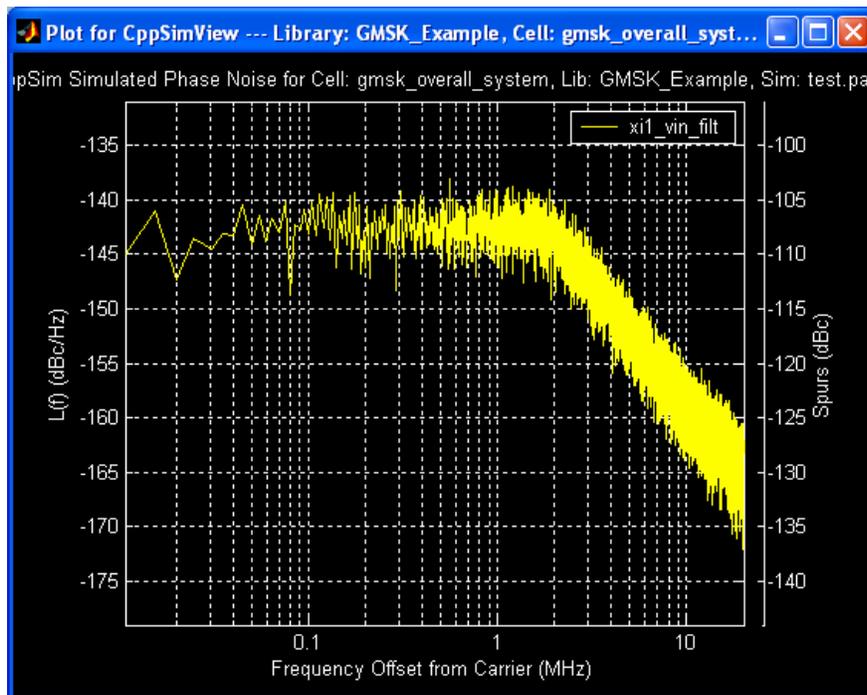


Figure 3: Simulated phase noise of the unmodulated PLL transmitter.

- Note that it would *not* be appropriate to plot  $L(f)$  to look at the *modulated* output. For  $L(f)$  to be meaningful, changes in the phase of the output sine wave need to be small.
- Also note that you are limited in how low of a frequency offset you can examine. You would need to increase the number of simulation points to lower the frequency offset range that you can observe.
- Finally, note that we will compare the above phase noise plot to calculated results in the PLL Design section of this document.

For those further interested in understanding the differences between the `plot_pll_mod_spectrum(...)` and `plot_pll_phasenoise(...)` functions, consider looking at the corresponding Matlab files `comp_psd_mod.m` and `comp_psd.m` within the directory `c:\CppSim\CppSimShared\MatlabCode` (where `c:\CppSim` should be replaced with the directory location that you specified when installing CppSim). These files assume that you are looking directly at the input voltage to the VCO (*after* the point at which any input-referred VCO noise sources are added), and therefore require  $K_v$  to be appropriately set (note that  $K_v=30e6$  in this case).

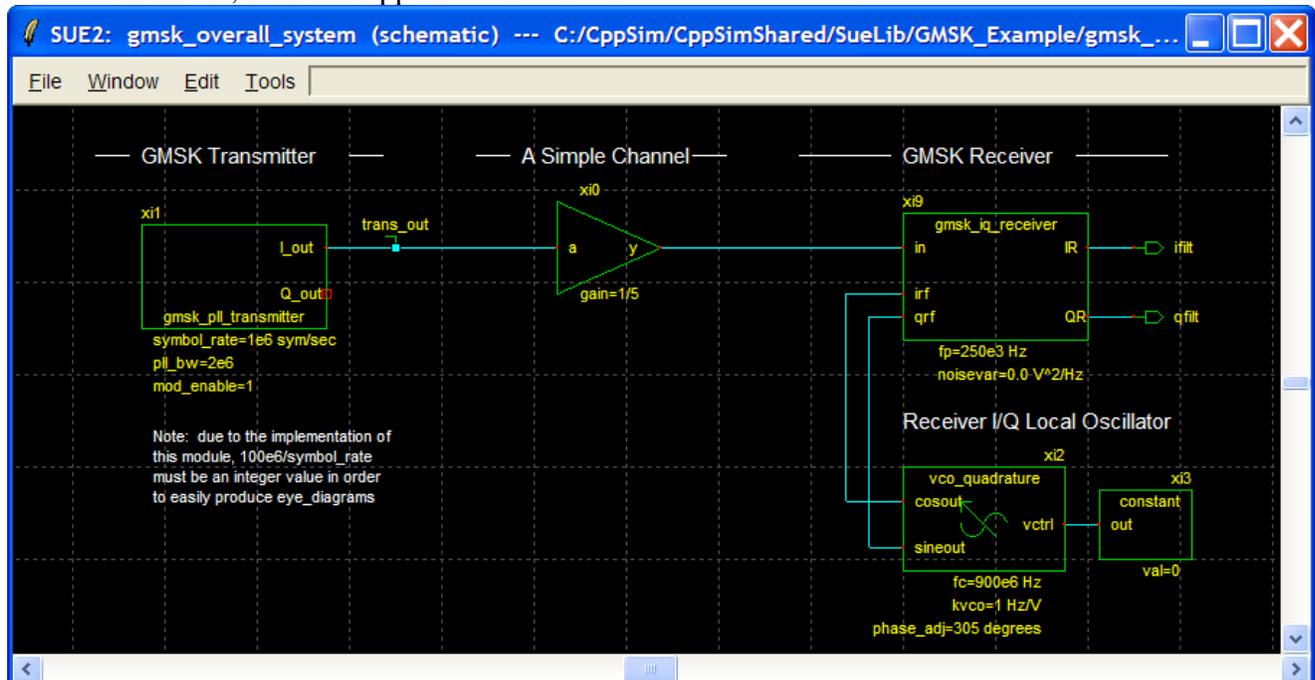
## Examining Nonidealities

In this section, we will briefly explore the impact of nonidealities such as intersymbol interference, poor phase synchronization, and noise.

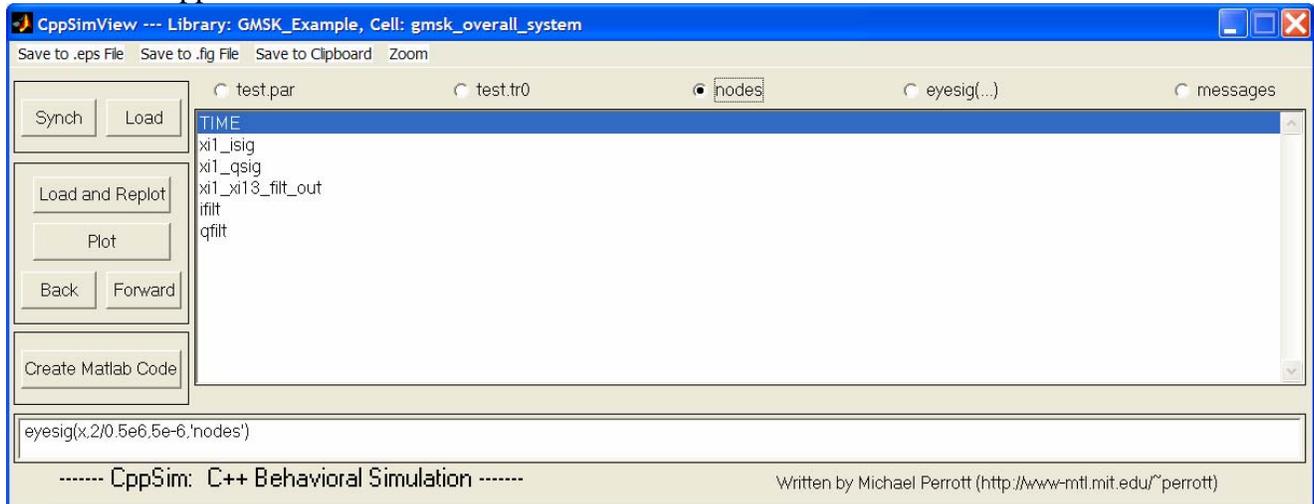
### A. Intersymbol Interference (ISI)

Let us first examine the influence of receiver-induced intersymbol interference.

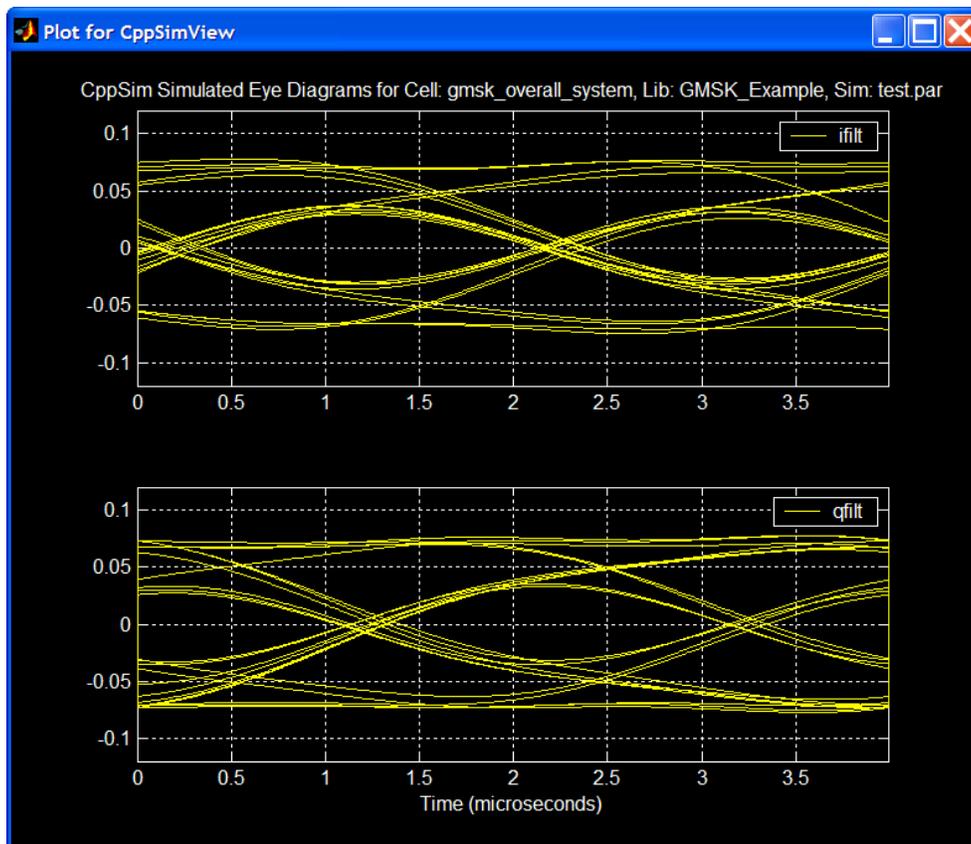
- Double-click on the `gmsk_pll_transmitter` icon within the Sue2 schematic window and set `mod_enable` to 1.
- Double-click on the `gmsk_iq_receiver` icon and set `fp` to 250e3. Upon saving the Sue2 schematic, it should appear as shown below.



- Click on the **Edit Sim File** button within CppSim Run Menu and then set **num\_sim\_steps** to 1e6 within the Emacs window that pops up. Save the file and then exit Emacs.
- Click on the **Compile/Run** button to run the CppSim simulation.
- Click on the **test\_spectrum.tr0** radio button and change the output file to **test.tr0**.
- Click on the **plot\_pll\_phase\_noise(...)** radio button and change the plotting function to be **eyesig(...)**.
- Click on the **No Nodes** radio button to load in the simulation signals. CppSimView should now appear as shown below:



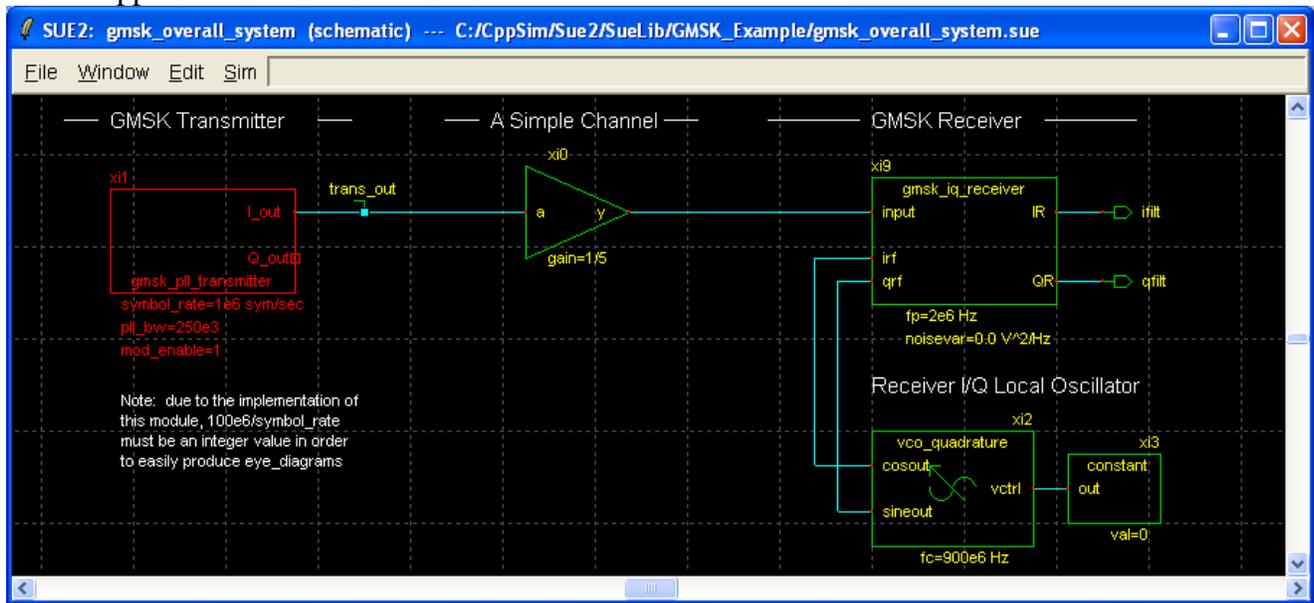
- Double-click on the ifilt and qfilt signals to plot their corresponding eye diagrams, which should appear as shown below.



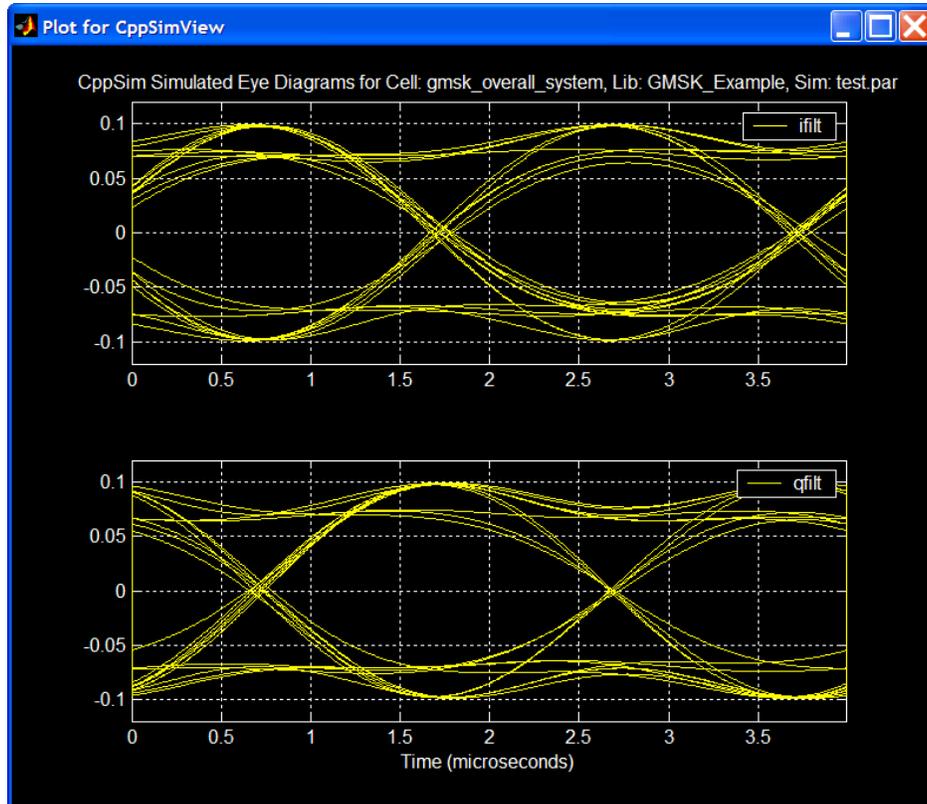
- Comparison of the above eye diagrams to previous ones plotted with 2 MHz receiver bandwidth reveals the impact of intersymbol interference that occurs due the reduction of the receiver **fp** parameter to 250 kHz. You might consider setting the **fp** parameter of the receiver to different values and observe the corresponding impact on the received eye diagrams.

Transmitter-induced intersymbol interference can be examined in similar fashion.

- Within Sue2, set the **fp** parameter of the **gmsk\_iq\_receiver** icon back to 2e6 and then set the **pll\_bw** parameter of the **gmsk\_pll\_transmitter** icon to 250e3. The Sue2 schematic should appear as shown below once it is saved.



- Run CppSim, and then plot the resulting eye diagrams for the ifilt and qfilt signals. The resulting plot should appear as shown below.

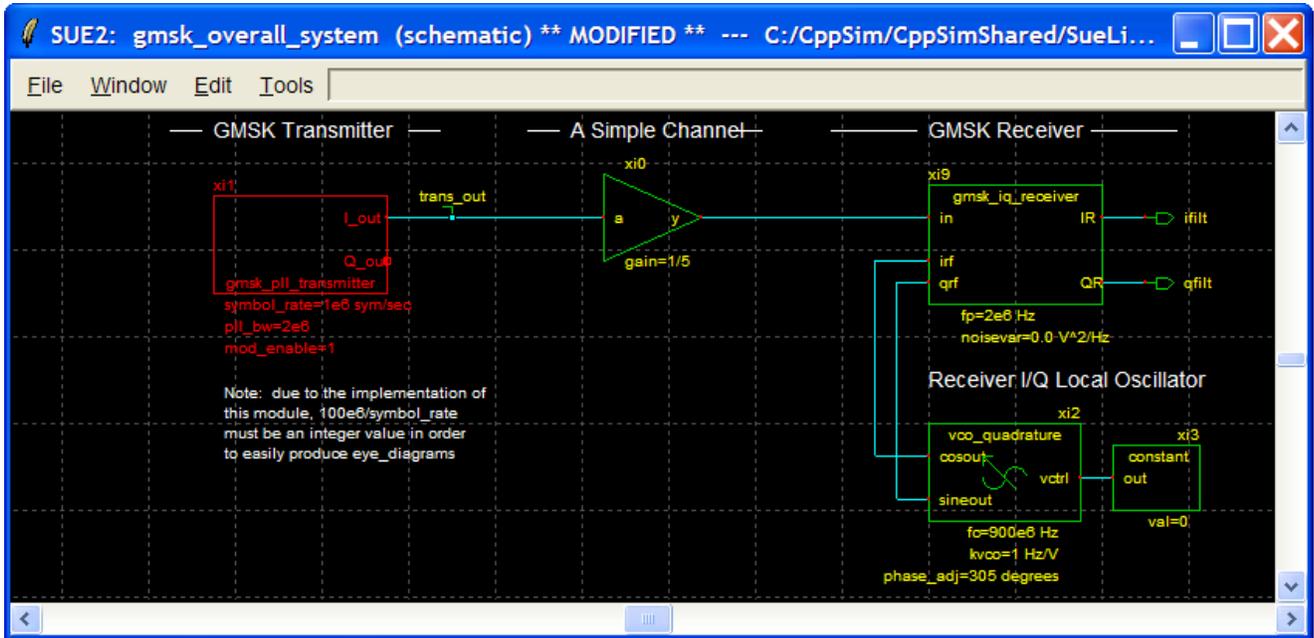


Notice that there is a different amount of intersymbol interference that occurs when setting the transmitter pll bandwidth to 250 kHz as compared to setting the received **fp** parameter to 250 kHz. This is due to the fact that the transmit and receive filters are different in their order and bandwidth characteristics. The transmit PLL filter is a *second order* lowpass with *asymptotic* bandwidth of 250 kHz (see Figure 5 of “PLL Design using the PLL Design Assistant Program” to understand the meaning of asymptotic bandwidth), while the receive filter is a *third order* lowpass composed of three cascaded first order sections that *individually* have 250 kHz bandwidth.

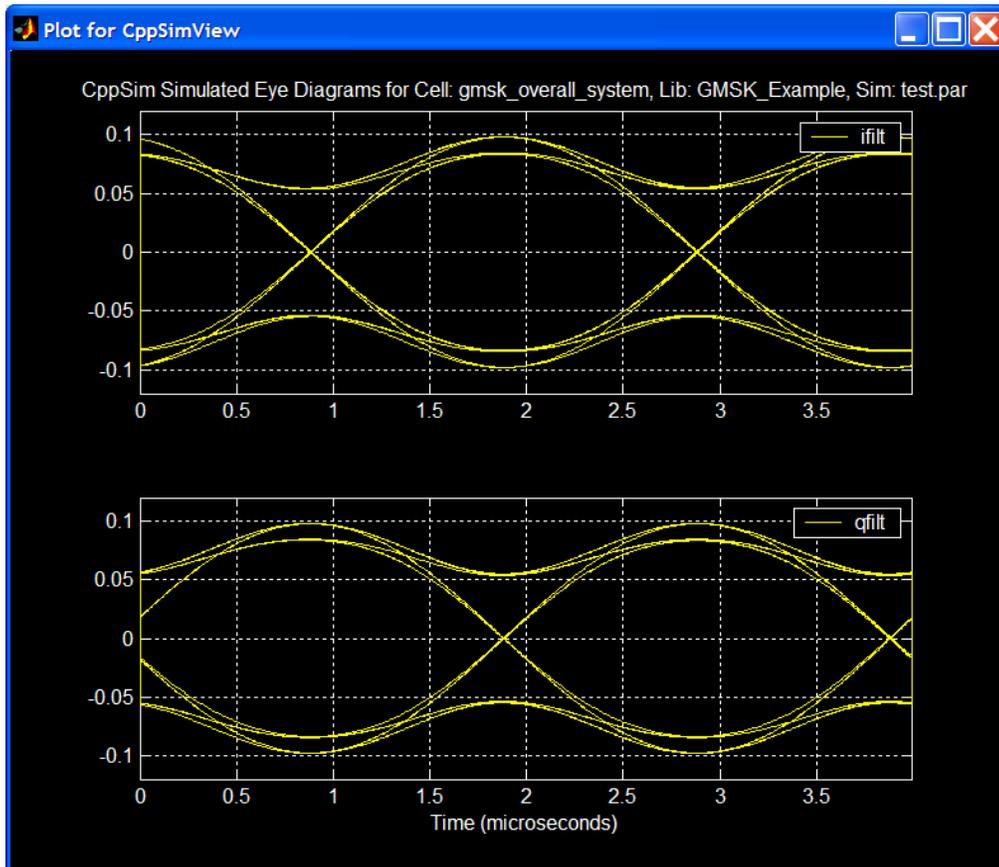
You might consider trying other values for the **pll\_bw** parameter of the **gmsk\_pll\_transmitter** icon to further observe its impact on intersymbol interference.

## B. Phase Synchronization

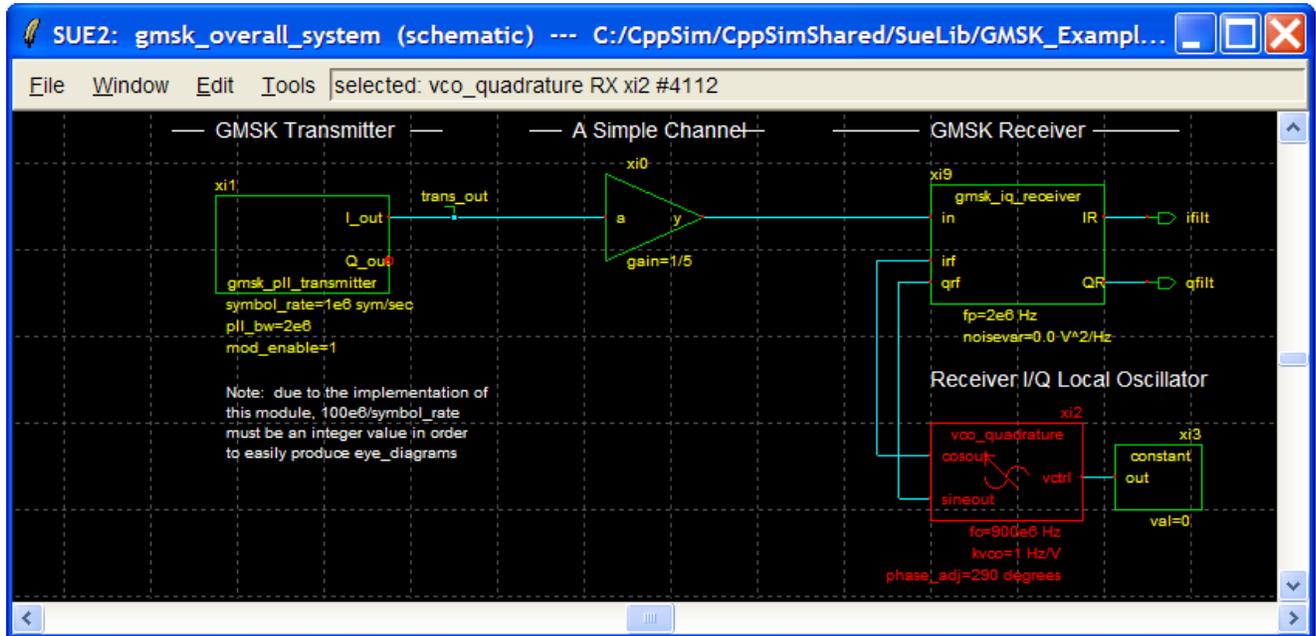
- Within Sue2, double-click on the **gmsk\_pll\_transmitter** icon and then set the **pll\_bw** parameter to 2e6. After saving the changes, the Sue2 schematic of the **gmsk\_overall\_system cellview** should appear as shown below.



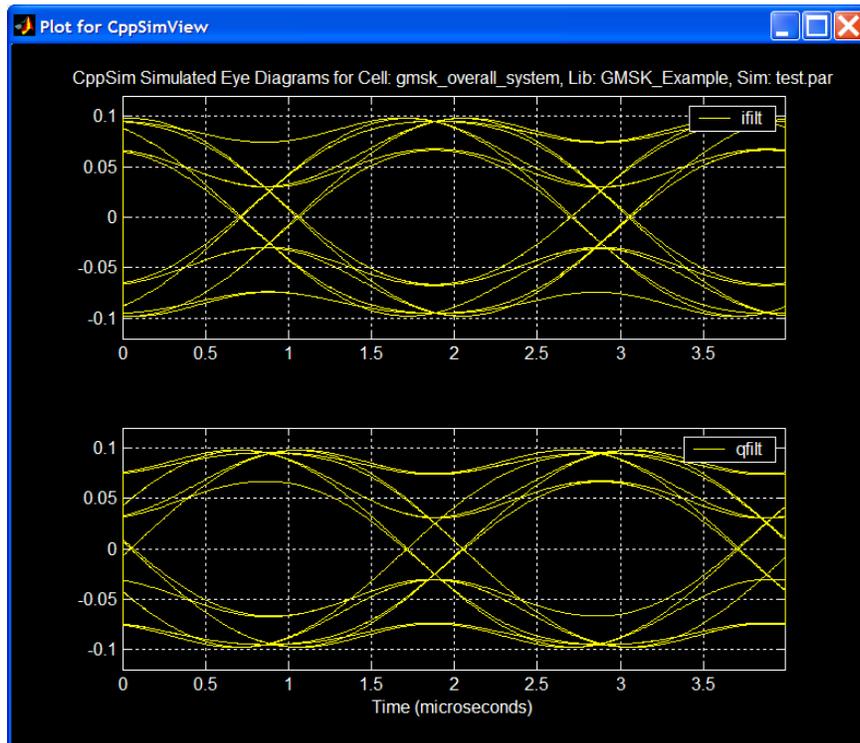
- Within CppSimView, click on the **Run CppSim** button to run the CppSim simulation.
- Replot the eye diagrams for signals **ifilt** and **qfilt** to verify that they look close to ideal as shown below.



- Within Sue2, double-click on the **vco\_quadrature** icon and then change the **phase\_adj** parameter to 290 degrees. After saving the changes, the **gmsk\_overall\_system** cellview should appear as shown below.



- Within CppSimView, click on the **Run CppSim** button to run the CppSim simulation.
- Replot the eye diagrams for signals **ifilt** and **qfilt** and observe that they now appear to be impacted by intersymbol interference as shown below.

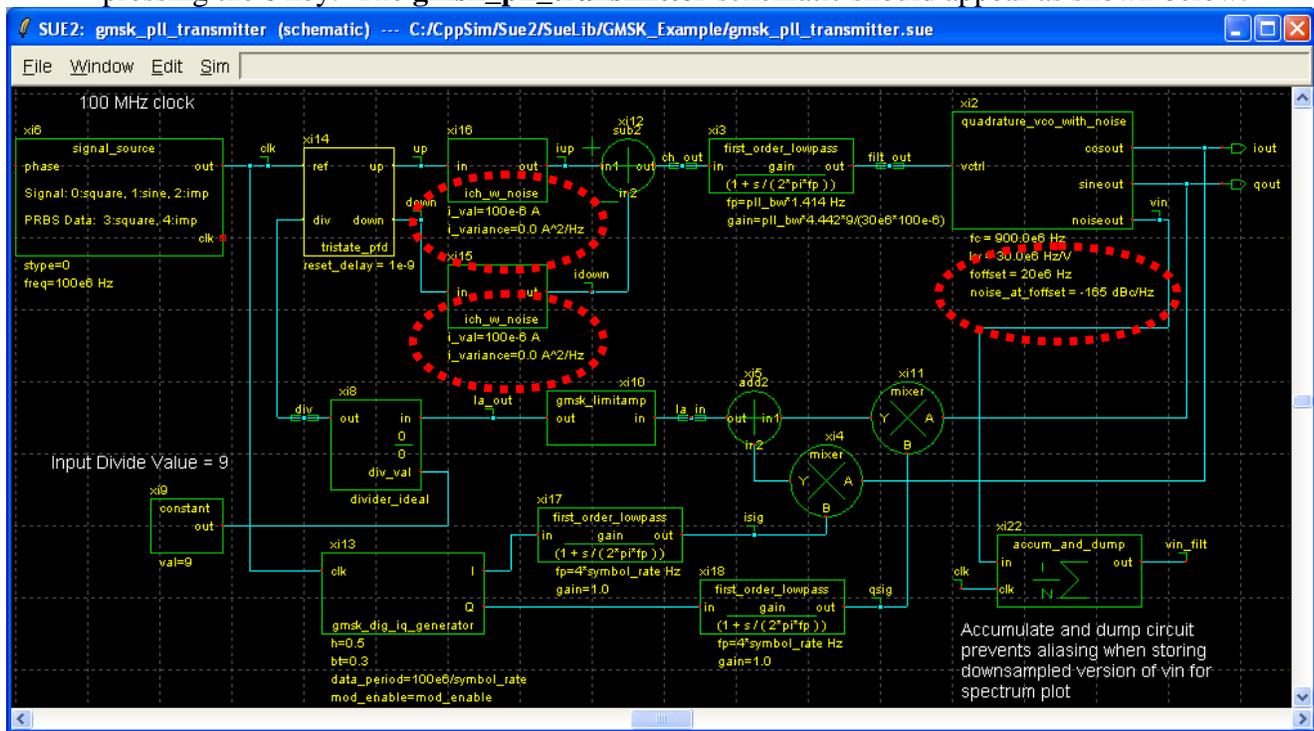


We observe that the closing of the eye observed above is *not* due to intersymbol interference by filtering, but rather by having the wrong phase setting when demodulating the I/Q signals. Therefore, when doing coherent demodulation, as we are in this case, we see that one must appropriately select the right phase offset in the receiver to get nice I/Q waveforms.

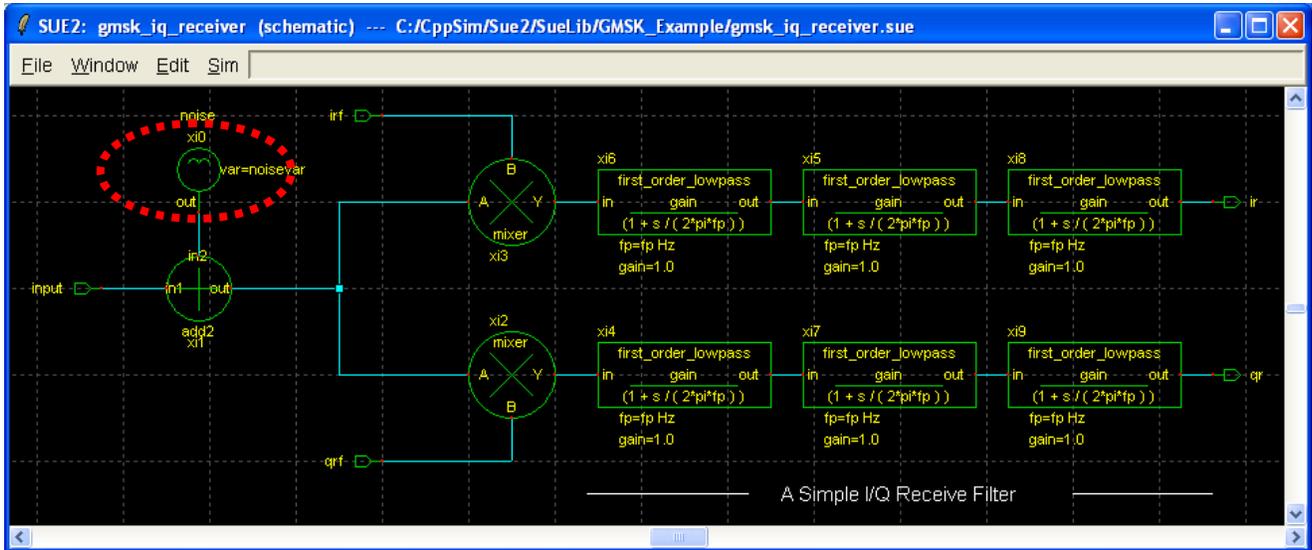
### C. Transmitter and Receiver Noise

We will not provide many details related to noise in this document, but will rather simply point out where to add primary noise sources within the transmitter and receiver of the given GMSK transceiver simulation framework.

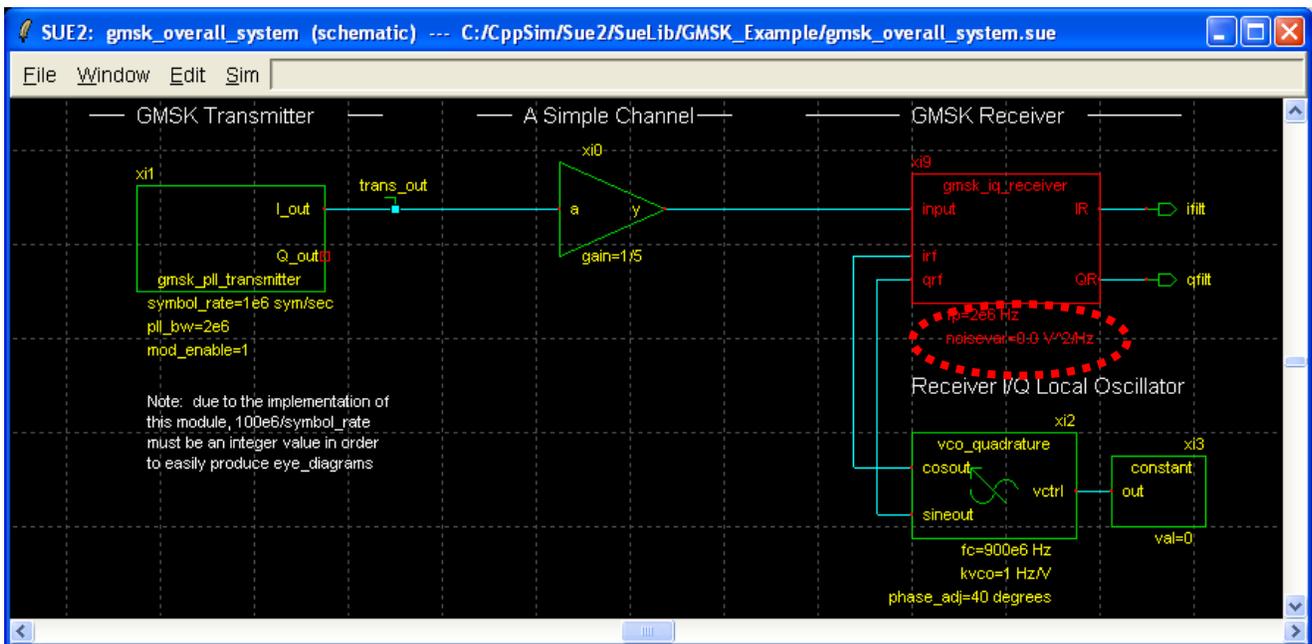
- Within Sue2, descend down into the **gmsk\_pll\_transmitter** icon by clicking on it and then pressing the **e** key. The **gmsk\_pll\_transmitter** schematic should appear as shown below.



- As indicated by the circled items above, the primary noise sources in the transmitter are:
  - Charge pump noise (left circles): this is currently set to zero. Check out the “PLL Design using the PLL Design Assistant Program” for more information on the impact of this noise source.
  - VCO noise: this is currently set such that the VCO noise is -165 dBc/Hz at a 20 MHz offset (which is essentially what is required for a GSM transmitter).
- Within Sue2, return to the **gmsk\_overall\_system** cell view by pressing **Ctrl-e**. Then descend down into the **gmsk\_iq\_receiver** – its cellview should appear as shown below.

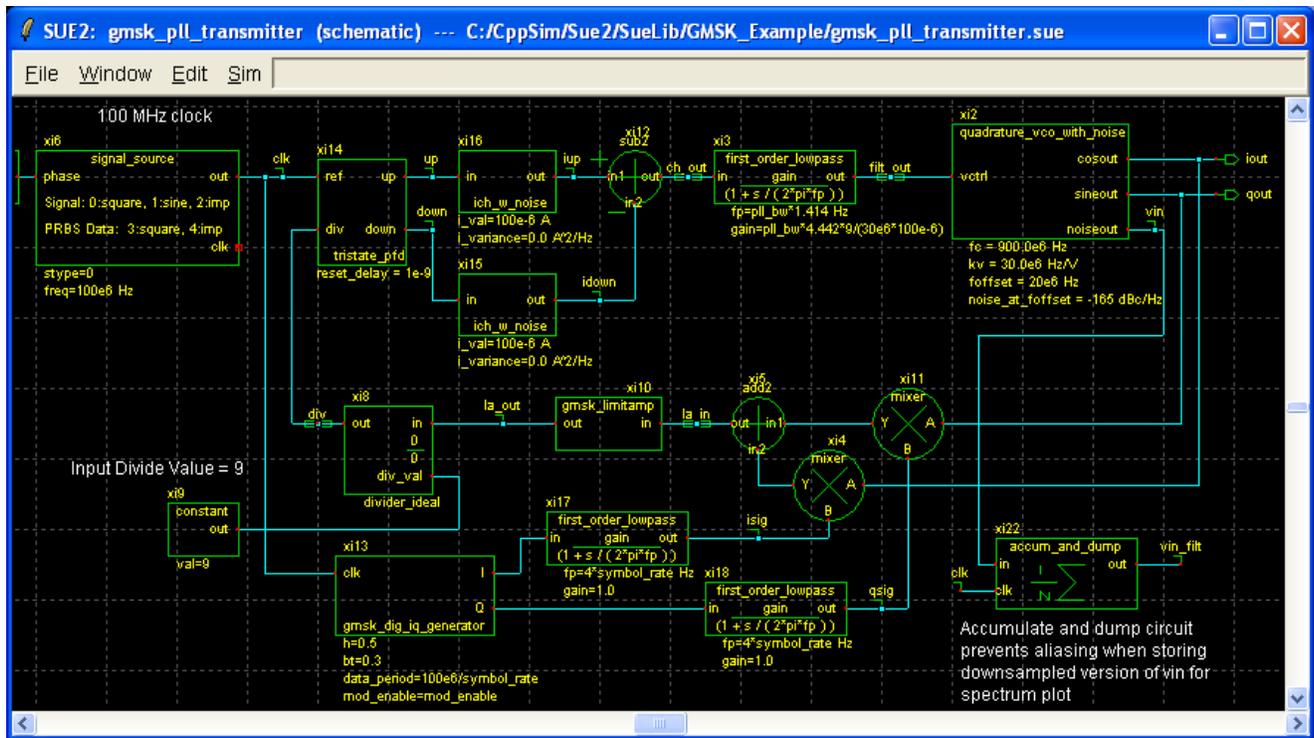


- The overall noise source within the receiver is simply modeled as an equivalent input noise source before the I/Q mixers. In practice, the equivalent input noise would primarily be a function of noise in the LNA, mixers, and filters in the RF path (none of which are included above). You might consider entering different values of **noise\_var** by returning to the **gmsk\_overall\_system** cellview (press **Ctrl-e**) and then double-clicking on the **gmsk\_iq\_receiver** icon as shown below.



## Design Overview of the PLL Transmitter

Let us now focus on calculation of the PLL loop filter parameters within the transmitter, which is shown below for convenience. In particular, we will list the assumed parameter settings for the PLL, provide a model for the impact of the I/Q mixing operation on the PLL, and then calculate the loop filter parameters using the PLL Design Assistant program.



## A. Assumed Parameter Settings

We will assume that the following parameters of the PLL are set by application constraints:

- Reference frequency: 100 MHz
- Output frequency: 900 MHz
- Divide value:  $N_{nom} = 900 \text{ MHz} / 100 \text{ MHz} = 9$
- PFD structure: Tristate
- PLL order: 2
- PLL type: 1 (i.e., an integrator is not included within the loop filter)
- VCO:  $K_v = 30 \text{ MHz/V}$  with phase noise of  $-165 \text{ dBc/Hz}$  at 20 MHz offset
  - Note that the VCO phase noise is modeled as falling at 20 dB/dec across its entire offset frequency range. While the phase noise of a practical VCO has a higher slope at low frequency offsets, and eventually levels off at higher frequency offsets, the approximation assumed here is adequate for illustrating the key points of simulating a GMSK transceiver.
- Charge pump:  $I_{ch} = 100 \text{ microAmps}$  (a reasonable value for many applications – see “PLL Design using the PLL Design Assistant Program” for more details on the tradeoffs associated with setting the charge pump current). Note that we have set the charge pump noise to zero for simplicity in this document.
- Simulation time step:  $T_s = 1/9e9$  (i.e., the sample rate is set to 10 times that of the highest frequency, which is the VCO output).

Given the above, we would like to calculate the loop filter parameters to achieve a given PLL bandwidth (i.e., **pll\_bw** within the Sue2 schematic of **gmsk\_pll\_transmitter**).

## B. Modeling

Figure 3 displays a block diagram corresponding to the PLL portion of the transmitter. The system is very similar to a classical integer-N PLL except for the inclusion of the limit amp and I/Q mixing section. The limit amp can be ignored in our modeling effort (at least to first order) since it has an output with the same zero-crossing values (i.e., phase) as its input, and the divider only responds to zero-crossings (i.e., phase) at its input. The impact of the I/Q mixing section can be modeled as a function of the phase of the transmitter output,  $\Phi_{out}$ , and the phase of the I/Q generator,  $\Phi_{mod}$ .

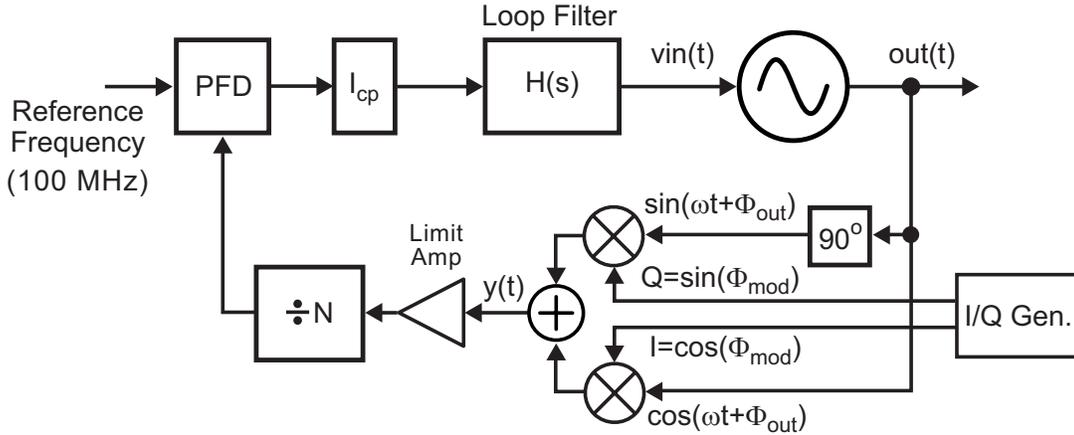


Figure 4: Simplified block diagram of the PLL used within the GMSK transmitter.

To model the relationship between the output,  $y(t)$ , and input phase signals,  $\Phi_{out}$  and  $\Phi_{mod}$ , of the I/Q mixing section, we write out the I/Q mixing operation as

$$y(t) = \sin(\omega t + \phi_{out}) \sin(\phi_{mod}) + \cos(\omega t + \phi_{out}) \cos(\phi_{mod}) = \cos(\omega t + \phi_{out} - \phi_{mod})$$

The above expression reveals that the phase of the output of the I/Q section corresponds to the subtraction between the phase of the transmitter output,  $\Phi_{out}(t)$ , and I/Q generator output,  $\Phi_{mod}(t)$ . Therefore, we can represent the overall PLL portion of the transmitter with the linearized PLL model shown in Figure 4 below. The I/Q mixer section is represented as a subtraction operation, and the remaining PLL blocks are modeled as discussed in the “PLL Design using the PLL Design Assistant” document. In the model, the divider value variations,  $n[k]$ , have a value of zero since we are using an integer-N approach, and the input reference phase,  $\Phi_{ref}[k]$ , is also assumed to be zero since we are ignoring reference noise at this point.

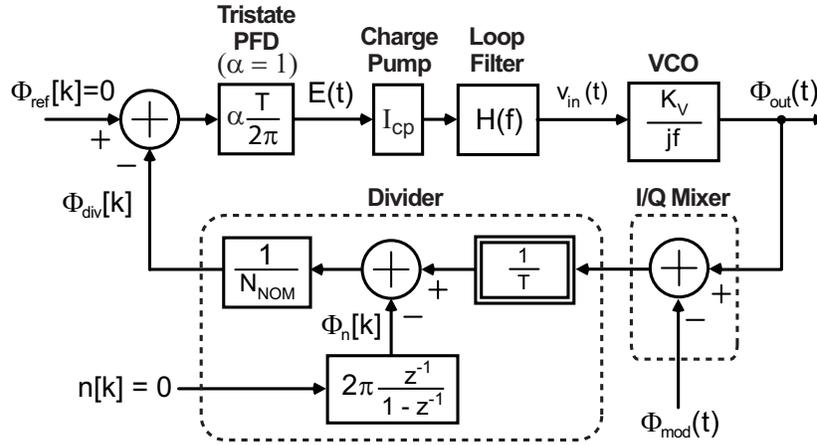


Figure 5: Linearized model of the PLL used within the GSM transmitter.

By taking advantage of the zero value of several of the signals in Figure 4, we obtain the simplified model shown in Figure 5. Here we have also parameterized the loop filter,  $H(f)$ , as a first order transfer function. Given the values that have already been chosen for the PLL, there are only two free parameters left – **gain** and  $f_p$ . The values of these parameters can be determined by using the PLL Design Assistant program.

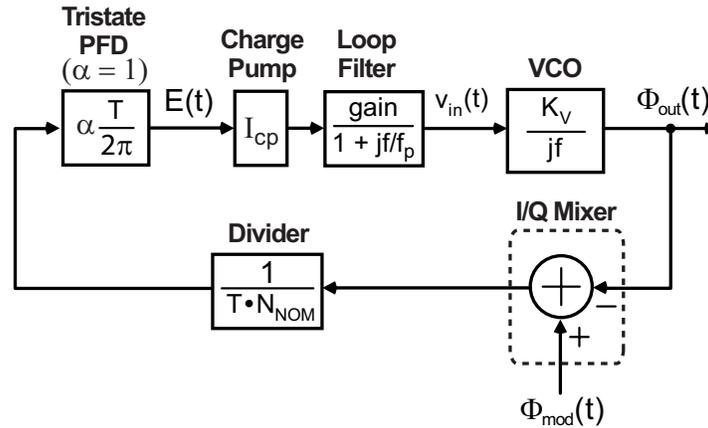


Figure 6: Simplified PLL model.

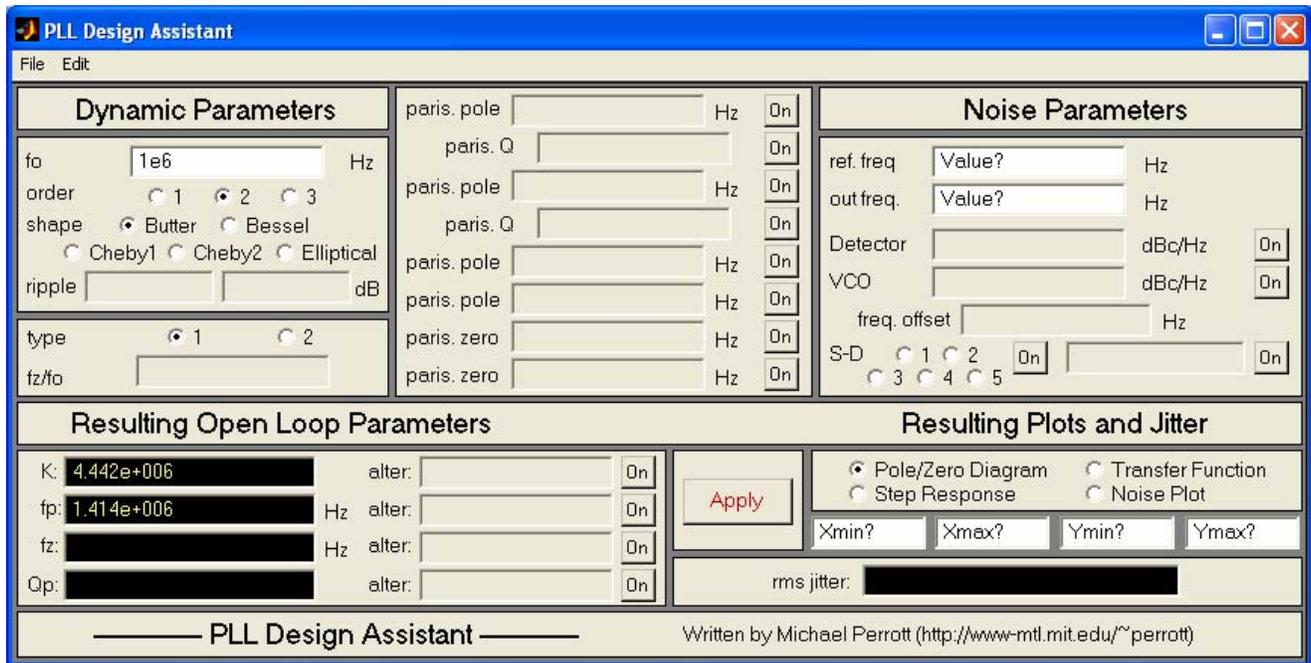
### C. Loop Filter Design using the PLL Design Assistant

Examination of page 10 of “PLL Design using the PLL Design Assistant Program” reveals that the **gain** parameter of the loop filter is related to the other PLL parameters as

$$gain = K \frac{N_{NOM}}{\alpha K_v I_{cp}}$$

where  $K$  corresponds to the desired overall open loop gain of the PLL. The desired overall open loop gain is, in turn, a function of the desired bandwidth of the PLL. We shall determine the value of  $K$  by using the PLL Design Assistant program.

- Click on the **PLLDesign** icon in Windows to start the PLL Design Assistant.
- Set the value of **fo** to 1e6, and then click on the **Apply** button. The PLL Design Assistant should now appear as shown below.



- Now change the value of **fo** to 2e6 and again click on the **Apply** button. You should notice that the values of  $K$  and  $fp$  are doubled such that they become 8.884e6 and 2.828e6, respectively. We therefore surmise that we can relate the bandwidth of the PLL, **fo**, to its open loop gain,  $K$ , as

$$K = pll\_bw \cdot 4.442$$

where we have defined  $pll\_bw = fo$  (note that we use  $pll\_bw$  within the Sue2 schematic of the **gmsk\_pll\_transmitter** cell as shown above).

Given the above, we can express the gain of the loop filter as

$$gain = pll\_bw \cdot 4.442 \frac{9}{1 \cdot 30e6 \cdot 100e-6}$$

Similarly, we can express the pole value of the loop filter as

$$fp = pll\_bw \cdot 1.414$$

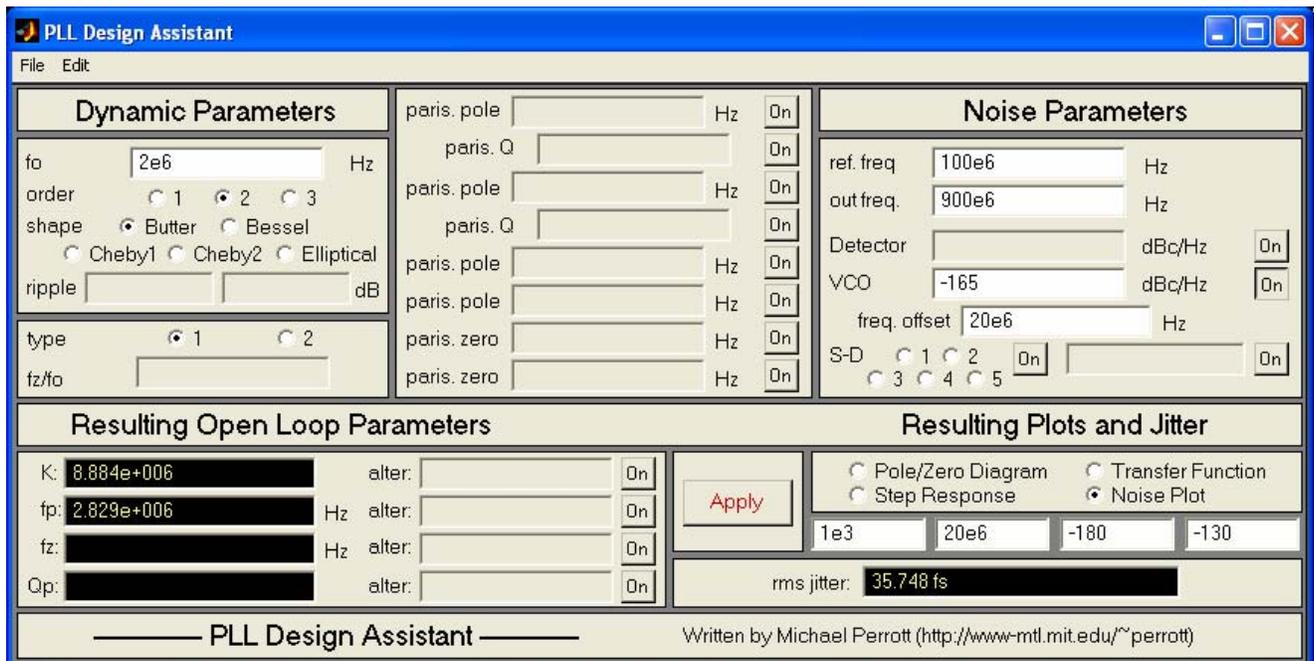
If one looks at the parameters for the loop filter within the Sue2 schematic of the **gmsk\_pll\_transmitter** cell, you will notice this very parameterization.

### Noise Analysis of the Unmodulated PLL using the PLL Design Assistant

We will now calculate the impact of VCO noise on the unmodulated PLL transmitter output. The resulting calculation will be compared to simulated results obtained earlier in this document.

- Within the PLL Design Assistant, set the value of the following parameters as specified
  - **fo** = 2e6
  - **ref. freq** = 100e6
  - **out freq.** = 900e6
  - **VCO noise** = -165 dBc/Hz at **freq. offset** = 20e6
- Click on the **Noise Plot** radio button
- Set the axis limits to: **1e3 20e6 -180 -130**
- Click on the **Apply** button

The PLL Design Assistant and its corresponding output plot should now appear as shown below



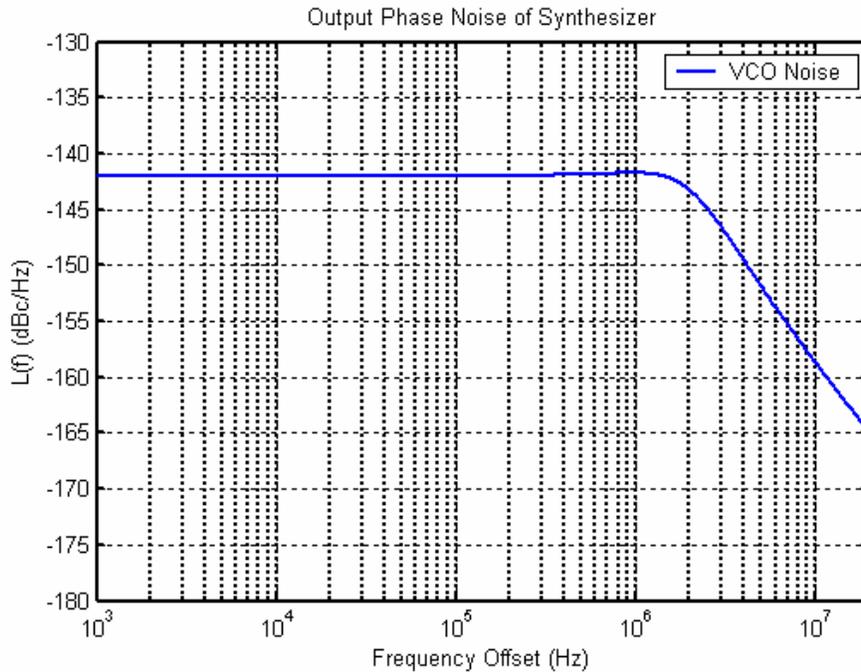


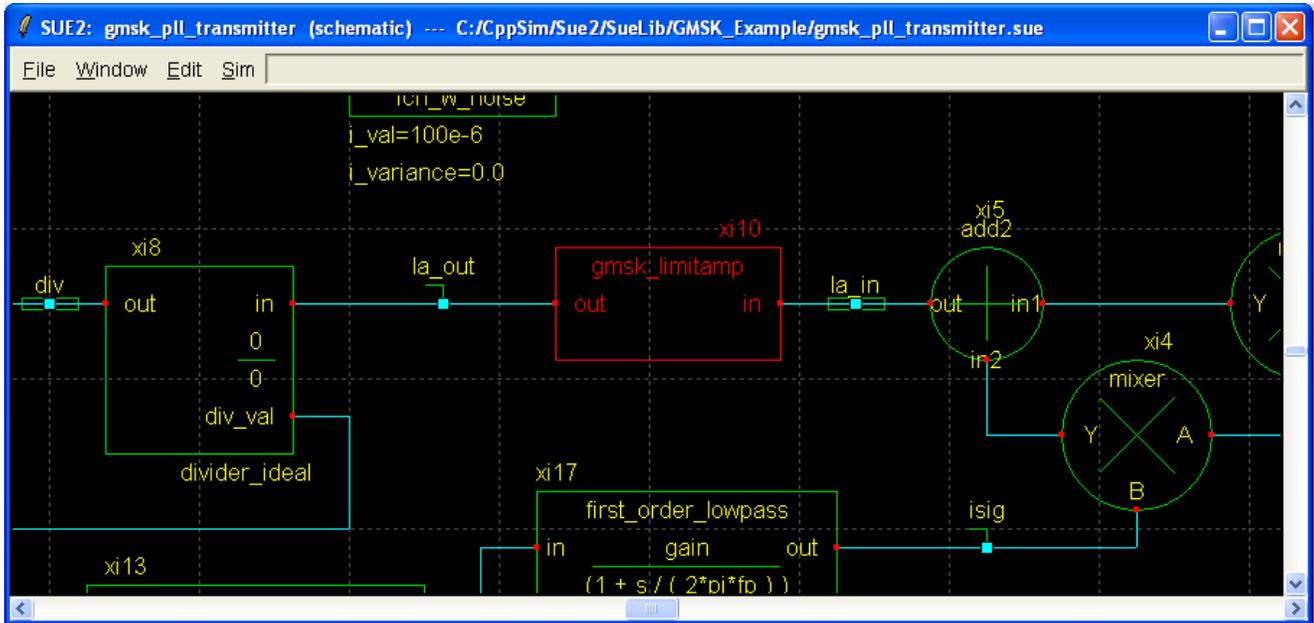
Figure 7: Calculated phase noise of the unmodulated PLL transmitter.

One should now compare the above calculated phase noise plot to the simulated one shown in Figure 3 on page 15 of this document. Note that, for simplicity, we have ignored other noise sources in the PLL such as charge pump noise. It is fairly straightforward to analyze the impact of such noise within the PLL Design Assistant, and include it within CppSim when seeking a more realistic system description.

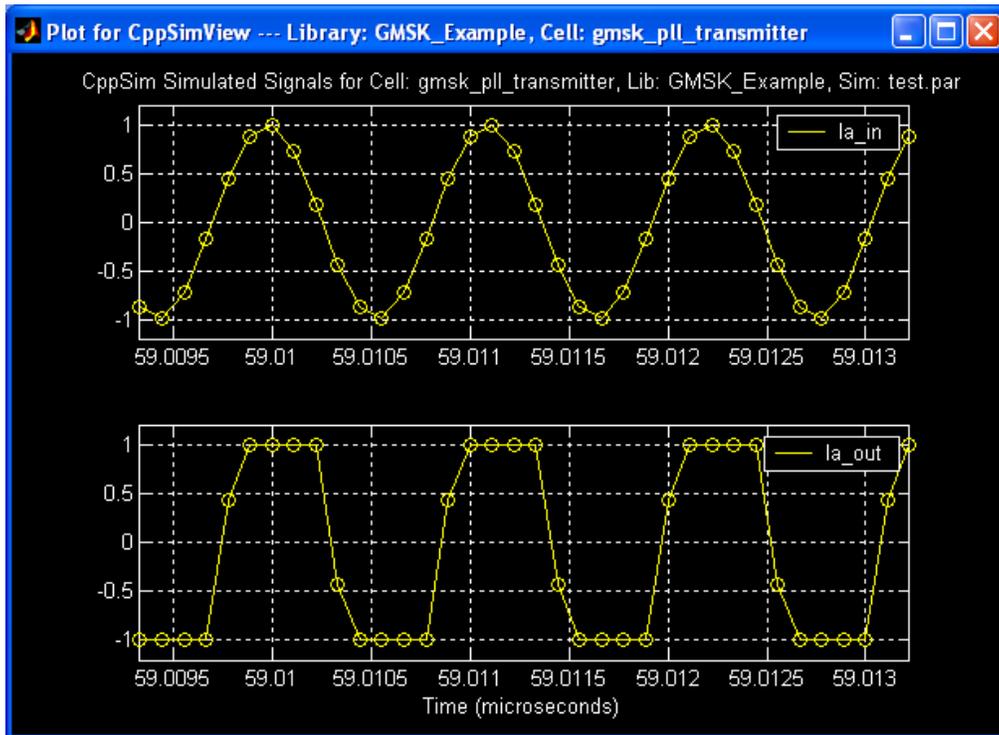
### The Need for the **gmsk\_limitamp** Module

Consider now the fact that we have demonstrated a close comparison between the calculated and simulated noise of the PLL transmitter, and have done so with a coarse simulation time step that is only a factor of ten higher than the transmitter output frequency. The secret of our success is the application of an area conservation protocol referenced below, which, in turn, relies on the inclusion of the **gmsk\_limitamp** block within the system simulation. We briefly describe the relevant issues on this topic in this section.

- Within Sue2, go to the **gmsk\_pll\_transmitter** cellview and then notice the presence of the **gmsk\_limitamp** icon as shown below.



The function of the **gmsk\_limitamp** block is to convert the sine wave output of the I/Q mixing section to a square wave signal that follows the area conservation protocol described in “*Fast and Accurate Behavioral Simulation of Fractional-N Synthesizers and other PLL/DLL Circuits*” (DAC 2002, PDF download available at <http://www-mtl.mit.edu/researchgroups/perrottgroup/publications.html>). In particular, the **divider\_ideal** block shown above requires that its input adhere to the area conservation protocol in order to avoid inducing quantization noise in time due to the relatively large time step of the simulator. The impact of the area conservation protocol on the output square wave and its relationship to the input sine wave of the **gmsk\_limitamp** block are seen in the plot below.



To better understand the functionality of the **gmsk\_limitamp** block, it is recommended that you take a look at its module description.

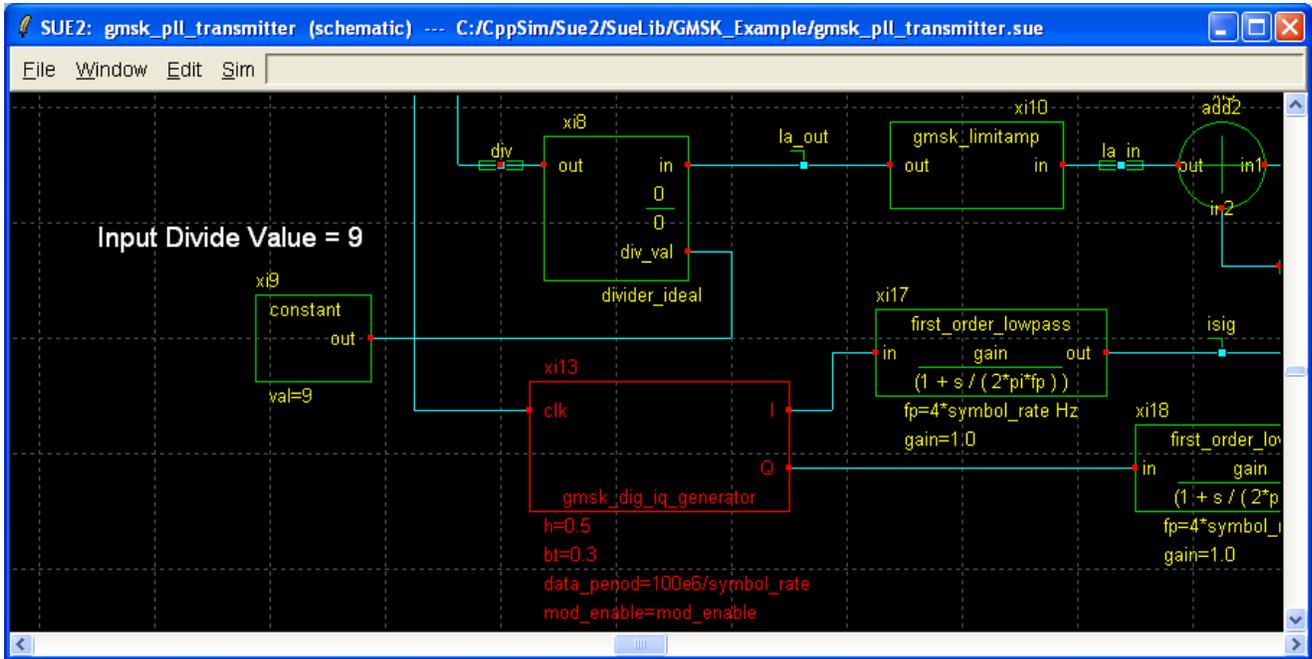
- Double-click the **gmsk\_limitamp** block on your schematic and left-click the **Edit CppSim Code** button.
- Within the Emacs window that opens up:
  - You should see the following segment of code:

```
.....  
module: gmsk_limitamp  
parameters:  
inputs: double in  
outputs: double_interp out  
.....
```

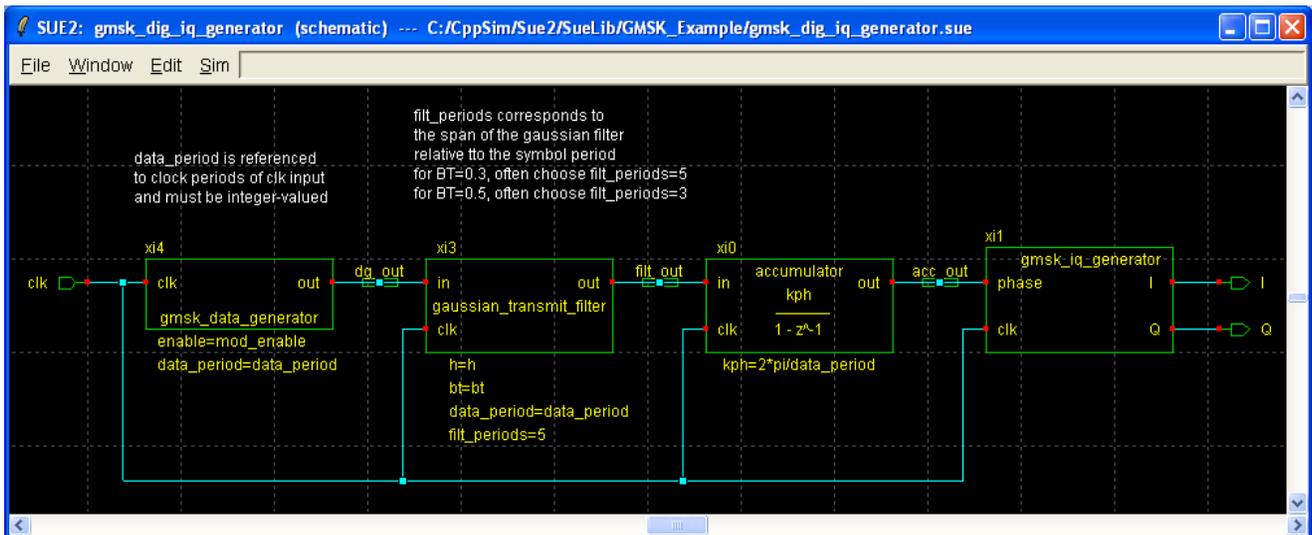
- Notice that the output has type **double\_interp** – this indicates to CppSim that the output follows the area conservation protocol. CppSim will actually treat the output as type **double** during the simulation, but uses the **double\_interp** label to perform error checking before simulation to insure that the area conservation protocol is adhered to where demanded (such as at the input of the **divider\_ideal** module).
- If you look in the **code:** section of the **gmsk\_limitamp** (which directly follows the code above), you will see that the area conservation protocol is fairly straightforward to realize. We simply perform first order interpolation of the input sine wave signal to estimate the location of its zero crossing values, and then use the estimated zero crossing values to generate the area-conserving output square wave signal.

## Overview of the I/Q Generator

We will now provide further details on the implementation of the I/Q generator block, which is highlighted below in the **gmsk\_pll\_transmitter** cellview. In particular, we will describe how clocking and filtering operations are implemented within its blocks.



- Descend down into the **gmsk\_dig\_iq\_generator** cellview in Sue2. The following schematic diagram should appear:



## A. Implementing Clocking Within Modules

CppSim runs through the code for each block every time step of the simulation. However, it's often the case that certain blocks should update their contents at a lower rate based on a clock signal. To illustrate how such clocking functionality is achieved within modules, let us examine the code for one of the blocks within the I/Q generator shown above.

- Double-click on the **gmsk\_data\_generator** block and then left-click on the **Edit CppSim Code** button.
- Within the Emacs windows that appears. You should see the following code:

```

.....
module: gmsk_data_generator
parameters: int enable, int data_period
inputs: double clk
outputs: double out
classes: Rand datagen("bernoulli"), EdgeDetect clkedge()
static_variables: int count
init:
datagen.reset();
count=1;
if (enable == 1)
    out = 1.0;
else
    out = 0.0;
code:
if (clkedge.inp(clk))
{
    if (count == data_period)
    {
        count=0;
        if (enable == 1)
            out = datagen.inp();
        else
            out = 0.0;
    }
    count++;
}
.....

```

Notice that the **classes:** section declares an **EdgeDetect** object called **clkedge**. The function of **clkedge** is to take on a value of 1 when a positive edge is encountered at its input, and to output 0 otherwise. In the **code:** section, the **clkedge** object is then used to prevent execution of the module code *unless* a positive clk edge is encountered. Note that if it is desired to execute the module code on a negative edge instead, one would change the statement

if (clkedge.inp(clk))

to

if (clkedge.inp(-clk))

## B. Creating the Gaussian FIR Filter

The **gaussian\_transmit\_filter** block within the **gmsk\_dig\_iq\_generator** cellview has the task of filtering the data sequence with an FIR Gaussian filter whose discrete-time impulse response  $p[k]$ , is formulated as

$$p[k] = \frac{h}{2} \frac{1}{\sqrt{\pi}\sigma} e^{-\left(\frac{k}{\sigma}\right)^2}$$

where

$$\sigma = \frac{1.178}{BT_d} \frac{T_d}{2\pi T}$$

- Again, double-click on the **gaussian\_transmit\_filter** block and then left-click on the **Edit CppSim Code** button.
- You should see the following code segment:

```

.....
module: gaussian_transmit_filter
parameters: double h, int data_period, int filt_periods, double bt
inputs: double in double clk
outputs: double out
classes: Filter filt("1","1"), List list1(), EdgeDetect clkedge()
static_variables:
init:
///< error checking removed here for simplicity ///<
double sigma, prefix, data_period_r, cur_val;
int half_filter_length,i;

data_period_r = (double) data_period;
sigma = 1.178*data_period_r/(bt*2.0*pi);
prefix = h/(2.0*sqrt(pi)*sigma);
half_filter_length = (data_period*filt_periods)/2;
for (i = -half_filter_length;i <= half_filter_length; i++)
{
    cur_val = prefix*exp(-(((double) i)/sigma)*(((double) i)/sigma));
    list1.inp(cur_val);
}
// list1.save("filt_vals.dat");
filt.set(list1,"1");
code:
if (clkedge.inp(clk))
{
    filt.inp(in);
    out=filt.out;
}
.....

```

There are two things to note about the above code:

1. The FIR filter is realized using the Filter class. In particular, within the **classes:** section, we declare **filt** as a Filter object whose numerator and denominator values are 1 (i.e., it is initialized as a unity gain block with flat frequency response). Within the **init:** section, we then compute the FIR coefficients based on the above equation, and store their values into a List object named **list1**. The list values are then transferred into the numerator transfer function of **filt** (while retaining a denominator of 1) through the statement  
`filt.set(list1,"1");`
2. The time step of the above filter is set according to the input **clk** signal rather than the simulation time step,  $T_s$ , by using the clocking method discussed in the previous section. In particular, within the **code:** section, the statement  
`filt.inp(in);`

is only executed on rising edges of the clock. Since the **filt** object only updates its contents when its above input function (i.e., the statement above) is executed, we have set the time step of the **filt** object to the period of the **clk** signal (rather than the simulator time step, **Ts**, which would normally be the case if the input function of **filt** were executed every time step of the simulation).

## **Conclusion**

In this document, we have covered basic issues related to behavioral simulation of a simple GMSK transceiver example using CppSim. In particular, we have introduced the reader to the tasks of running CppSim simulations, plotting the I/Q signals in the time domain, examining the output spectrum of the transmitter under both modulated and unmodulated conditions, and viewing the impact of nonidealities such as intersymbol interference, poor phase synchronization, and noise. We also covered basic design calculations for the PLL within the transmitter, showed close correspondence between the calculated and simulated noise performance of the unmodulated transmitter, and provided details of how that close correspondence was achieved with a coarse time step for the simulation. Finally, implementation details of such operations as clocking of blocks and realization of FIR filters were provided within the context of the I/Q generator block used within the transmitter.