DSP and Communication Lab

ECE Course in Jacobs University

Protected: Transmitter Simulink Study

Transmitter Simulink Study

Objective

In this lab you will learn how a basic single-link communications system operates. This lab focuses on high-level simulation of the transmitter with Simulink, to give you a feel for what operations are required and what signals are involved. After this lab, you should understand the operation and purpose of

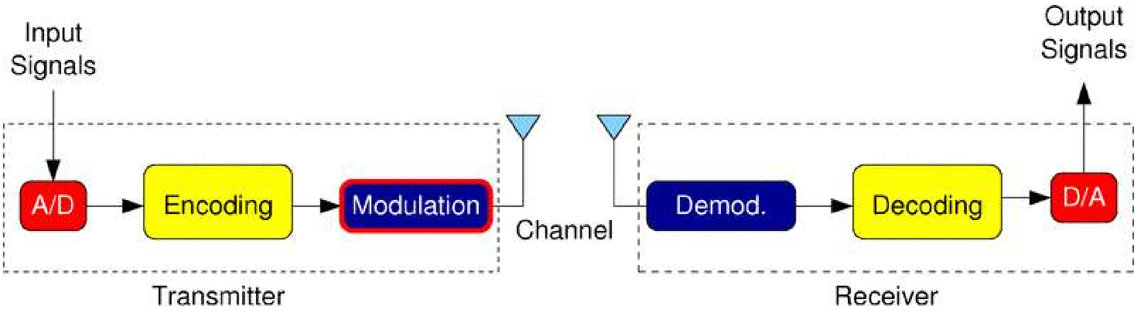
- Bit to symbol mapping
- Constellation diagrams
- Pulse shaping filters
- Up conversion

Pre-lab

For the pre-lab, just read through this lab outline so that you have an idea of what you will be doing. Especially make sure you understand the background section.

Background

The figure below depicts a basic block diagram of a communications system:

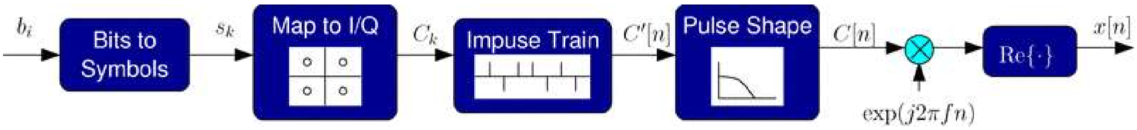


The purpose of a general communications link is to transmit signals from one place to another. If the input signals are analog, such as audio speech, they must be first converted to a digital signal with an A/D converter. To ensure reliable and secure transmission of the signals, digital signals are encoded using the coding block. The modulation block converts these signals to a form suited for the channel (wireless, telephone lines, optical fiber, etc.). At the receiver, the demodulation block converts the signals back to a digital form, after which the binary information can be decoded and converted back to analog.

This semester, we will focus on the modulation and demodulation blocks, which I believe will best reinforce your understanding of DSP programming and communications theory. To keep the lab simple, we will study and implement “uncoded” transmission/reception, which means that the encoding/decoding blocks are trivial pass-throughs. Also, we assume that input and output information is in digital form, so that not much time need be spent on the A/D and D/A blocks.

Transmitter

For this lab, we are concerned with understanding the purpose and operation of the transmitter, whose job is to take an input binary stream and convert this to a form suitable for transmission through a physical channel. The figure below depicts the basic blocks that are involved. For simplicity, phasor domain (complex baseband) processing is assumed. Each of the operations will now be explained in detail



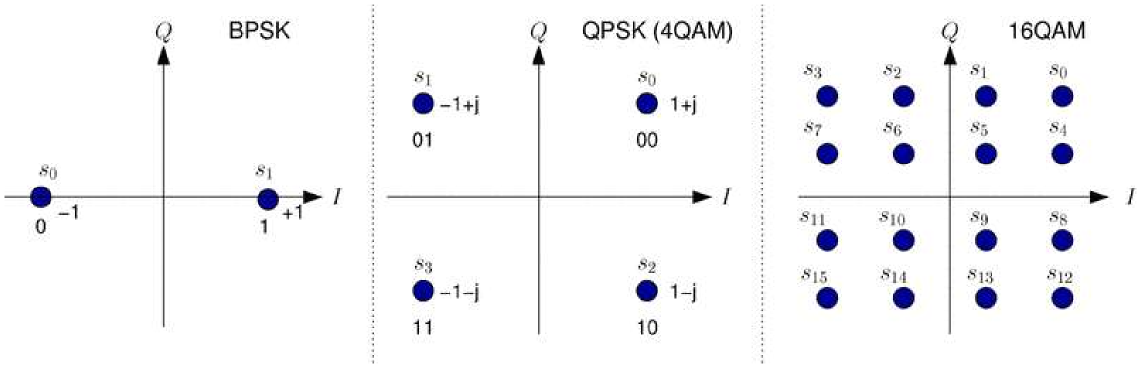
Symbol Mapping

With sufficient signal-to-noise ratio (SNR), we can typically transmit more than two different signals through a communications channel and detect which signal was sent on the other side. For this reason, groups of multiple bits are mapped to symbols, where for a group of *N* bits we require 2^N symbols denoted s_k, which are usually just represented as integers.

Constellation (I/Q) Mapping

Each type of communications channel has a range of “nice frequencies” where propagation characteristics are good. Also, we may want to support multiple users and services on the same channel by dividing users in different frequency bands (called frequency division multiplexing). For this reason, information is usually transmitted by modulating a carrier (sine wave) at some prescribed frequency. For most channels and systems, information can be conveyed using both the phase and amplitude of the signal. A convenient way to exploit these two dimensions is to represent transmitted signals in the complex plane in terms of the I=in-phase (real part) and Q=quadrature (imaginary part) components.

The I/Q mapping block takes each symbol s_k and maps it to a corresponding complex baseband value C_k in the complex plane. These values are usually chosen to be as different (or far apart) as possible so that at the receiver we can discern as clearly as possible what was transmitted. A nice way to represent the mapping of symbols to complex baseband I/Q values is with a *constellation diagram*. The figure below depicts a few different constellations ranging from BPSK (that has only 2 symbols) to 16QAM (16 symbols).



Pulse-Shaping

If we simply transmit C_k as rectangular pulses (like what would be present in a digital circuit), we obtain a sinc-like spectrum, which is fairly wide (decays slowly with increasing frequency). For applications like wireless, where the bandwidth must be limited, a sinc spectrum is very undesirable. Instead, we can map each symbol to a pulse with a desired shape (and more confined spectrum) and transmit the superposition of these pulses in time.

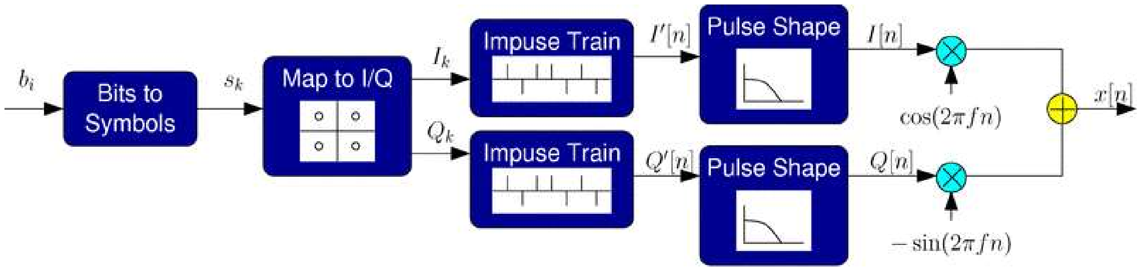
Pulse-shaping is accomplished by converting the symbol stream to a train of pulses that are 1 sample wide and separated by the desired symbol period. By passing this pulse train through a filter that has the desired frequency response of our pulse, we obtain a set of shifted and superimposed pulses that are weighted by the desired I/Q values of the symbols. The resulting spectrum typically is much more confined than square pulses, but conveys the same information.

Up-conversion

Since we often do not transmit signals at baseband, we need to shift the resulting complex baseband waveform *C[n]* to a higher frequency, which is accomplished by the final multiplication block. Note here that since we use the sample number *n* in the exponential, *f* is a *discrete* carrier frequency on the range from 0 to 0.5. All practical channels support only real signals, so the final block takes the real part of the complex signal before physical transmission.

Real Processing

Although the complex baseband (phasor) representation of signals is often more compact, it is usually computationally more efficient to only perform DSP operations on real-valued quantities. For this reason, for actual implementation, we would likely implement the transmit chain using the form below:



Note that this is mathematically *identical* to the previous complex case, but is more efficient to implement.

Receiver

Although not drawn here, you should keep in mind that although the signal is now well-suited for transmission through the channel, the receiver will now need to undo all of these operations. But, if you understand the transmitter well enough, you can imagine what the receiver will need to do.

Procedure

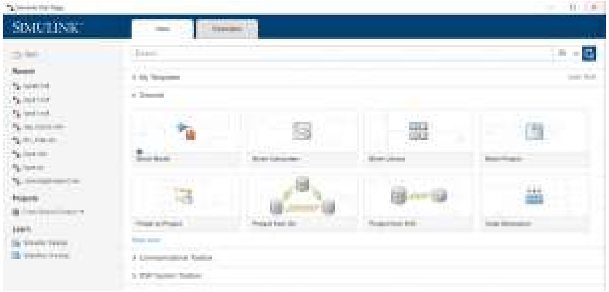
To complete this lab, you need access to a computer with MATLAB and Simulink. Simulink will allow us to incrementally construct the transmitter, where we can probe signals along the way. I hope this will help you understand how the whole transmitter works and what the signals inside are supposed to look like. This intuition will be invaluable when you are doing later labs to know whether things are operating correctly or not!

Starting and Navigating Simulink

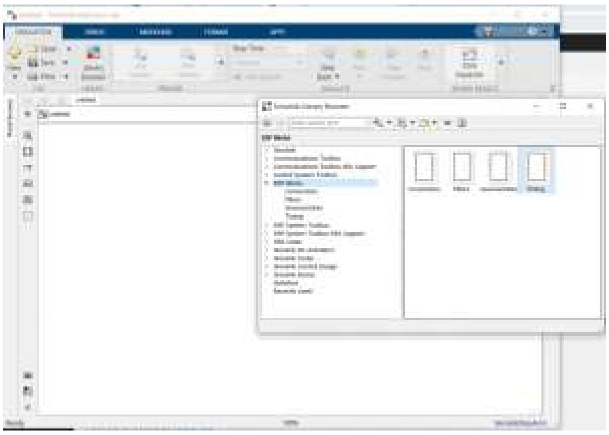
Simulink is a graphical environment for constructing and simulating block diagrams, which makes it well suited for understanding the communications system to be developed in this lab. Although Simulink can simulate both continuous and discrete systems, we will focus on the discrete case, since DSP operates in this regime. For this reason, the sample time should always be 1, meaning that time units are in samples, frequency is discrete frequency (cycles per sample), etc.

For this particular lab, you will need to unzip the custom DSP blocks in dsp_lab.zip to your /MATLABxxx/toolbox/matlab directory, which should create a new directory there called dsp_lab.

You can start Simulink by simply typing `simulink` in the MATLAB command window. Then choose "Simulink -> Blank Model" as showed in the picture below:

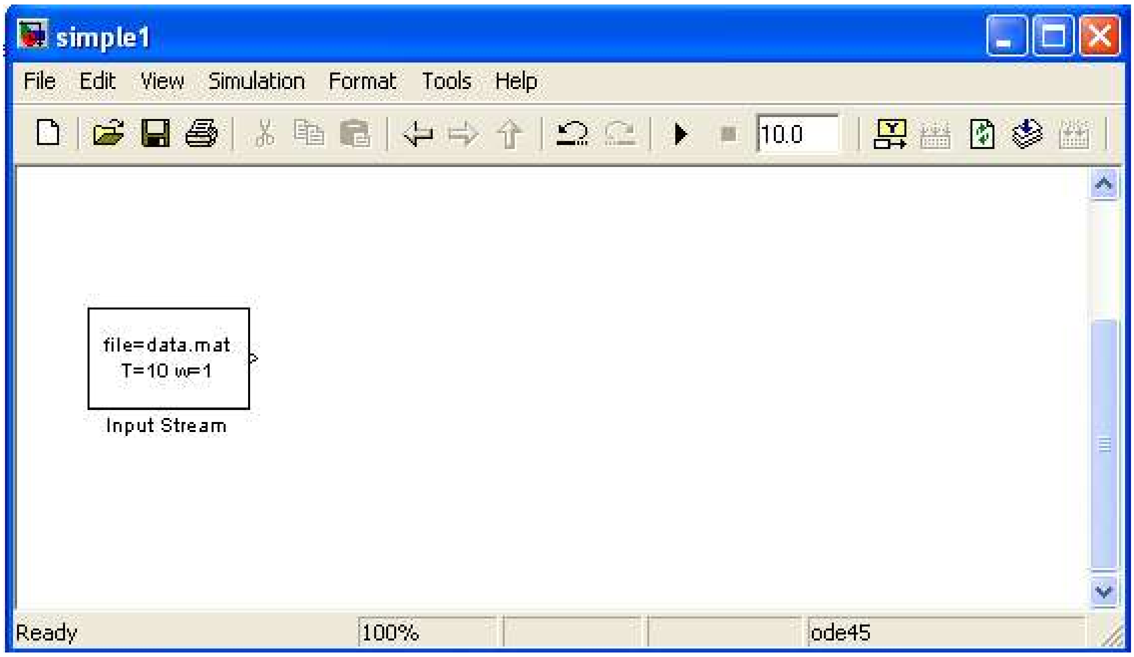


A blank window will appear where you can build your model. Click "Library", you should see a menu that gives you a palette of blocks that can be used to construct and simulate block diagrams. There is also a "DSP Blocks" category which is a set of blocks that have been developed to simplify these labs. Depend on different operating system, there may be a message show to ask you to "fix the problem to transfer old version files into new version file, please click the message to **fix the problem by choosing "keep the old version"** option.



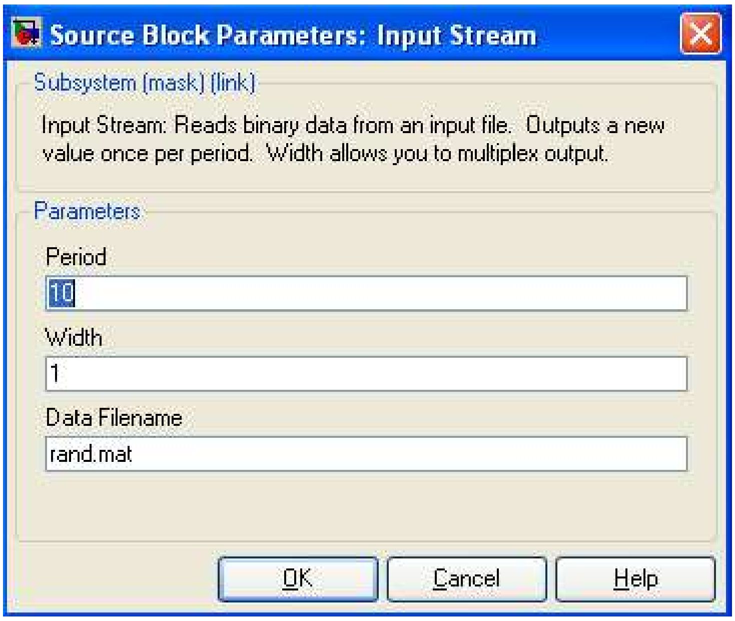
Starting Simple

First, in the palette go to `DSP Blocks->Sources/Sinks` and drag an `Input Stream` block to your design. Your design should look like



The input stream block reads binary data from a MATLAB data file that has a single vector called `data` containing 0s and 1s. As you can see, the Simulink blocks often have useful information printed on them. Here you can see that the block will read data from a file called data.mat, that the bit period is 10 (samples), and that the output width (how many bits are presented at the output per period) is 1.

To change the parameters of the block, double click on it, and you should see the following menu. Modify the parameters as shown.


Source Block Parameters: Input Stream

Subsystem (mask) (link)

Input Stream: Reads binary data from an input file. Outputs a new value once per period. Width allows you to multiplex output.

Parameters

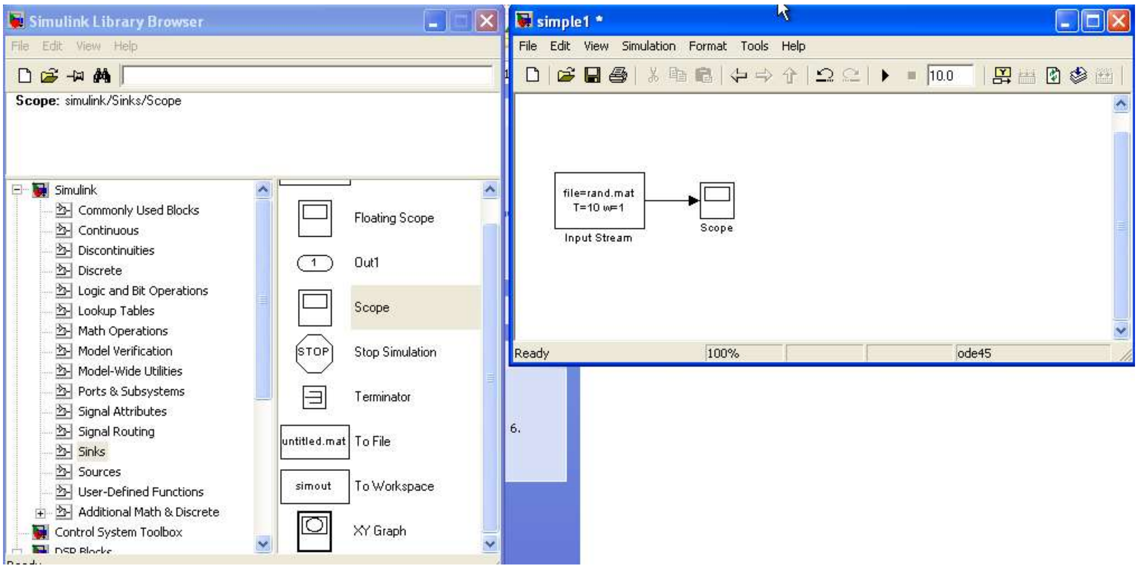
Period: 10

Width: 1

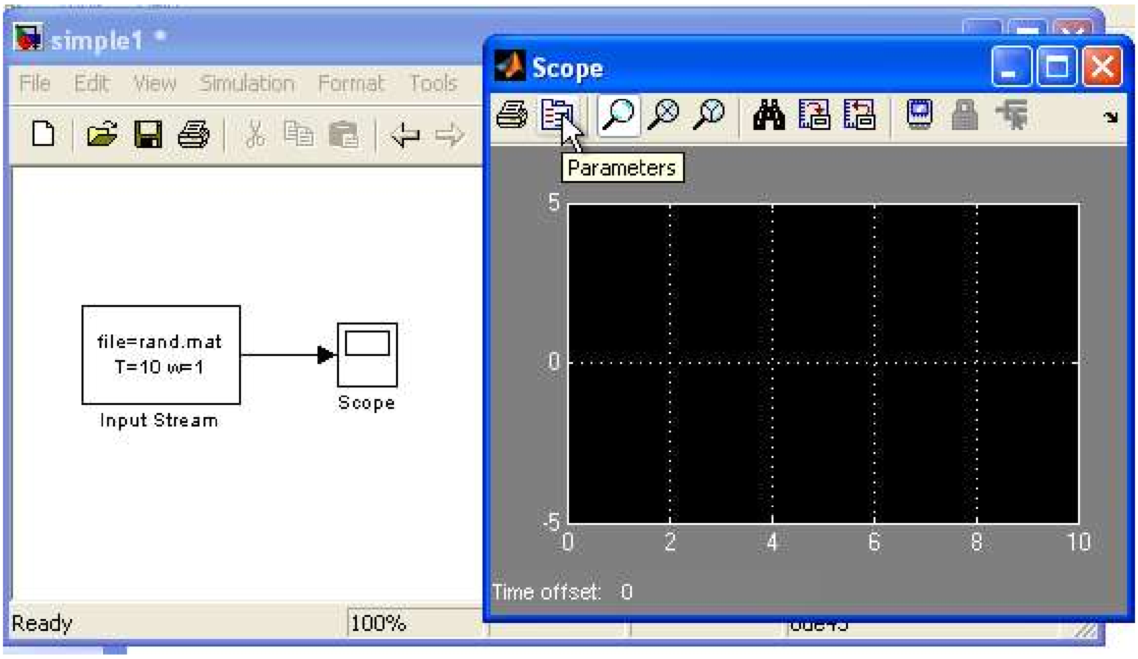
Data Filename: rand.mat

Next, grab the file `rand.mat` and place it in the same directory as you saved your model (.mdl) file. This file just contains 10000 random bits. *Note that you need to change the working directory of MATLAB to be your project directory, or you will probably get errors.*

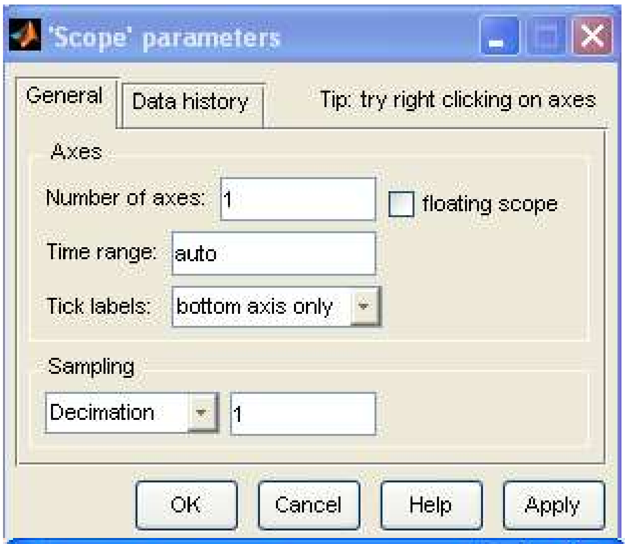
In order to see something, we need to connect a sink to the source block, so grab the block `Simulink->Sinks->Scope` and place this in your design. Connect the scope to the output of your source by dragging between the two terminals. Alternately, you can select the source block, and then while pressing [CTRL], click on the destination block. Your design should look something like the picture below:



To see the scope, double click on it, and a scope display window will open:



To modify the parameters of the scope, click on the the little icon as depicted above. The important parameter in the “General” tab is the Decimation or Sample Time. For most of the lab, we will do discrete simulations, so leave this as Decimation of 1 sample. Note here you can increase the number of axes (or number of inputs to the oscilloscope) when multiple signals must be plotted. The time range allows you to set how many samples are shown at once. For “Data History” we will uncheck the box “Limit Data Points” so that all samples are stored. For very long simulations, you may need to limit the samples, however.


'Scope' parameters

General | Data history — Tip: try right clicking on axes

Axes

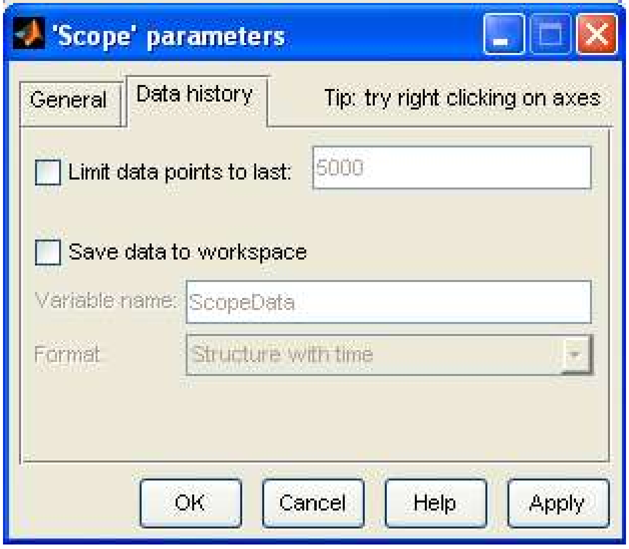
Number of axes: 1 □ floating scope

Time range: auto

Tick labels: bottom axis only

Sampling

Decimation 1


'Scope' parameters

General | Data history — Tip: try right clicking on axes

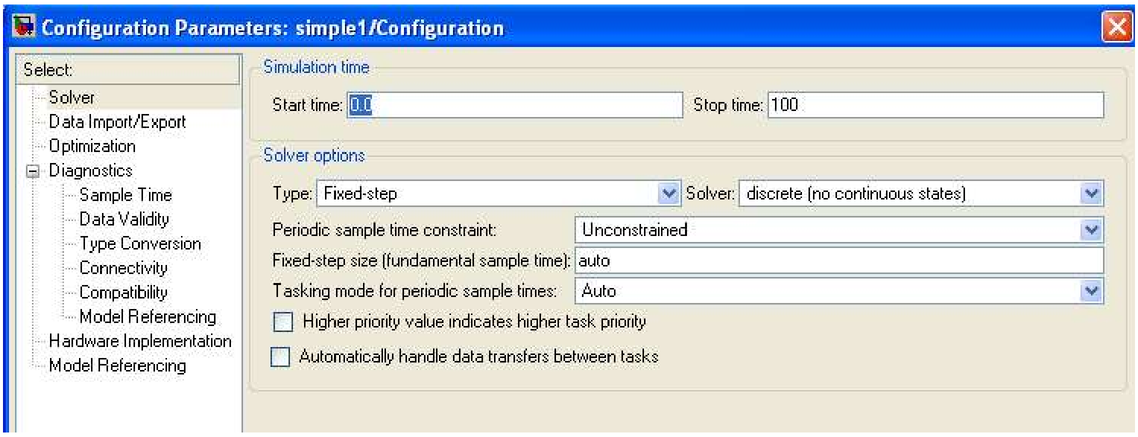
□ Limit data points to last: 5000

□ Save data to workspace

Variable name: ScopeData

Format: Structure with time

Finally, we need to set some global parameters for the simulation. Go to the menu item `Simulation->Configuration Parameters...`. You will see a window open up with lots of options. Just set the parameters to the values shown below:


Configuration Parameters: simple1/Configuration

Simulation time

Start time: 0.0 Stop time: 100

Solver options

Type: Fixed-step Solver: discrete (no continuous states)

Periodic sample time constraint: Unconstrained

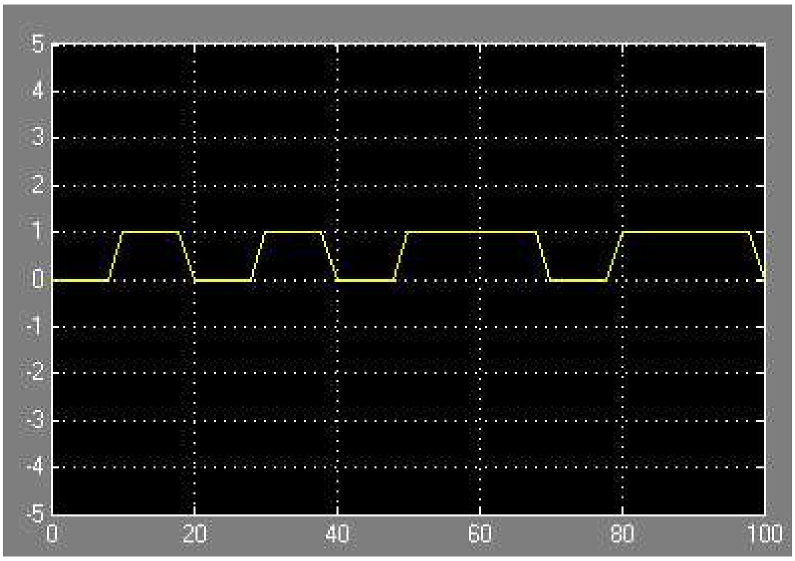
Fixed-step size (fundamental sample time): auto

Tasking mode for periodic sample times: Auto

□ Higher priority value indicates higher task priority

□ Automatically handle data transfers between tasks

We are indicating that it is a discrete-time simulation and we want to run for 100 samples. Note that there is also a field where you can change the number of samples at the top of your model window. Finally simulate the design by clicking the ▶ icon or going to `Simulate->Start`. You should see the output as shown below, which just shows that you are getting your random bits, with one new bit per 10 sample period!

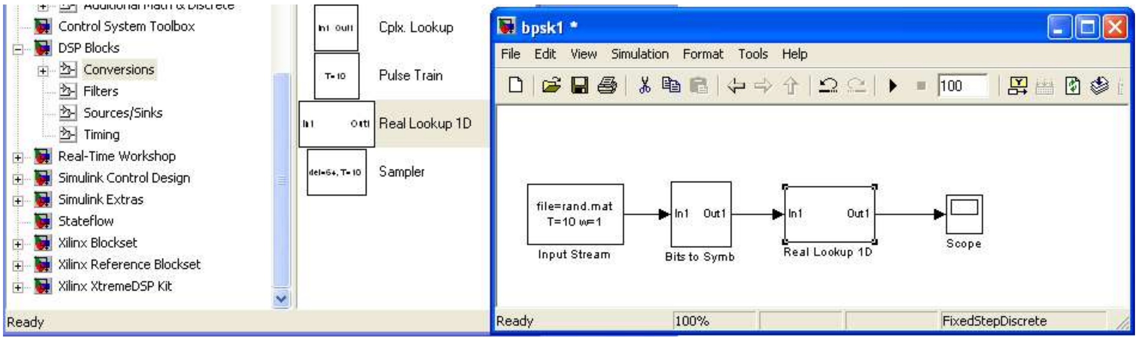


Before leaving this section, save your simple design (maybe as simple1.mdl).

BPSK Symbols

We will start with BPSK modulation, since it only requires us to consider the in-phase (I) component. Save your simple design under a new name (say bpsk1.mdl), before modifying the design.

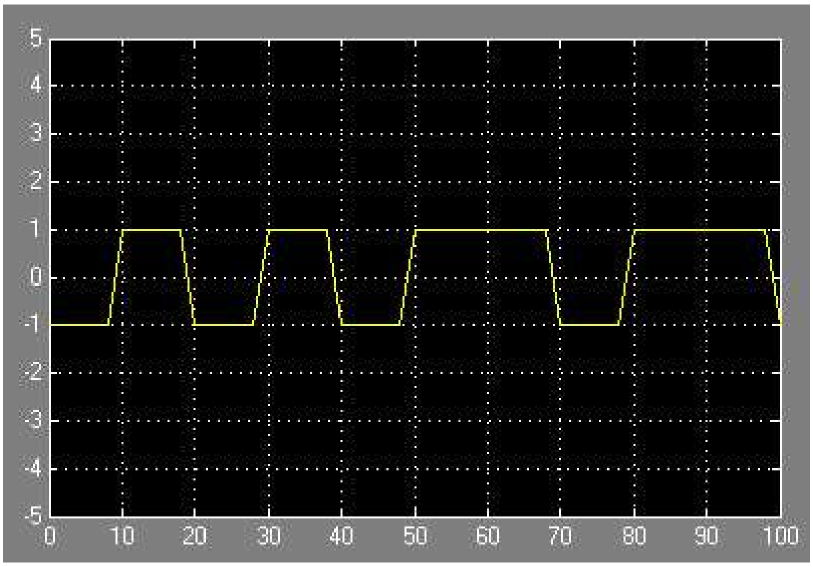
For BPSK, the binary bits 0 and 1 map trivially to s_0 and s_1 which then need to map to I values of -1 and 1, respectively. For this we need a couple more blocks from the `DSP Blocks`library: a bit to symbol mapper and a lookup table. Modify your design so it looks like the diagram below:



For this case, the symbol mapper block doesn’t really do anything, since it just maps bit 0 to symbol index 0, and bit 1 to symbol index 1, but this is the same as just using the bit value. In general, however, the symbol mapper takes a vector of N bits at its input and maps them to the indices 0 to 2^N-1 (LSB first), which will be useful for larger constellations.

Double click on the look-up table block and you will see the single parameter `Table`, which is a vector that contains the outputs you need to generate for each of the symbols. For example, a value of [-1 -0.5 0.5 1] would map 0 to -1, 1 to -0.5, 2 to 0.5, and 3 to 1. For BPSK we just need 0 and 1 to map to -1 and 1, so set Table to [-1 1].

Now resimulate, and you should see something like the following:



When you're done, save your design.

Rectangular Pulses and PSD

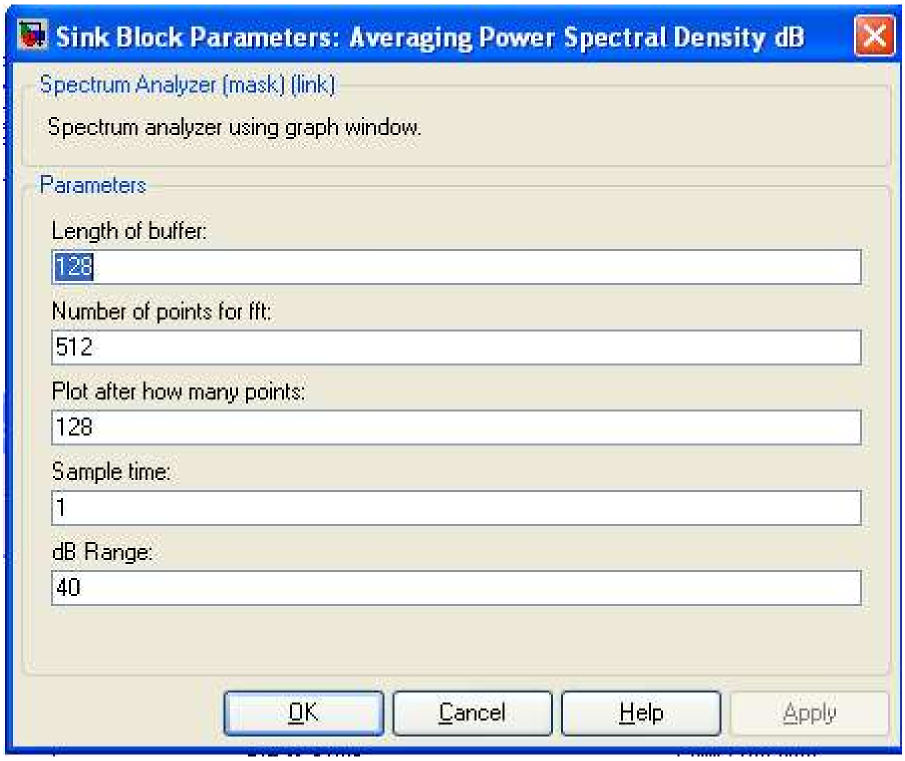
Next, let's look at the power spectral density (PSD) of the BPSK signal we are generating. For a simple communications system, we might consider just sending rectangular pulses like this without any pulse shaping. Let's see what the PSD looks like.

Save your design under the name `bpsk2.mdl` for this next segment. Add the PSD graph from the DSP `Blocks->Sources/Sinks` menu, remove the scope, and connect the PSD graph in its place.

The PSD graph operates by taking FFTs of slices of the input stream and averaging these results, to generally yield a smooth plot showing the spectrum of the signals. If you double click on the block, you will see a number of parameters. The not-so-obvious ones are as follows:

PARAMETER	MEANING
Length of Buffer	Size of the buffer that stores the samples that are operated on by the FFT.
Number of points for fft	The size of the FFT that is computed. Can be bigger or smaller than the buffer length. If it is smaller, FFTs are computed on segments of the buffer and the results are averaged. If the FFT size is bigger than the buffer, the buffer is zero padded and a single FFT is computed.
Plot after how many samples	This controls how many new samples need to come in before a new plot is generated. This also controls the degree of power averaging, because for each plot, the results are averaged with the existing spectrum.

I have found that I get a reasonably smooth curve with the following parameters:


Sink Block Parameters: Averaging Power Spectral Density dB

Spectrum Analyzer (mask) (link)

Spectrum analyzer using graph window.

Parameters

Length of buffer: 128

Number of points for fft: 512

Plot after how many points: 128

Sample time: 1

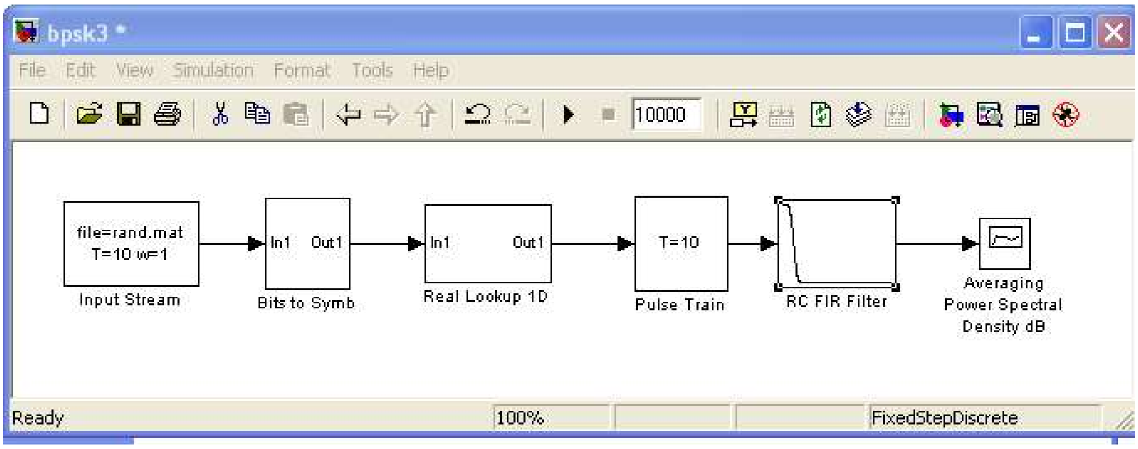
dB Range: 40

We need many more samples to generate a decent PSD, so increase the maximum simulation time to something like 10000, and then simulate your model. (Note: the maximum simulation time can be changed in `Simulation->Configuration Parameters` or by changing the number next to the ▶ icon.

When interpreting your PSD, recall that we have one symbol per 10 samples, or a discrete symbol frequency of 0.1. Efficient modulation can convey this information using a discrete (one-sided) bandwidth around 0.05. Look at your plot. How efficient does the modulation look?

Spectrally-Efficient Pulse Shaping

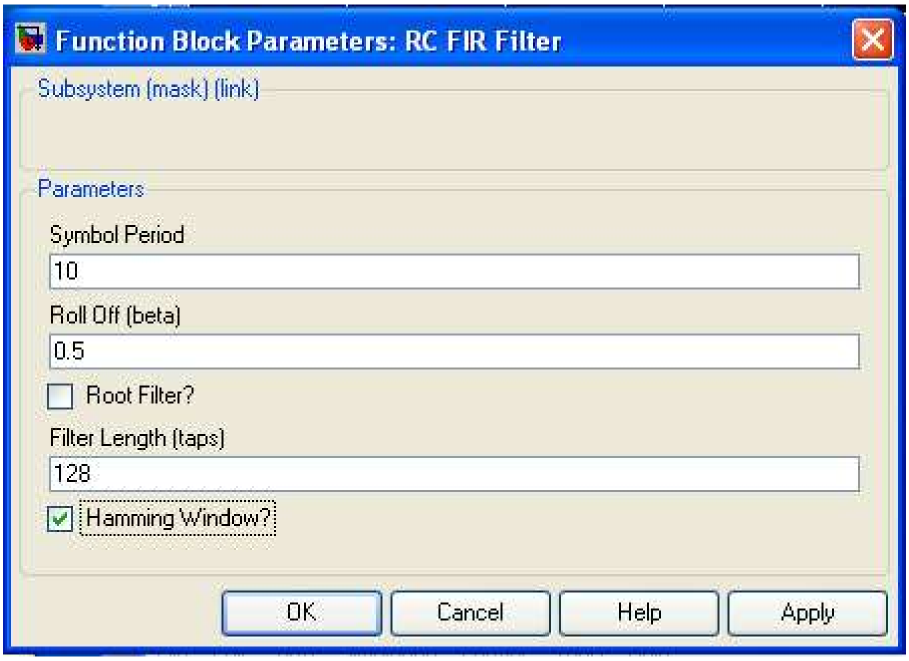
Now let's try introducing pulse shaping to see if this improves our spectral shape. Save your previous design and then save under a new file called `bpsk3.mdl` (I will stop reminding you to do this, but it is always good practice!). You will need the pulse train block from the `DSP Blocks->Conversions` library and the `RC Filter` block from the `DSP Blocks->Filters`library. Your design should look like the following one:



The pulse train block samples its input once per period and generates just a single unit impulse (be sure *T* is set to your symbol period of 10!). Feeding this into the pulse-shaping filter then generates a series of pulses. We will use the raised cosine (RC) filter in this lab which is used extensively in communications systems, where the name raised cosine refers to the shape of the roll-off in the frequency domain (as plotted on the block).

A key parameter of the filter is the roll-off `beta` that controls the bandwidth of the filter. Setting `beta=0` results in a frequency response that is an ideal square shape in the frequency domain. Although this would give the most efficient use of spectrum, the time response (sinc) of the filter is very wide, requiring many filter taps. For `beta=1` we have a pure cosine shape in the frequency domain, which has a short time response, but is also not spectrally efficient. Beta values in between are often used as a compromise between the filter length and the spectral efficiency. Try playing with different values of `beta` and the number of taps and see what happens to the plotted response.

After you try a few values, set the parameters as depicted below:



Function Block Parameters: RC FIR Filter

Subsystem (mask) (link)

Parameters

Symbol Period

10

Roll Off (beta)

0.5

□ Root Filter?

Filter Length (taps)

128

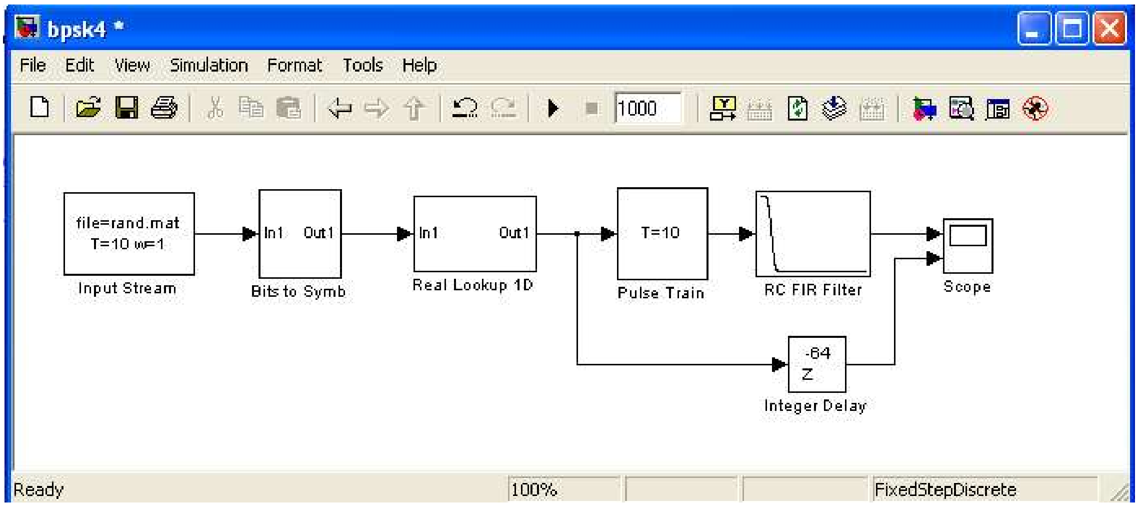
☑ Hamming Window?

Note that the `Hamming Window` option refers to whether the time-domain response is multiplied by the hamming window. This should be used when the number of filter taps is small compared to the length of the impulse response of the filter. For now, just enable it.

Now, rerun your simulation and look at the PSD. How does it compare to the one with rectangular pulses? Note: you may need to increase the dB range to see the plot better.

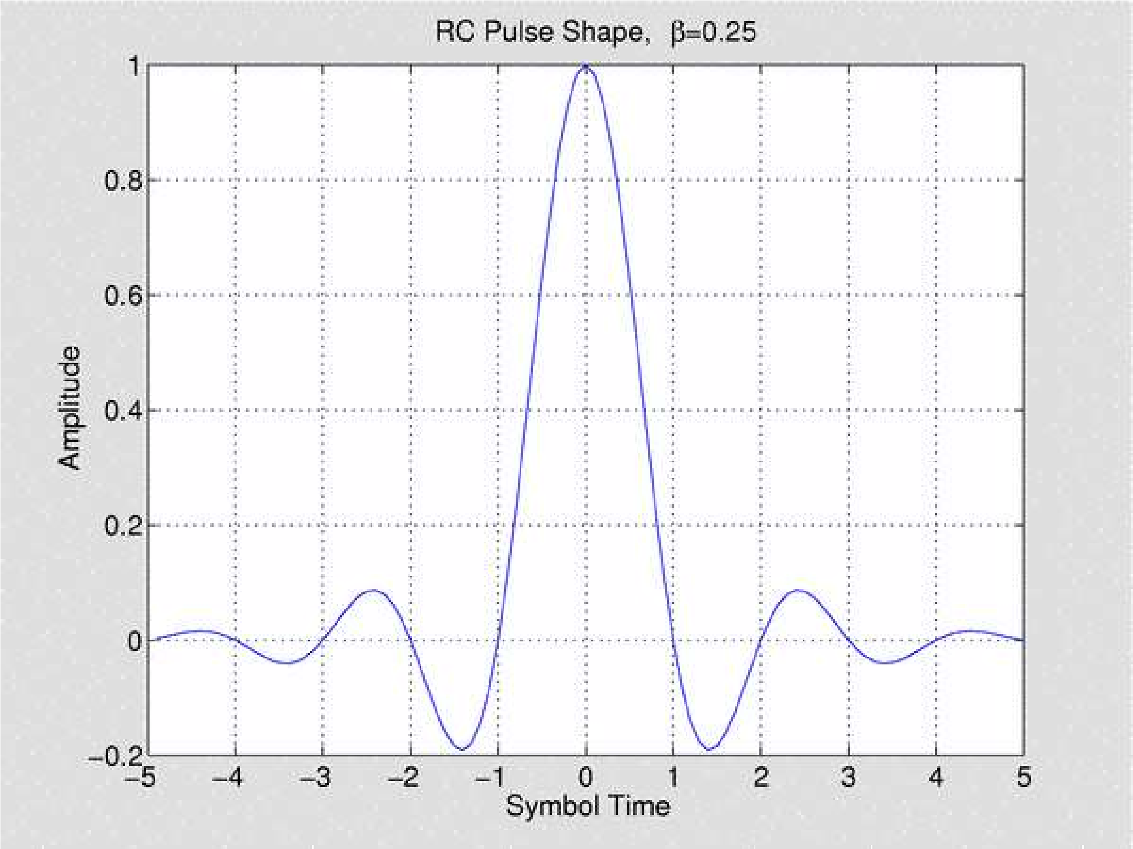
Decoding Pulses

You may have noticed on the time waveform shown on the PSD window for pulse shaping that your transmit waveform now looks modified. To compare the pulses for rectangular and pulse-shaped waveforms, place a scope element in your design as shown below. To make the comparison easier, the extra delay added by the filter (64 samples) is compensated using the delay block (found under `Simulink->Discrete`). Also, I had to change the oscilloscope to have two axes in the parameters to get two inputs.



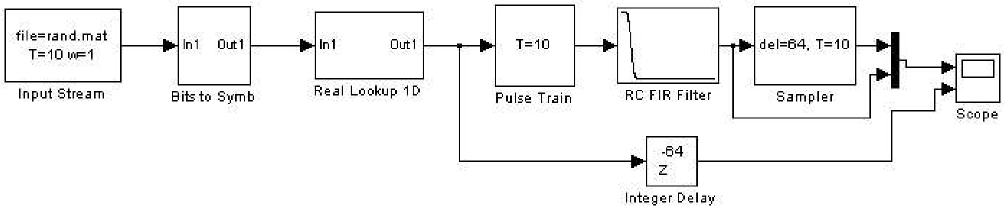
Run the simulation (you will need to reduce the number of samples to something like 1000). What do you notice? It should be clear how you would decode rectangular pulses, such as just sampling them in the middle. How do you decode the RC pulses without getting errors from to the wavy filter response?

It turns out that the RC filter is in a class of filters that have a zero inter-symbol-interference (ISI) property, as long as the waveform is sampled at the optimal point. This occurs because the RC time waveform has nulls (goes through zero) at any integer multiple of the symbol period, as depicted below:



Since the response goes to zero at all integer symbol times, it means if we have multiple pulses that are separated by the sample time, they do not interfere at these zero points when superimposed. So, unlike the rectangular pulses that we could sample anywhere with a symbol period, with the RC pulses, we have to sample these *optimal sample points* where there is no ISI.

To show that this actually works, lets do a simple experiment where we “decode” the RC pulses with a sample and hold block. Change your design so that it looks like this one:

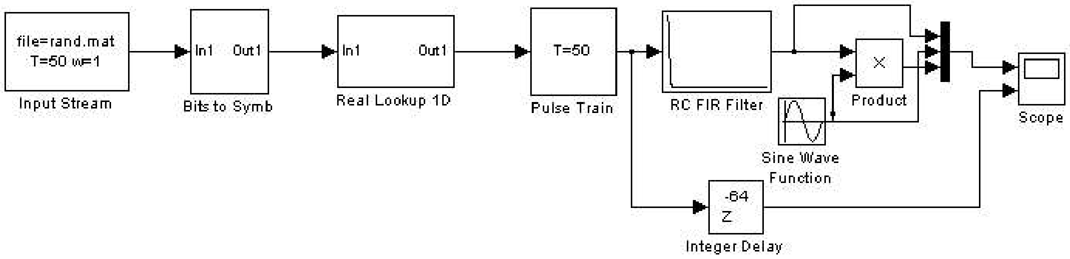


Here, the sampler block was taken from the `DSP Blocks->Data Conversion` library. The multiplexer (the solid bar) combines two signals into a vector signal, which is in the`Simulink->Signal Routing` library. This is useful here to make the two signals appear on the same plot (in different colors). The sampler block just samples the input waveform at regular intervals (in this case the symbol period) after some initial delay (offset). Here the offset of 64 was chosen since it is know that this is how much delay is imposed by the filter. After running the simulation, you should see two plots on top, where one is the incoming waveform, and the other is the sampled (and held) waveform. The bottom plot shows your original digital stream. If everything is set up right, this should convince you that perfect decoding of the signal is possible.

Although we can hard code the known delay in the transmitter like this, what do you think we do at the receiver? The receiver doesn’t know exactly where these ideal sample points are, so what can it do? We will learn in a later lab that a synchronization element called a PLL will be used to find and track these points.

Up-Conversion

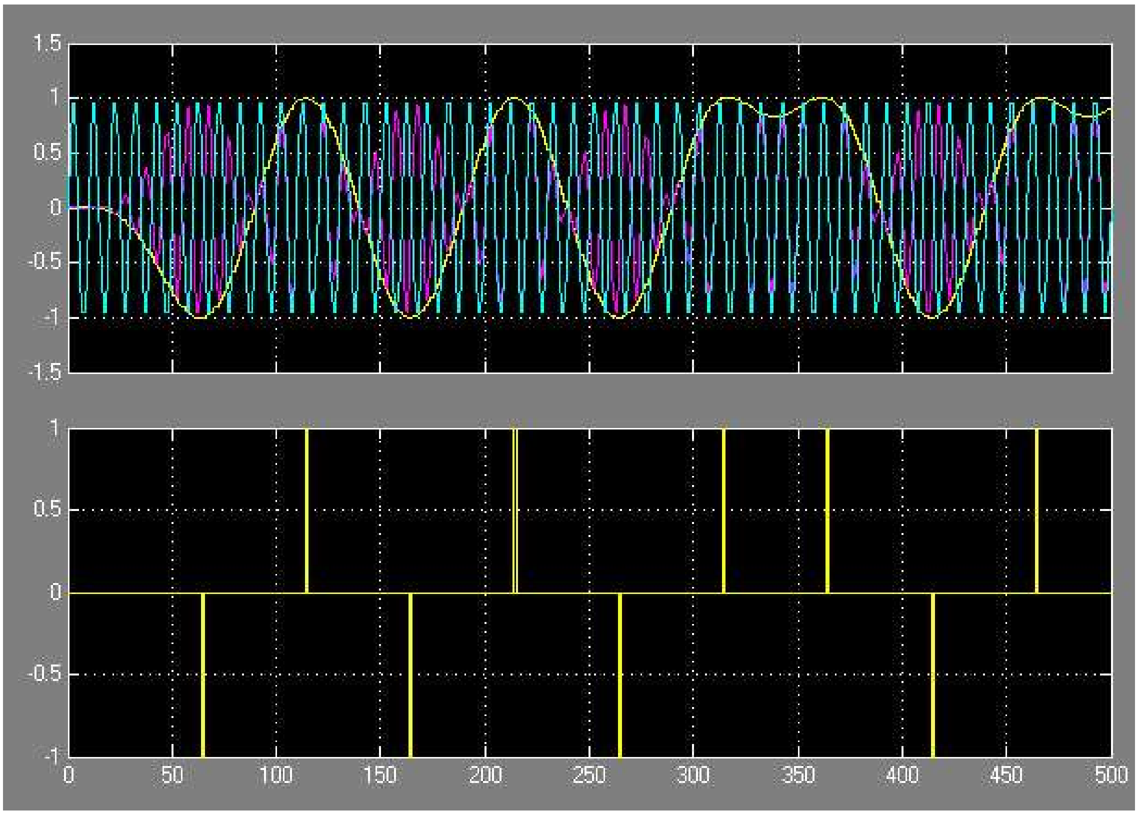
As stated before, usually we will not operate at baseband, but need to have our signal centered about some carrier frequency. Modify your BPSK transmitter as depicted below:



In a typical digital up-converter, the required carrier frequency requires a much higher sampling rate than what is required by the baseband processing blocks (like you have considered so far). Usually, a real up-converter oversamples (interpolates) the baseband signals at a higher rate before multiplying by the carrier. For simplicity in our case, we will just change the sample rate of all the baseband blocks. As you can see I set the symbol period to 50 samples (also in the filter block!).

The sine wave function was taken from the `Simulink->Math Operations` library, and for the frequency use 2*pi*0.1 rad/sample (discrete frequency is 0.1). For our purposes, choose the `time-based` sine wave and for `time (t)` choose the `simulation time`.

After re-simulating, you should get something like the following picture:



In the top plot, you see the baseband signal (yellow), the up-converted signal (purple), and the unmodulated sine wave (blue). As you can see, with BPSK the signal envelope follows the baseband signal and the phase is either in-phase with the carrier or 180 degrees out of phase. With other modulations, other phase relationships with the carrier are also possible.

QPSK Transmitter

Now you are ready to try modifying your design on your own. In this last step, you need to expand your design to do QPSK modulation by adding a parallel branch for the quadrature (Q) component. Some hints:

1. You will need to change the input source block to have an output width of 2, so that you get symbols 0-3.
2. You need to replace the real 1D lookup table with a complex one. For the table, I recommend using [1 j -1 -j], since this will be easier to check than the [1+j, -1+j ...] version.
3. There is a block in the math library for splitting a complex value into real and imaginary parts.
4. Get cos(.) and -sin(.) functions by changing the phase of the sine generator block.

Once you have it put together, simulate it like you did with the last BPSK transmitter with a scope. For example, you could have a scope with 3 axes showing the following information:

1. The up-converted signal multiplied with the cos() carrier, and the unmodulated cos() carrier.
2. Real part of pulse train (delayed)
3. Imaginary part of pulse train (delayed)

You should be able to tell if the modulated signals look correct by comparing the phase of your modulated carrier to the unmodulated carrier.

Check-off

Get the TA and show the following things:

1. Show your complete BPSK transmitter and explain what the different blocks are for.
2. Simulate the BPSK transmitter and explain the signals you are seeing. Explain how you can tell the transmitter is working correctly.
3. Show your complete QPSK transmitter, and explain what you added to support I and Q modulation.
4. Simulate the QPSK transmitter and explain what your are seeing.
5. Answer any additional questions the TA asks.

Lab Write Up

For the write-up, provide the following things:

- Explain in one or two paragraphs what you learned about communications systems that you didn’t know before.
- Show a picture of your final QPSK transmitter. Explain briefly what the different blocks are for. Explain how you checked that it was working correctly. A plot of the output showing important signals would be very helpful.
- Explain why we do pulse-shaping in communications systems, rather than just sending rectangular pulses.
- Tell why synchronization (e.g. sampling the received signal at the right place) is critical in a communications system that uses pulse shaping.
- Describe any difficulties you experienced in getting your design to work and how you fixed these problems. I will be really surprised if everything worked perfectly!