(Approx. 1200 words)

## System Simulation

By Dick Maybach, Brookdale Computer User Group

www.bcug.com

n2nd (at) att.net

Before engineers begin the detailed design of large systems, they usually simulate them at a high level; that is, they enter a functional block diagram into suitable simulation software and measure how it performs. Each block is a mathematical definition of the hardware or software in the final design. A good example of such a simulator is GNU Radio, <https://www.gnuradio.org/>, available free for Linux, Windows, and Mac systems. As you probably suspect, the program is far too complex to cover in this short article. I’ll just introduce it so you have an idea of what it is and drop a few hints about what it can do. You probably aren’t interested in system simulation, but I hope this article will encourage you to look for PC applications that can help you enjoy your own interests. Figure 1 shows the GNU Radio design window.



Figure 1. GNU Radio Design Screen.

At the top is the usual menu bar, with icons for the common tools; below it, you see the workspace where you assemble your system; and at the right is a virtual parts bin, from which you drag components onto the workspace. Note the search window at the top of the parts bin, here set to display only the GUI components, a small portion of all those available. The panel at the bottom displays messages from the program as it operates.

The system shown has only one real component, a signal source. The time sink is an oscilloscope to display waveforms, and the frequency sink is a spectrum analyzer to display spectra. The throttle is present only to reduce the load on the PC during the simulation; it isn’t a part of the system. We’re using a digital computer to simulate an analog system, which it can do only approximately. The simulation begins by computing the state of the system at time equals zero. It next computes how the system will change in a short time (1/32,000 second in this case), and repeats this as many times as desired. The sample rate (32 kHz here) is a key parameter in any digital simulation.

Notice that there are two groups of blocks in Figure 1. Those at the top aren’t connected to anything; instead, they set some needed parameters that I’ll discuss a bit later. Below them are the blocks that define the system, and these are connected just as the real system components will be. After moving the blocks from the parts bin to the workspace, you draw in the connections. What you’re doing here is designing a program. The blocks are subroutines and the connections show how data flows between them. To run the simulation, GNU Radio sends the program you’ve developed to a Python language run-time system, which does the computing.

Let’s return to the blocks at the top. The left one is a title block that identifies this particular simulation. The next one sets the sample rate. Normally this is the same for every block, and defining it here makes it easy to change should that be needed. Finally, the QT GUI Range also defines a parameter, but instead of being constant, it can be varied while the simulation is running. Double-clicking on this box allows you to set its parameters.



Figure 2. QT GUI Range Setup.

As Figure 2 shows, this component sets “freq,” the frequency of the sine wave from the Signal Source. Initially, it’s 2000 Hz, but can be varied from 1000 to 40,000 in steps on 1000. (There is a trap hidden here. The numbers must have valid Python formats. Note that the stop value is 40e3, but in Figure 1, it’s displayed as 40k. The latter is not a valid Python format, and entering it in that form will generate a cryptic Python error message when you run the simulation. Ask me how I know.)

After the design is complete, you select the icon in the center of the menu bar and to the right of the red circle, which creates a Python program that implements the simulation. Run the program by selecting the triangle just to its right to produce Figure 3.



Figure 3. Simulation Result.

The slider bar at the top results from the QT GUI Range block and allows us to vary the frequency produced by the signal source. The sample rate must be greater than twice the highest frequency present in the system, usually considerably greater. Figure 4 shows what happens if this isn’t so.



Figure 4. Too Low a Sample Rate.

Here, we’re sampling a 15 kHz signal at 32 kHz, and the simulation results are garbage. The measured frequency is correct, but the waveform is bizarre. Figure 5 shows a case that even worse.



Figure 5. A Far Too Low Sample Rate.

Sampling a 20 kHz signal at 32 kHz produces both a garbage waveform and a garbage spectrum. The only conclusion you can make is, don’t do this.

Figure 6 shows a less trivial example, an AM transmitter.



Figure 6. AM Transmitter.

Again, the simulation was built by dragging components from the parts bin to the workspace and connecting them. Here too, the boxes that set parameters are located at the top. The Audio Source (on the left) takes its input from the PC’s microphone connector, and the ZMQ PUB Sink (on the right) sends its output to a virtual network within the PC. We can use this to connect an AM receiver, shown in Figure 7, to this same network, which will allow the two simulations to communicate.



Figure 7. AM Receiver.

The ZMQ PUB Source (on the left) connects to the AM transmitter simulation using the internal virtual network, and the Audio Sink (on the right) connects to the PC speaker. When both are running, you can talk into the microphone and hear your voice (delayed) from the speaker. The result appears in Figure 8.



Figure 8. AM Transmitter and Receiver in Operation.

Note that the received signal is delayed from the transmitted one by something over a millisecond. The volumes are controlled by the QT GUI Range modules, and the traces are produced by the GUI QT Time Sinks.

Because the simulations communicate over the virtual network, each can be fairly small, making them quick to design and debug. As I mentioned above, they are implemented as Python programs, meaning the entire simulation is run by starting all the individual programs. In this case, I first started the receiver from the GUI screen of Figure 7 and then started the transmitter from the command line with this sequence. (I’m running this on Linux; the Windows command will be slightly different.)

 Python3 -u AM\_Transmitter.py

The examples in this article, and others, are described on the GNU Radio website, which is a good place to begin learning about the program, and of course, there are articles on the Internet. A related area is Software Defined Radio (SDR), which uses hardware to convert radio signals so their frequencies are low enough that computers can process them. You can learn about this in the book, The Hobbyist’s Guide to the RTL-DSR by Carl Laufer.