Wireless Channel Simulator for Linux

Josef N. Peterson

***Abstract*—Computer simulations of wireless communication channels allow engineers to evaluate design trade-offs before the first bit of circuit design. While simulations can easily be created in MATLAB, open-source high level programming languages, like Python, allow engineers another path to simulate system characteristics. Python offers clear readable syntax, often similar to MATLAB, but at much lower cost and often at higher speed. The language contains complex data objects, like lists and arrays, numerical analysis packages, plotting capabilities, and freely available on-line documentation. It offers a path to graphical user interface (GUI) implementation, using the Gnome Tool Kit (GTK+) and Glade. Coupled with Linux open sound system (OSS) audio, an engineer can rapidly implement a complete wireless channel simulator, from microphone to speaker.**

I. Introduction

THIS report documents the design of a wireless channel simulator implemented with only open-source tools. Some design decisions are implied in the title itself. The software uses open-source resources. Specifically, it uses the Python programming language and Linux open sound system. This report presents the details of the programming and computing resources, modules, and references used to implement the software.

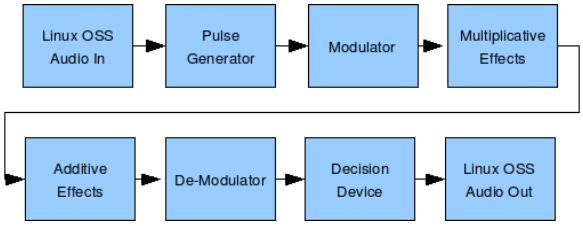

Figure 1: Wireless Channel Simulator Block Diagram

The block diagram for the wireless simulator is shown in Figure 1. It is a simplex channel. The audio input device produces a μ-Law pulse code modulation (PCM) stream, and is discussed further in section II.A. Together, the pulse generator and modulator produce an array of voltage or current values representing the modulated PCM. Multiplicative effects and additive effects include small-scale fading and AWGN respectively. The de-modulator is a simple correlator. The decision device is a simple maximum likelihood estimator. It attempts to recover the PCM stream, which can then be delivered to the Linux audio system. Although the original intent was for the system to operate in near real-time, processing requirements exceeded the amount of time available between samples. The system currently allows a user to record one second of audio, process that audio, then play the result.

J. N. Peterson is a student at the University of Florida Research and Engineering Education Facility, Fort Walton Beach, FL 32542 (phone: 229-834-5704; e-mail: joecool2002@ gmail.com).

II. Channel Simulator Modules

A. Open Sound System Audio Interface

A Python programmer can access the Linux audio system using the ossaudiodev module. To prep the system, the programmer first creates an instance of the ossaudiodev.open() object, then sets the sample rate of the object using its speed method. The programmer can then grab or put an arbitrary number of samples from/in the object. The samples are encoded in mu-law PCM. Mu-law PCM is a non-linear voice encoding, described in figure 1. The AudioInterface class implemented in the channel simulator provides sampleGrab and samplePut methods. These methods provide or receive mu-law PCM in a binary string format, i.e. '10100101.' In contrast, the ossaudiodev.open object will provide/accept simple packed binary, interpreted by Python as a hex format, i.e., staying with the previous bit example, '\x10\ x05'. Python's struct.pack and struct.unpack functions provide an easy way for the programmer to convert between the hex string format and a decimal integer format. The .bin and .unbin method of the AudioInterface class then allow the programmer to convert between the decimal integer format and a string binary format. All classes in the channel simulator that expect binary input accept binary string format.

Quantized μ-law algorithm

13 bit Binary Linear input code	8 bit Compressed code
+8158 to +4063 in 16 intervals of 256	0x80 + interval number
+4062 to +2015 in 16 intervals of 128	0x90 + interval number
+2014 to +991 in 16 intervals of 64	0xA0 + interval number
+990 to +479 in 16 intervals of 32	0xB0 + interval number
+478 to +223 in 16 intervals of 16	0xC0 + interval number
+222 to +95 in 16 intervals of 8	0xD0 + interval number
+94 to +31 in 16 intervals of 4	0xE0 + interval number
+30 to +1 in 15 intervals of 2	0xF0 + interval number
0	0xFF
-1	0x7F
-31 to -2 in 15 intervals of 2	0x70 + interval number
-95 to -32 in 16 intervals of 4	0x60 + interval number
-223 to -96 in 16 intervals of 8	0x50 + interval number
-479 to -224 in 16 intervals of 16	0x40 + interval number
-991 to -480 in 16 intervals of 32	0x30 + interval number
-2015 to -992 in 16 intervals of 64	0x20 + interval number
-4063 to -2016 in 16 intervals of 128	0x10 + interval number
-8159 to -4064 in 16 intervals of 256	0x00 + interval number

Fig. 1: Mu-law Encoding

Usage:

```
from channelaudio import *
aiinstance = AudioInterface(sampleRate = 11025)
sample = aiinstance.sampleGrab()

aiinstance.samplePut(sample)
```

B. Pulse Shaping

The channel simulation program offers two choices for pulse shaping: rectangular or raised cosine. The raised cosine pulses allow the user to specify alpha. A few values are unavailable due to asymptotes in the raised cosine function. For example, alpha = 0.5 results in a module failure. The values of alpha for which faliures occur is not fully characterized. 0 < alpha < .5 and .5 < alpha < 1.0 function correctly. The formula used to generate an individual pulse is from Rappaport, p.287, and is shown below.

$$h_{rc}(t) = \frac{\sin(\frac{\pi t}{T_s})}{\pi t} \frac{\cos(\frac{\pi \alpha t}{T_s})}{1-(\frac{4\alpha t}{2T_s})^2}$$

The raisedCosine.__init__ method converts the four input variables (alpha, period, samples, and numperiods) to object attributes. The raisedCosine.raisedCos method generates an array of numperiods raised cosine pulses. The raisedCosine.run method simply calls the raisedCosine.raisedCos method. The pulse array is used later in the symbol dictionary module, where it is multiplied by a modulated binary string. Examples of the raised cosine output are shown in figures 2-4. The first is an example of raised cosine pulses with varying values of alpha. The second is an example of pulse shaped QPSK. The third is a comparison of the spectrum of a raised cosine pulse with the spectrum of a rectangular pulse.

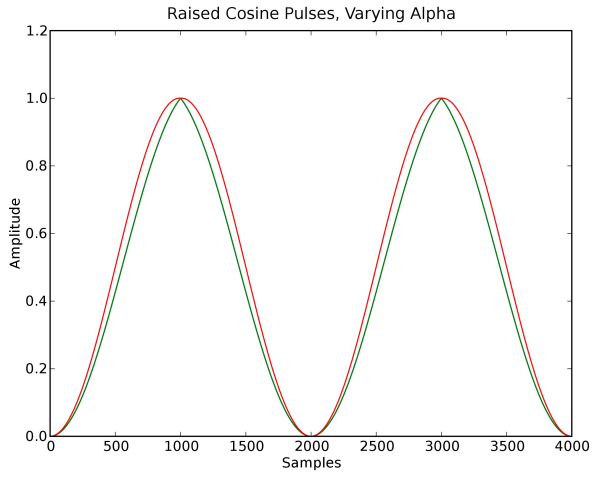

Fig 2: Raised cosine pulses with two different values for alpha.

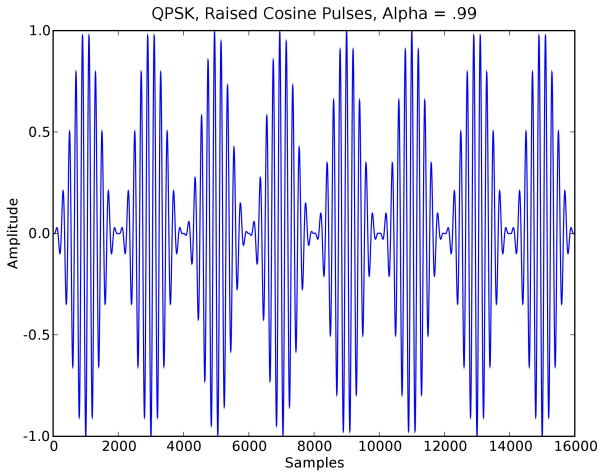

Fig 3: Raised cosine pulse shaped QPSK.

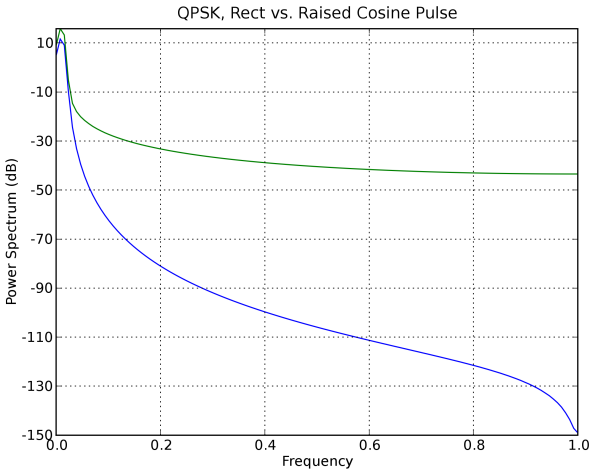

Fig 4: Spectral comparison of raised cosine and rectangular pulses.

Usage:

```
from pulses import *
rcinstance = raisedCosine(alpha, period, samples, numperiods)
outputarray = rcinstance.run()

or

from pulses import *
rectinstance = rect(alpha, period, samples, numperiods)
outputarray = rectinstance.run()
```

alpha: A factor in the raised cosine class that determines rolloff. Unused in rect.
symperiod: The duration of one symbol period
samples: The number of samples per period in the array returned
numperiods: The number of symbol periods in the array returned
outputarray: An outgoing pulse train

C. Modulation Module

The modulation module in the channel simulator allows the user to select from three modulations: binary phase shift keying (BPSK), quadrature phase shift keying (QPSK) and two frequency minimum shift keying (2-MSK). These three modulations exhibit different spectral and bit error rate (BER) characteristics. The module allows the user to set the symbol period (T_s), the number of samples per period (S_s) and the carrier center frequency (f_c). Because of the large number matrix operations carried out throughout the simulator, S_s is calculated in the Channel simulator main class and passed to all submodules. Otherwise, each submodule would need to reshape incoming arrays before executing multiplications, additions, and a few other operations. The amplitude of all modulations is set at $\sqrt{\frac{2E_b}{T_b}}=1$, since the user can arbitrarily set the SNR of the channel by adjusting parameters in the multiplicative (fading) and additive (AWGN) noise modules.

1. Binary Phase Shift Keying (BPSK)

The simulator's BPSK is based on the modulation described in Wireless Communications, page 295-298. The formula for the BPSK waveform is given by:

$$s_{BPSK}(t)=\sqrt{\frac{2E_b}{T_b}}\cos(2\pi f_c t+\theta_c), 0\le t\le T_b$$
$$binary\,1=s_{BPSK}(t)$$
$$binary\,0=-s_{BPSK}(t)$$

The modulation object accepts a string variable representing the binary information to be modulated. For example, an acceptable input to the bpskMod.run(binaryString, [plot]) method is "10010011", or any other combination of ASCII '1' and '0' of any length. Using any other ASCII character will result in an exception in the bpskMod.bpskMod method, specifically a failure to find the character in the lookupdict variable. The bpskMod.run method calculates the length of the string, and calls the bpskMod.bpskMod method $N_{periods}$ times to generate waveform arrays based on the number of samples per period, the symbol period and the carrier frequency. Each array returned by the bpskMod.bpskMod method will have length S_s. The bpskMod.run method uses the Numpy command "hstack" to combine the arrays returned by bpskMod.bpskMod. The resulting array is of length S_s x $N_{periods}$. If the run command is called with the argument plot = True, the method will use Pylab to plot $N_{periods}$ of the returned waveform. This is true for all three modulations implemented in the channel simulator. An example of the BPSK waveform is shown in figure 5.

2. Quadrature Phase Shift Keying (QPSK)

The QPSK modulator is based on the modulation described in Wireless Communications, page 300 – 303. The formula for the QPSK waveform is given by:

$$s_{QPSK}(t)=\sqrt{\frac{2E_s}{T_s}}\cos(2\pi f_c t+(i-1)\frac{\pi}{2}),$$
$$0\le t\le T_s, i=1,2,3,4$$

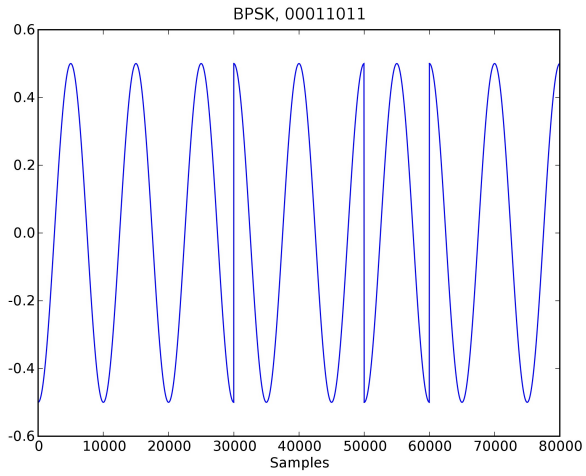

Fig. 5: Example of BSPK modulation generated by bpskMod.run method. In this case the carrier frequency is 10kHz and the symbol period is 1ms.

In this implementation:

Symbol	Phase Offset
"00"	0
"01"	π/2
"10"	π
"11"	3π/2

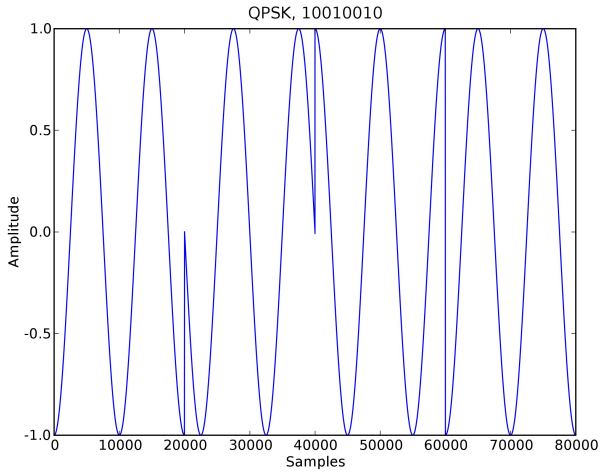

Fig. 6: QPSK modulation generated by qpskMod.qpskMod method. In this case the carrier frequency is 10kHz and the symbol period is 1ms.

The class operates similarly to the BPSK modulation class, with the exception that binary values are passed in two bit chunks to the qpskMod.qpskMod method. Passing a binary string to the qpskMod.run method with length indivisible by two does not result in an exception, but the method will only return arrays representing the first $N_{periods}$ -1. An example of the QPSK modulation output is show in figure 6.

3. Two Frequency Minimum Shift Keying (2-MSK)

The 2-MSK modulator is based on the modulation described in Digital Communications, page 192 – 195. The formula for the 2-MSK waveform is given by:

$$s_{MSK}(t)=\sqrt{\frac{2E_b}{T_b}}\cos(2\pi f_c t+\theta_n+\frac{1}{2}n\pi(-1)^i-1)$$
$$0\le t\le T_b, i=1,2$$

Unlike the two memoryless modulations BPSK and QPSK, MSK requires that the modulator track two pieces of information: the value of the last bit, and the current phase of the signal. When the mskMod.run method calls the mskMod.mskMod method, an array is generated by adjusting the phase of the resulting signal with these two pieces of information. The array is delayed by the mskMod.phase attribute, and then, depending on whether there has been a change in the bit from the previous bit, delayed $\pi/2$ if there *has* been a change.

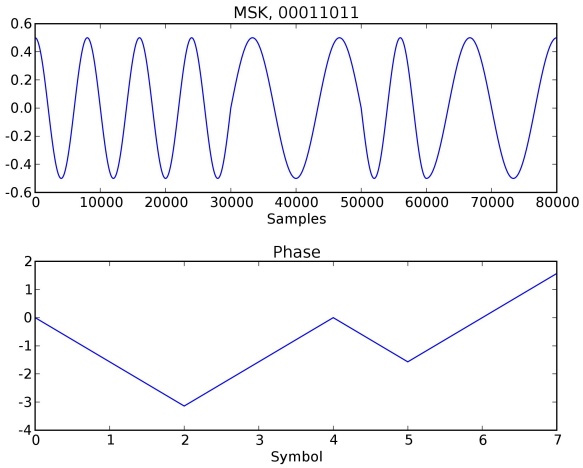

Fig. 7: 2-MSK modulation generated by mskMod.run method. In this case the center frequency is 10kHz and the symbol period is 1ms. The lower subplot shows the signal phase changes.

4. Spectral Performance

The FFT function in the Numpy.fft module allows the user to transform the time-domain signal into a frequency domain representation. Taking the FFT of the three modulations in the wireless channel simulator yields figure 8. The three modulations perform as described in Digital Communications, by Proakis and Wireless Communications by Rappaport. The carrier frequency is set at 50kHz and the modulations are all invoked using the same binary sequence. The 2-MSK main lobe is slightly wider than the QPSK modulation, but the side lobes drop off much more quickly with frequency. The results shown in figure 8 match closely with Figure 4.4-5 in Proakis.

Usage:

```
from modulation import *
mminstance = mskMod(samples, symperiod, frequency)
outputarray = mminstance.run(binaryString)
```

samples: The number of samples of the waveform in one symbol period
symperiod: The duration of one symbol period
frequency: The modulation carrier frequency
binaryString: A binary string input, i.e. '00011011'
outputarray: An outgoing signal array

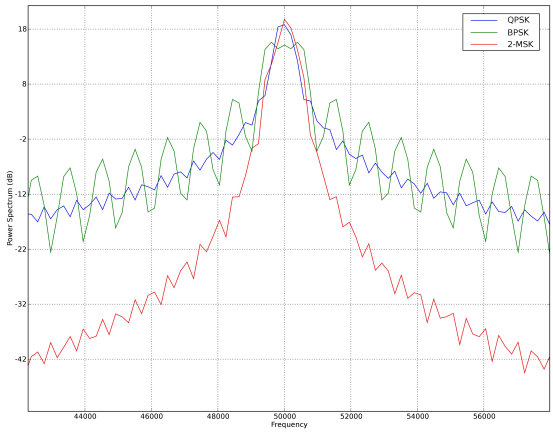

Fig. 8: Spectra of QPSK, BPSK, & 2-MSK generated by the modulation.py module.

D. Symbol Dictionary

While the modulation .run methods do create an array representing a modulated version of an input binary string, their performance is insufficient to be included in the main loop of a real-time audio processing program. Instead, in the channel simulation program, a symbol lookup dictionary is used to produce waveform arrays. Each 8-bit code word is generated and stored in a 256 element Python dictionary that is can be referenced using the 8-bit binary string, i.e. '00011011'.

Use of the symbol lookup dictionary does not introduce any unrealistic characteristics into BPSK and QPSK modulations. However, the 2-MSK modulation is not accurately portrayed by the symbol lookup scheme. A genuine MSK modulation would have coherent phase across all symbols. Using the lookup dictionary introduces a phase discontinuity into successive symbols. The performance difference between using the .run method and lookup table, however, is on the order of 6000. In fact, the average time required by the .run method of the three modulations is 1ms, far exceeding the amount of time between samples from the audio interface, or 90µs. The results of time trials for the

three modulations using their run methods and the symbol lookup scheme is shown in figure 9.

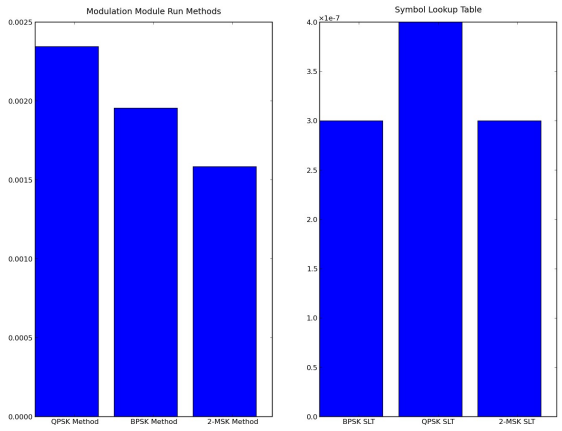

Figure 9: Result of 100K trials of the Run Methods and Symbol Lookup Table.

Usage:

```
from symboldict import *
slt = symbolLookupTable(pulsetype, modtype, samples,
symperiod, alpha, frequency).symbolDict
outputarray = slt[binaryString]
```

pulsetype: Sets the pulse type, 'raisedCos' or 'rect'
modtype: Sets the modulation type 'bpsk', 'qpsk', or 'msk'
samples: Sets the number of samples per period of the modulated waveform
symperiod: Sets the bit period of the modulated waveform
alpha: Sets the alpha for raised cosine pulse shaped waveforms
frequency: Sets the carrier frequency for the modulated waveform

E. Fading

The fading effect is based on the simulation method in Rappaport, p.223 and the two rayleigh fading MATLAB files provided by Dr. Wu. The module has five methods: __init__, dopplerSpectrum, genRandomVar, runFourier, and run. The dopplerSpectrum method creates a doppler spread PSD from the formula in Rappaport, p.218. The genRandomVar method creates an array of complex Gaussian distributed values. The runFourier method multiplies the random array with the doppler PSD, then takes the inverse fast Fourier transform to create time domain Rayleigh fading. An example is shown in figure 10.

To adjust the fading to different channel lengths, power delay profiles, and doppler effects, the __init__ method allows the user to vary the maxdoppler value and the seqlength value. The init method also allows the user to set the period and samples per period of an expected incoming signal (period, samplesperperiod), the samplerate of the Rayleigh fading (samplerate). The maxdoppler value allows the user to generate fast or slow fading, as shown in the figure above. The seqlength value changes two characteristics of the fading: the channel length and the power delay profile. The channel length is set directly by the seqlength value. For a seqlength of 200, period of 1 ms and samplesperperiod of 2000, the channel length is 0.1 ms. The power delay profile is an exponential decay function, and is set at seqlength/samplesperperiod, with a minimum of 1 path. For example, given a seqlength of 200 and samplesperperiod of 2000, the fading module will generate 1 multipath. For a seqlength of 20000 and samplesperperiod of 2000, the fading module will generate 10 mulipath echoes. An example is shown in figure 11.

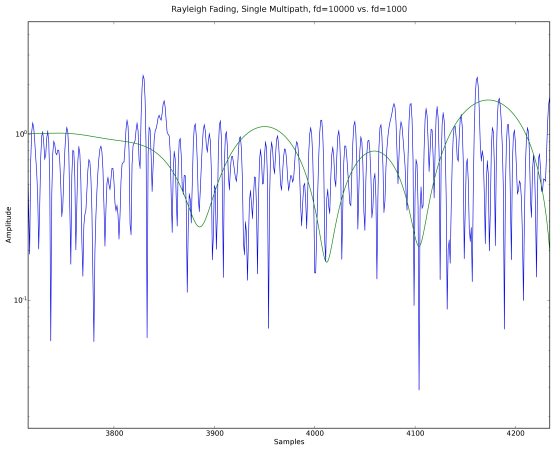

Fig 10: Rayleigh fading in the time domain on a semi-log y scale.

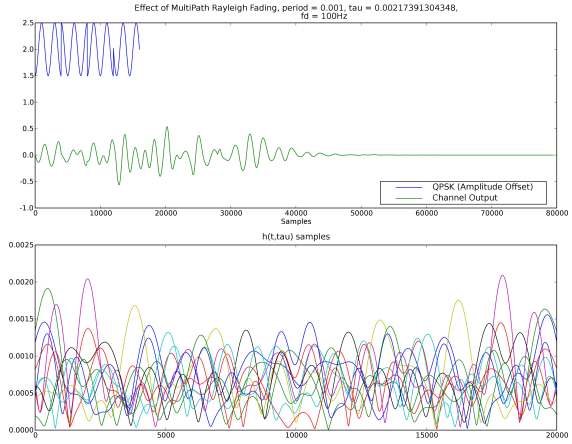

Fig 11: Rayleigh fading method with 10 multipath echos.

Usage:

```
from rayleighfade import *
rfinstance = rayleighfade(period, samplesperperiod,
samplerate, maxdoppler, seqlength)
outputarray = rfinstance.run(inputarray)
```

period: The period of the incoming symbols
samplesperperiod: The number of samples in each of those periods

samplerate: The sampling rate of the generated rayleigh fading.
maxdoppler: The maximum frequency shift due to doppler effects.
seqlength: The length of the PDP decayed to 1%.

F. Additive White Gaussian Noise

The awgnGen module adds AWGN to an incoming signal array. The awgn is simulated by a single, i.i.d. Gaussian random process. The numpy.random module provides a Gaussian distributed random number generator. The awgnGen.__init__ method requires the anticipated period and samples per period of an incoming array to create an object instance. It also requires the desired power of the AWGN. The awgnGen.run method generates an array of random numbers and adds them element-wise to the signal. Five plots of the output of the awgnGen.run method are shown in figure 12 at varying SNR values.

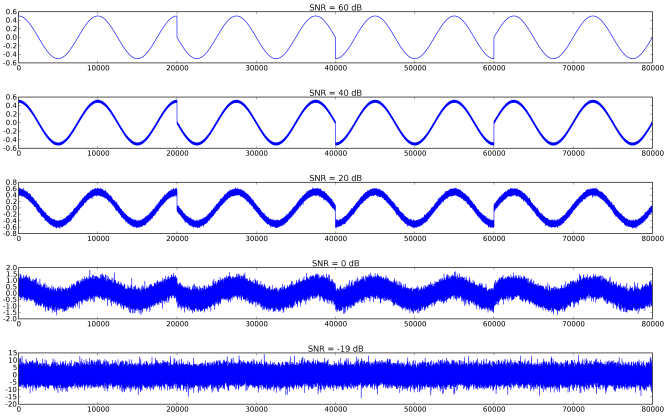

Fig. 12: Output of the awgnGen.run method for varying noise power settings.

Usage:

```
from awgn import *
awgninstance = awgn(period, samplesperperiod, power)
outputarray = awgninstance.run(inputarray, plot = False)
```

period: The symbol period of the expected signal inputs
power: The average power of the awgn.
samplesperperiod: The number of samples in each symbol period
inputarray: An incoming signal array
outputarray: An outgoing signal array, corrupted by AWGN

G. Demodulator

The demodulator module takes the symbol lookup dictionary and an input symbol array (presumably corrupted by noise effects) and attempts to recover the binary string. The module uses a simple maximum likelihood algorithm. The incoming array is correlated with each item in the symbol dictionary. The binary string resulting in the highest correlation value is returned. In the case of 8-bit symbols it executes 256 multiplications and array summations. This results in a fairly high time cost. The module could be improved by doing a bit by bit estimation, but the cost would be that the MSK modulation would no longer perform as if though it is coherently detected. The MSK modulation generated by the symbol dictionary module has coherent phase between bits, but not between multiple symbols. Correlating the received signals with the original stored symbol maintains coherent phase.

Usage:

```
from demodulator import *
demodinstance = demodulator(symboldict)
outputstring = demodinstance.run(inputarray)
```

symboldict: The symbol dictionary used to store the symbol waveforms
inputarray: An input waveform

H. Channel Simulator Main Class

The channel simulator main class ties together and sets up all the other channel simulator modules. The class has six methods: __init__, setupConstants, setupObjects, evalreq, passAudioThruChannel, and run. In contrast with other modules in the channel simulator, the main class must be called in the format instance = channelsim(carrierfreq = 10000). Other modules can simply be called with the syntax: instance = class(10000). The __init__ method creates instance attributes from the input values. The __init__ method uses the evalreq and requirement lists, i.e. faderequals = ['maxdoppler', 'seqlength', 'samplerate'] to ensure that the instance is called with the correct arguments. The setupConstants method sets the samplesperperiod for all symbols and instances in the channel simulator. The samplesperperiod is set at five times the Nyquist sampling rate. The setupObjects method creates object instances of all modules in the channel simulator. If the fading and awgn inputs are set to "True" then it will create instances of rayleighFade and awgnGen objects. PassAudioThruChannel records one second of audio, processes it through the channel, and plays back the result. It will only pass the sample through fading and AWGN channels if fading and awgn boolean values are set to "True", respectively. The run method simply calls the passAudioThru Channel method. Real time audio processing is not implemented in this class, which is discussed further in the real time audio performance section of this report.

Usage:

```
from ChannelSimulator import *
cs = channelsim(carrierfreq = int, symperiod = float, pulsetype = str, alpha = float, modtype = str, fading = bool, awgn = bool, audsamprate = int, maxdoppler = int, seqlength = int, samplerate = int, power = int)
cs.run()
```

carrierfreq, symperiod, pulsetype, alpha, modtype: refer to symbol dictionary usage

maxdoppler, seqlength, samplerate: refer to rayleigh fading usage
power: refer to AWGN usage
fading: True → fading enabled, False → fading disabled
awgn : True → AWGN enabled, False → AWGN disabled
audsamprate: refer to channel audio interface usage

III. Real Time Audio Processing

Processing arrays through the channel simulator requires (relatively) large amounts of time, especially if they are passed through the fading channel. The serial time processing budget for each sample, given an audio sampling rate of 11025 Hz, is only 90µs. AWGN processing, by itself with samples = 200, requires 123µs. The time budget analysis is shown below.

Audio Processing Time Budget

Time Between Samples	90µs
AWGN	(123µs)
Fading	(2876µs)
Modulation & Demodulation	(5831µs)
Net	(8770µs)

Clearly, using this process for simulating the channel in real-time will not yield success. If the sample processing could be pipelined, so that processing for sample N+1 begins before processing for sample N starts, then it may be possible to have additional processing cause only a lag in the input to output audio duration. Multi-threading was attempted to solve the problem, but was unable to compensate for the large time requirements. Python's multi-threading module provides only minimal performance gains for this application. Real-time audio channel simulation may be realizable with a radically different Python architecture, such as the use of DSP type filter implementations and faster C-based data types (in contrast with the numpy arrays), or by implementing the entire program in C. The method described in this report, however, was not successful in meeting the originally desired outcome.

IV. Conclusion

Python, combined with the numpy, scipy, and pylab modules is a powerful substitute for MATLAB. The cost and performance of Python makes it a valuable addition to any engineer's toolbox. However, the speed performance gained is still insufficient to allow implementation of a real-time audio processing system. The ChannelSimulator.py program does allow a user to record audio, pass it through a simulated wireless channel, and listen to the results. However, the initial goal of implementing a real-time wireless channel simulator was not achieved.

References

[1] B. Porat. *A Course in Digital Signal Processing*. New York, NY: John Wiley & Sons, Inc., 1997.

[2] M.C. Jeruchin, P. Balaban, and K.S. Shanmugan. *Simulation of Communication Systems: Modeling, Methodology, and Techniques, 2nd Edition*. New York, NY: Kluwer Academic/Plenum Publishers, 2000.

[3] T. S. Rappaport. *Wireless Communications: Principles and Practices, 2nd Edition.* Upper Saddle River, NJ: Prentice Hall PTR, 2002.

[4] J. G. Proakis. *Digital Communications, 4th Edition*. Boston, MA: McGraw-Hill, 2001.

[5] F. G. Stremler. *Introduction to Communications Systems, 3rd Edition*. Reading, MA: Addison-Wesley, 1990.

[6] H. Stark and J. W. Woods. *Probability and Random Processes with Applications to Signal Processing, 3rd Edition.* Upper Saddle River, NJ: Prentice-Hall, 2002.

[7] M. Pilgrim. *Dive into Python*. http://diveintopython.org, 2004.

[8] Anonymous. *µ-law Algorithm*. http://www.wikipedia.org, 2008.

[9] Anonymous. *Open Sound System – Audio Programming.* http://www.4front-tech.com/pguide/audio.html, 2008.

[10] G. van Rossum. Python Library Reference. http://python.org/doc/2.5/lib/lib.html, 2006.

[11] PyGTK. http://www.pygtk.org, 2008.

[12] NumPy, http://numpy.scipy.org, 2008.

[13] SciPy, http://www.scipy.org, 2008.

[14] Matplotlib, http://matplotlib.sourceforge.net, 2008.