Report Lab 1: Time Synchronization

Emily Tumang and Victoria Preston

February 2016

1 Introduction

In this lab, we used a binary phase-shift keying (BPSK) demodulation method in a wireless communications system in order to transmit data over a distance of 2-3 feet using universal software radio peripherals (USRPs). The demodulation process involved using a fast Fourier transform (FFT) methodology as a first pass, then implementing a phase-locked loop, or Costas loop, to improve the signal quality. We were able to successfully demodulate a complex signal with an analytical bit error rate between 1.2e-4 and 1.4e-4 for a data transmission rate of 50 kilobits per second.

As an extension of this project, we also implemented a quadrature phase-shift keying (QPSK) demodulation scheme and transmitted complex signals over 3 feet with a data rate of 25kbps. We were successfully able to demodulate the received complex signal.

2 Algorithmic Explanation

For the purposes of this lab, a signal was transmitted, received, and the time synchronization applied in post-processing. To do this, the received signal was discretized, as discussed in class:

$$y[m] = (hx[m])e^{-j\phi[m]} + n[m]$$

in which $y[m]$ is the received signal, h is the channel characteristic, $x[m]$ is the transmitted signal, $n[m]$ is noise, and $\phi[m]$ is the phase shift. For the sake of illustration, we ignore noise in our demodulation scheme. In order to remove the effect of the phase shift which occurs between transmission and reception, we will need to identify the frequency at which the shift is. In order to do this, we raise $y[m]$ in polar coordinates to the fourth power and apply an FFT in order to identify the location of the unwanted frequency, which will be described as an impulse and four times the frequency shift:

$$\begin{aligned} y[m]^4 &= h^4 e^{-4j\theta_m} e^{-4j\omega_m m} x[m]^4 \\ &= -4h^4 e^{-4j\theta_m} \delta(\Omega - 4\omega_m) \end{aligned}$$

Solving for ω_m will provide the frequency offset of interest, which can then be removed from the received signal by multiplying the received signal by the complex frequency offset.

To further improve upon this method, we can use a Phase-Locked Loop, or Costas Loop, to further tune the correction offset. In this approach, we calculate an error term for each discrete point, then use a proportion of the error in the offset over many samples in order to best respond to the received signal. This can be summarized as the following process in which we create a simplified received signal term, solve for the error term, and use this to identify a phase offset:

$$\begin{aligned} v[m] &= x[m]e^{-j(\psi[m]-\hat{\psi}[m])} \\ e[m] &= R(v[m]) \times I(v[m]) \\ &= -(\psi[m] - \hat{\psi}[m]) \\ \hat{\psi}[m+1] &= \hat{\psi}[m] + \beta e[m] \end{aligned}$$

By tuning the value of β, the loop can "lock in" more effectively to the true phase offset of the system. An α term related to the total error over the signal can also be applied over time for even finer tuning.

For a QPSK system, the FFT method still holds, but the Costas loop to refine the demodulation undergoes a slight adjustment, in that the error term must be expressed as:

$$e[m] = a \times sign(R(v[m])) \times I(v[m]) \times b \times sign(I(v[m])) \times R(v[m])$$

in which a and b are constants $\pm\frac{1}{\sqrt{2}}$ as a manifestation of the relationship of the quadrature parts.

3 Experimental Implementation

For this lab, we used USRPs from Ettus Research. The interface we selected for these radios was provided by GNURadio. This eased the logistics of transmission and reception of signals which we designed. Our signal processing was done using numpy and scipy which are specialized computation libraries and packages in the Python language.

We transmitted over the 2.4855GHz band, placing 20dB of gain on both the receiving and transmitting antennas. Our transmitted sample frequency was 250e3 with an ultimate bit rate of 50e3 bits per second. The USRPs were placed between 2 and 3 feet during data collection.

3.1 Transmission

For the BPSK implementation, we used Matlab to generate a random vector of ± 1 with a sample length of 40 which was repeatedly transmitted over the USRPs. The code for this was a slightly modified version of Siddhartan's vector generation script; for this reason, it will not be included here. It is interesting to note that although we initially used numpy to generate this vector, GNURadio was unable to parse the numpy vector formatting accurately, causing the transmitted vector to have twice the desired amplitude. This caused a distortion in the received signal due to saturation, which caused the demodulation process we used to fail because the method assumes that the signal has been convolved with a cosine. The results of this are shown in figure 1.

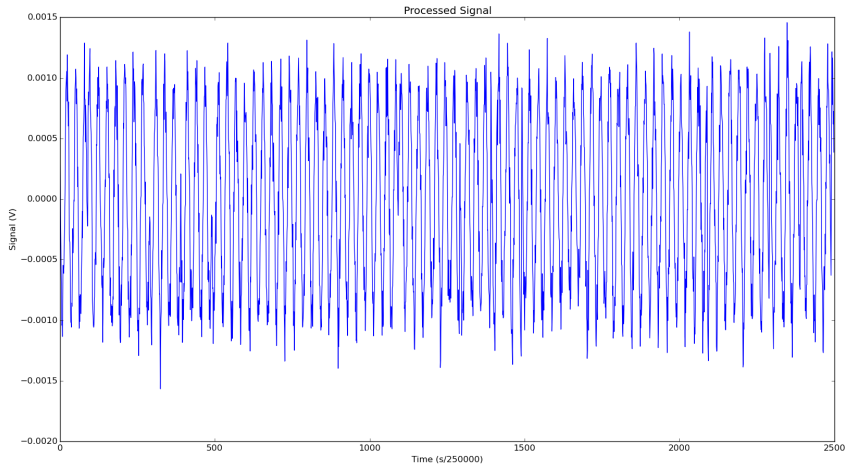


Figure 1: Processed saturated data. Obviously, there is no clear signal which can be distinguished in the data. There is also a cosine which can be seen in the background of the data, indicating that the method of phase shifting has failed to accurately detect or resolve the needed time synchronization.

For the QPSK implementation, we also used MatLab to generate random signals of $\pm 1, \pm i$. Our real and imaginary signals were mirrored.

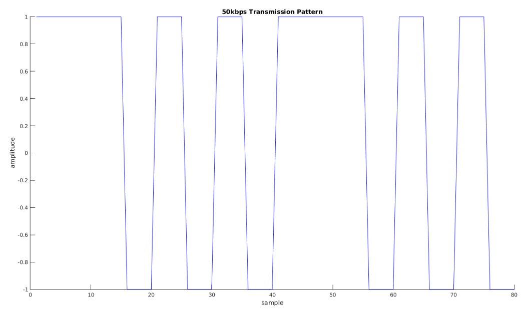
3.2 Reception

In order to generate an initial estimate of the error, we took the Fourier transform of the received signal raised to the fourth power which we discussed earlier. This allowed us to isolate the initial phase shift. We then refined this estimate on a sample-by-sample basis using a digital Costas Loop. The key elements of this implementation are provided below. For the entirety of our implemented code, we have provided an Appendix.

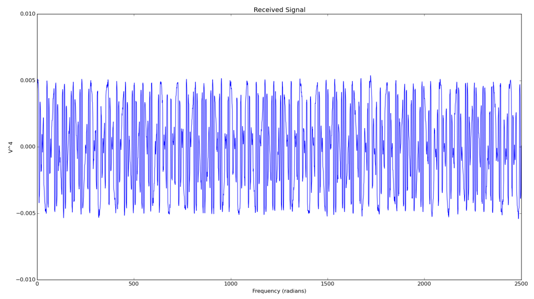
```python
def fourth(data):
  transform4 = fft.fft(numpy.power(data,4))
  freq = fft.fftfreq(len(data),SAMPLE_RATE)
  freq = [f*2*numpy.pi for f in freq]#convert frequency to radians
  impulse = (numpy.argmax(transform4))#get the frequency at which the impulse occurs
  return freq[impulse]

#apply offset frequency and use PLL
def process(offset, data, time):

  #PLL constants
  beta = 0.2
  alpha =  0
  a = -1./numpy.sqrt(2)
  b = -a

  #PLL storage
  total_error = 0
  pll_res = []

  #Use fft to do initial correction
  offset = offset/4 #offset is raised to the 4th, so when converted it is *4

  for index in range(0, len(data)-1):
    pll_res.append(data[index]*numpy.exp(-1j*offset*SAMPLE_RATE))
    error = (b*(numpy.sign(pll_res[index].imag) * pll_res[index].real)) * (a*(numpy.sign(
    pll_res[index].real)*pll_res[index].imag))
    # error = pll_res[index].real*pll_res[index].imag
    total_error += error
    offset = offset + beta*error + alpha*total_error

  return numpy.asarray(pll_res)
```

4 Results

Using a 50 kilobit per second data transmission rate, we transmitted a vector that can be seen in figure 2a. We received a strong - if illegible – signal for processing as seen in figure 2b.



(a) The transmitted signal, repeated twice to illustrate the pattern.



(b) Original signal as received by the USRP. It is clear that there is some sort of pattern in the data, but there is both noise and phase shift that will need to be accounted for.

Figure 2: The raw transmission and received signals.

We were able to find the initial phase shift by taking the FFT of the data to the fourth power, as shown in figure 3.

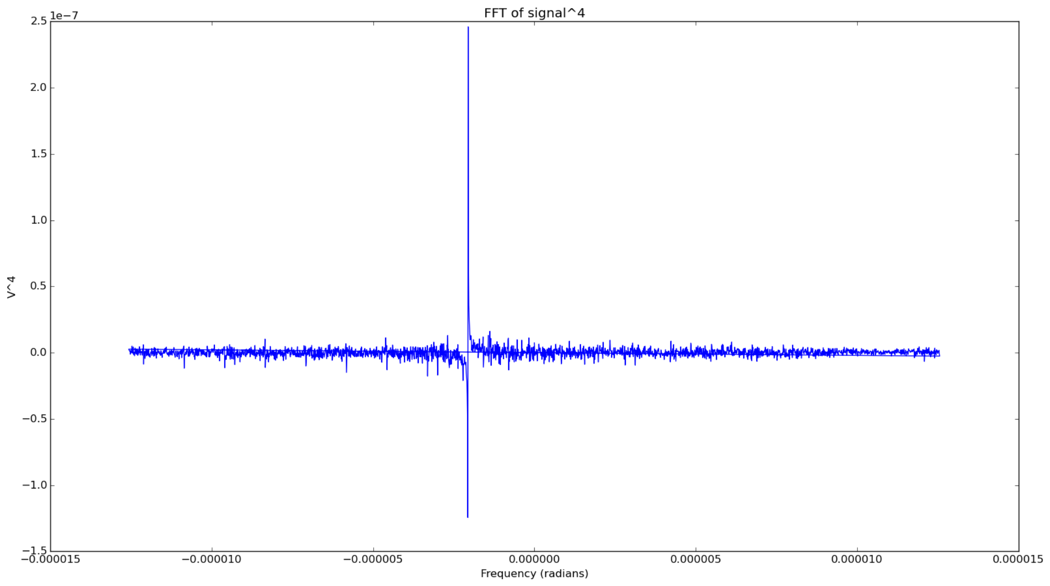


Figure 3: The FFT of the data raised to the fourth, showing a impulse at the location of the phase shift.

Using this information our BPSK synchronization algorithm was able to produce the results shown in figure 4.

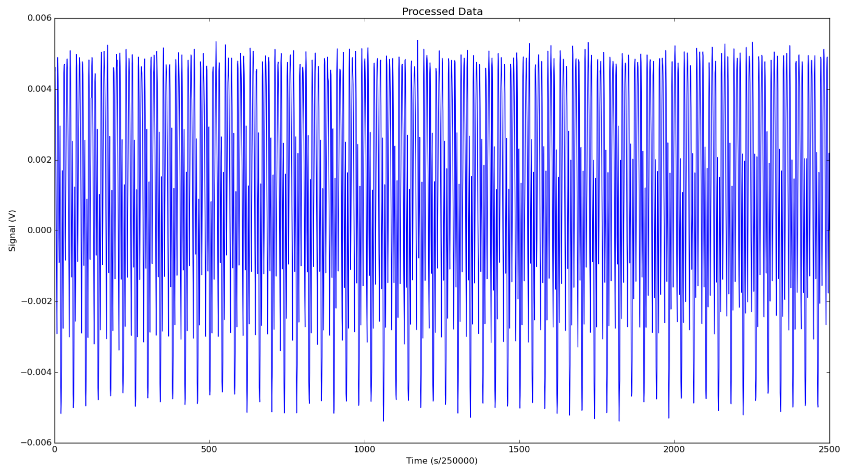


Figure 4: All samples post-synchronization using FFT and Costas Loop implementation, $\beta = 0.1$.

As shown in figure 5, this is very similar to the transmitted data in figure 2a.

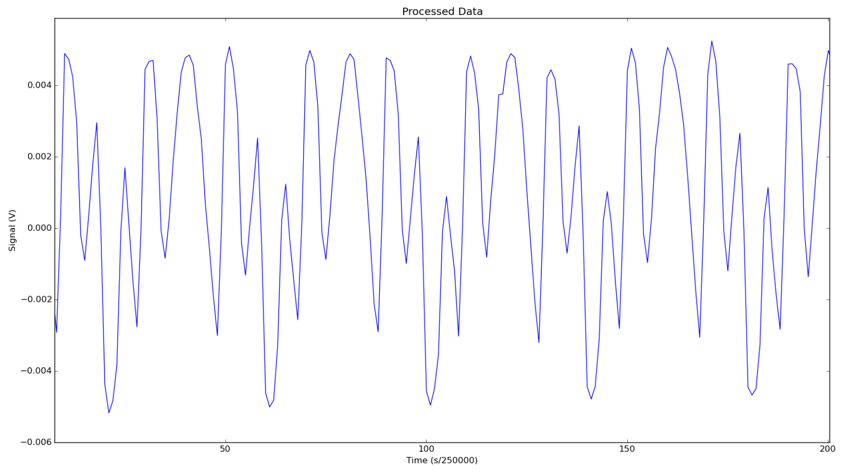
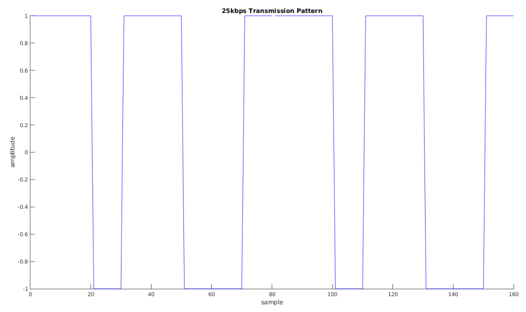
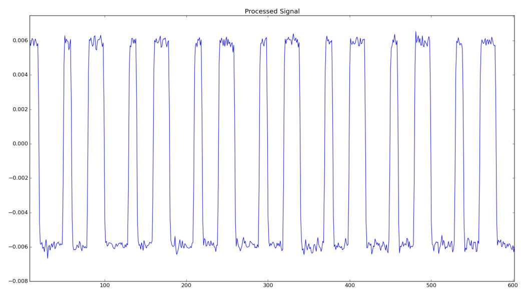


Figure 5: A small fraction of the transmitted data post-processing. A comparison with the transmitted signal shows that the signal is inverted. Knowing the transmitted pattern would allow us to “flip” the results intelligently.

We were able to repeat these results for a data rate of 25 kbps, as shown in figures 6b and 6a.



(a) The transmitted signal with data rate of 25kbps.



(b) The processed signal.

Figure 6: The raw transmission and received signals.

In both of these scenarios, the bit error rate for the communications channel was calculated to be between 1.2e-4 and 1.4e-4 using the relationship:

$$p_{error} = Q(A/\sigma)$$

in which A was estimated as the average of all pulses registered as positive readings (1s) and σ was the root-mean-squared (RMS) value of the channel noise, which is roughly Gaussian and was collected over several seconds with no incoming transmissions.

For our QPSK implementation, the results of our demodulation can be seen in figure 8b as compared with the complex transmitted signal in figure 7.

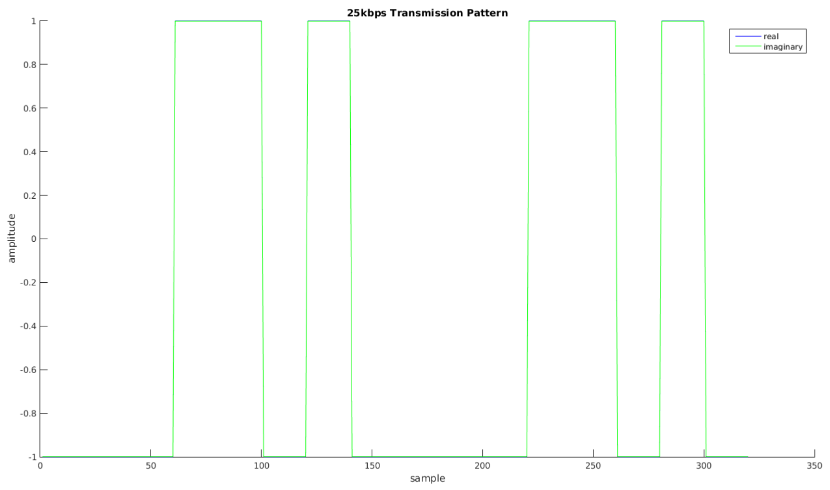
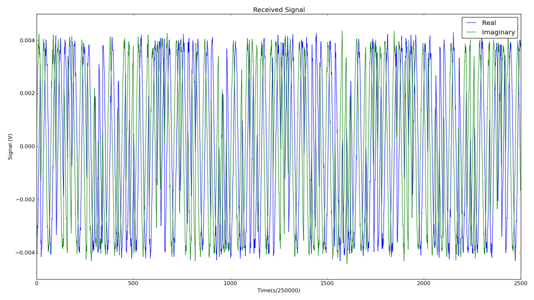
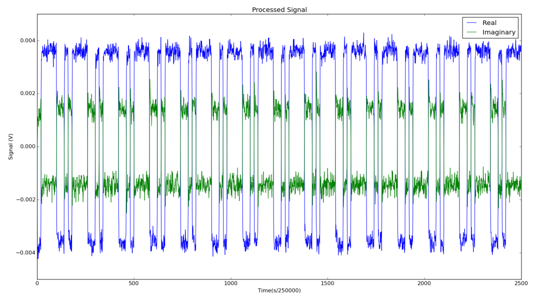


Figure 7: The complex signal transmitted to test QPSK. It was a mirror in both real and imaginary space.



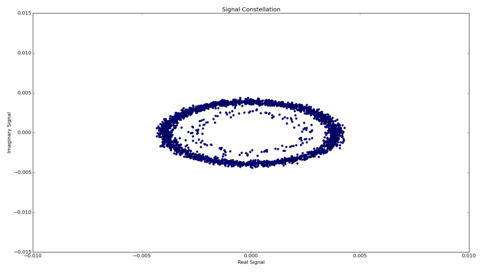
(a) The received complex signal.



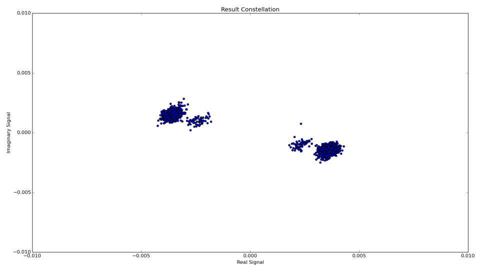
(b) The complex signal, post-processing.

Figure 8: The complex signal before and after processing.

We are uncertain why the imaginary signal's amplitude is so much lower than that of the real signal. We suspect that some of the imaginary signal may have shifted and been processed as real, since the received signal was all over the spectrum as shown in figure 9.



(a) The received signal constellation. As expected, it is very noisy.



(b) The constellation of the processed data,, clearly showing separate transmitted signals.

Figure 9: Constellation for the complex signal as received and after processing

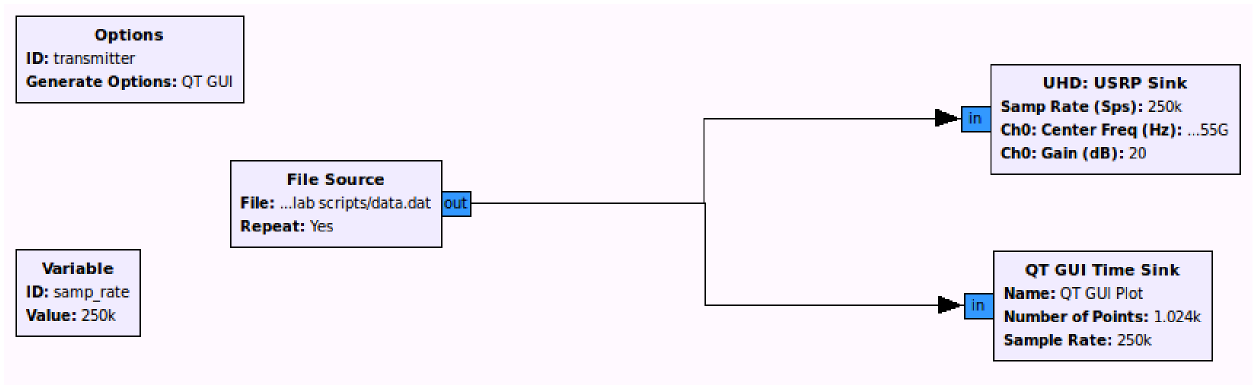
5 Conclusions

We found that the combination of a Fourier approach and a Costas Loop works well for time synchronization of a signal that has been multiplied by a known cosine, as in this lab. Through this, we gained a better understanding of the mathematics of time synchronization and the mechanical effects of phase shift. Given the extension, we were able to explore QPSK techniques in addition to the BPSK, which we found allowed us to think more critically of complex space and trignometric relationships. This lab had a non-trivial equipment onboarding process which took some amount of time to learn. Choosing to use GNURadio and Python particularly aligned with our learning goals for this lab in which we hoped to use familiar opensource computational tool on a new piece of hardware.

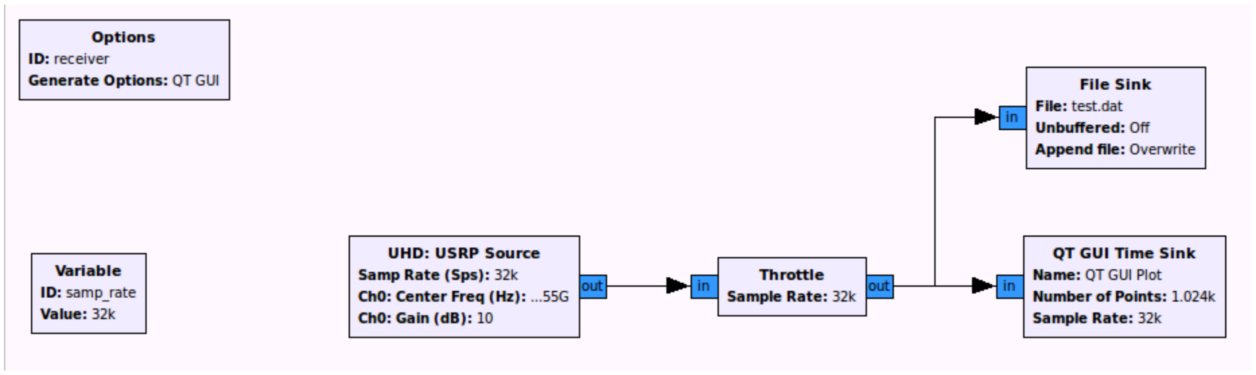
6 Appendix

6.1 GNU Radio

We used GNU Radio and gnuradio-companion to generate Python interfaces for the USRPs. Though the vectors that were transmitted were self-described, GNURadio handled the actual transmission and reception process through executable Python scripts.



(a) Data transmission flow diagram. The "Vector Source" block was used to read in pregenerated binary vectors.



(b) Data reception flow diagram.

Figure 10: Flow diagrams created in GNURadio Companion.

6.2 Processing Code

To access our entire code base for the project, see `https://github.com/vpreston/wirelesscomms_timesyc`.

```python
import numpy
from numpy import fft
import matplotlib.pyplot as plt
import scipy
from scipy import special

SAMPLE_RATE = 250e3#samples/second
TRANSMIT_FREQ = 2.4855e9*2*numpy.pi

data = []
real = []
imag = []

#get and process data
def open(filename='GRC scripts/Data/imag_5ps.dat'):
  data = numpy.fromfile(filename, dtype = 'float32')[5000:10000]
  real = data[0::2]
  imag = data[1::2]
  data = real+1j*imag

  t = numpy.linspace(0, len(data), len(data))
  return data, t

#find offset frequency
def fourth(data):
  transform4 = fft.fft(numpy.power(data,4))
  freq = fft.fftfreq(len(data),SAMPLE_RATE)
  freq = [f*2*numpy.pi for f in freq]#convert frequency to radians
  impulse = (numpy.argmax(transform4))#get the frequency at which the impulse occurs
  return freq[impulse]

#apply offset frequency and use PLL
def process(offset , data, time):

  #PLL constants
  beta = 0.3  #0.1 for BPSK, 0.2 for QPSK on slower data rate
  alpha =  0
  a = -1./numpy.sqrt(2)
  b = -a

  #PLL storage
  total_error = 0
  pll_res = []

  #Use fft to do initial correction
  offset = offset/4 #offset is raised to the 4th, so when converted it is *4

  for index in range(0, len(data)-1):
    pll_res.append(data[index]*numpy.exp(-1j*offset*SAMPLE_RATE))
    error = (b*(numpy.sign(pll_res[index].imag) * pll_res[index].real)) * (a*(numpy.sign(
    pll_res[index].real)*pll_res[index].imag))
    # error = pll_res[index].real*pll_res[index].imag
    total_error += error
    offset = offset + beta*error + alpha*total_error

  return numpy.asarray(pll_res)

def error_calc(data):

  noise = numpy.abs(open('GRC scripts/Data/noise.dat'))
  sigma = numpy.sqrt(numpy.mean(numpy.power(noise,2)))
  A = numpy.mean([x >.005 for x in data])

  error =  0.5-0.5*special.erfc((A/sigma)/numpy.sqrt(2))

  return error

if __name__=='__main__':

  data, time = open()
  offset = fourth(data)
  res = process(offset,data, time)
  print error_calc(res)

  # plt.plot(res)
  plt.plot(res.real)
  plt.plot(res.imag)
  plt.legend()
  plt.title("Processed Signal")

  plt.show()
```