ECE 5650/4650 Computer Project #2: DSP in GPS Signal Acquisition and Tracking

This project is to be treated as a take-home exam, meaning each student or student team of two is to due his/her own work without consulting others. The grading for this third computer project will be handled differently than the first two. A separate grade category exits for this project, thus allowing this project to count up to 20% percent of the final grade (details given below). **The project due date is 5:00 PM Wednesday, December 18, 2019 (Final Exam week).**

The grading options are as follows: (1) (current syllabus) Computer project #2 20%, final exam 25%; (2) Computer project #2 15%, final exam 30%. The other grade distribution percentages remain the same as the syllabus sheet discussed the first day of class. Computer project #1 is integrated into the homework average.

Introduction

In this Python-based computer simulation project you will get a taste of digital signal processing found in the code acquisition and tracking functions of a global positioning system (GPS) receiver.

GPS was started in 1973 with the first block of satellites launched over the 1978 to 1985 time interval[1]. The formal name became *NAVSTAR*, which stands for NAVigation Satellite Timing And Ranging system, in the early days. At the present time there are 31 GPS satellites in orbit. The

Notice that the orbits are inclined in six orbital planes, so the nominal design of 24 total satellites means there are 4 satellites per orbital plane.

NAVSTAR = Navigation System Using Timing and Ranging

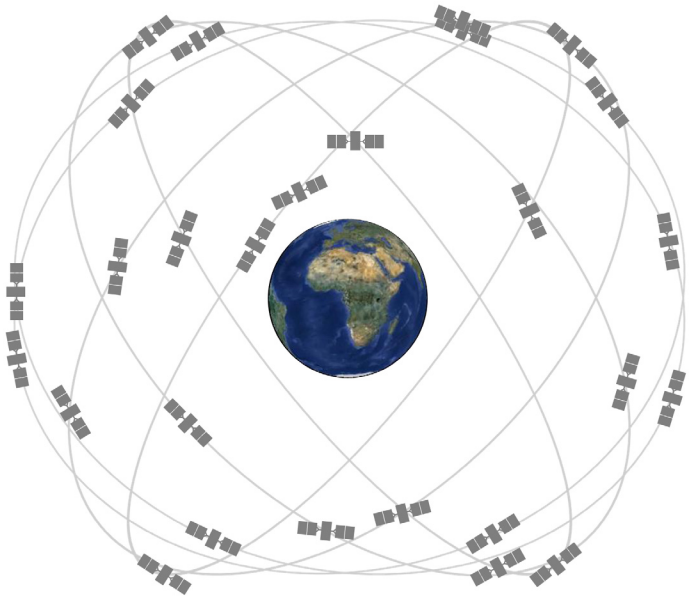


Figure 1: A pictorial of the Navstar-GPS constellation.

original design called for 24 satellites, commonly referred to as *space vehicles* (SVs). The satel-

1. https://en.wikipedia.org/wiki/Global_Positioning_System

lites orbit at an altitude of about 20,350 km (~12,600 mi). This altitude classifies the satellites as being in a medium earth orbit (MEO), as opposed to low earth orbit (LEO), or geostationary above the equator (GEO), or high earth orbit (HEO). The orbit period is 11 hours 58 minutes with six SVs in view at any time from the surface of the earth. Clock accuracy is key to the operation of GPS and the satellite clocks are very accurate. Four satellites are needed for a complete (x, y, z) position determination since the user clock is an uncertainty that must be resolved. The maximum SV velocity relative to an earth user is 800m/s (the satellite itself is traveling at ~7000 mph), thus the induced Doppler is up to 4.2 kHz on the L1 carrier frequency of 1.57542 GHz. This frequency uncertainty plus any motion of the user itself, creates additional challenges in processing the received GPS signals.

Background Theory

In this project the focus is on DSP as found in a GPS receiver. Specifically GPS uses unique ranging codes from each SV to ascertain the distance, r, between a particular SV and the user. With three range measurements and perfect timing, you can arrive at the exact location to within an ambiguity[1] as shown in Figure 2. The ambiguity is resolved by choosing the solution closest to

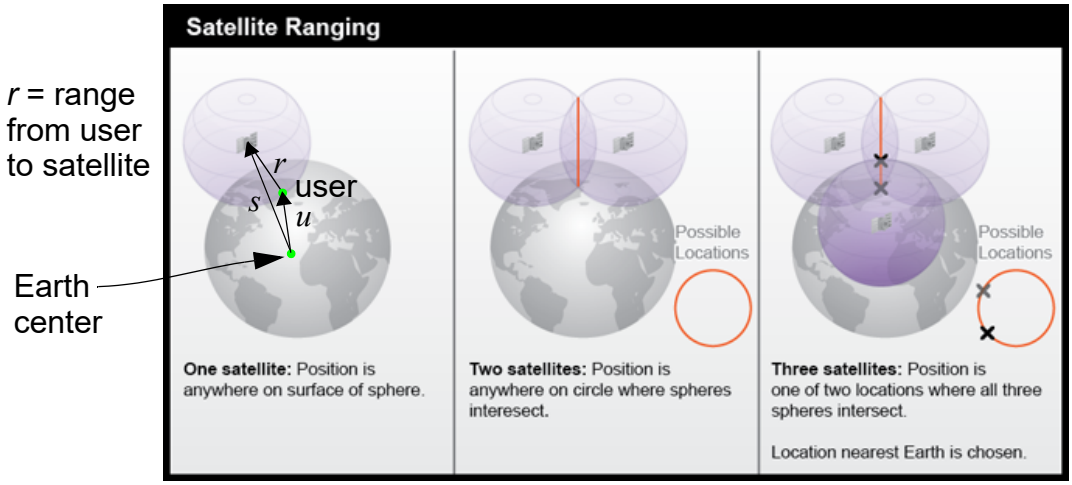


Figure 2: Determining position on the earth's surface using three satellites and an accurate local clock.

earth's surface. A fourth satellite allows elevation to be determined, and is also used in resolving local clock errors.

As vectors in the *earth centered earth fixed* (ECEF) coordinate system you can write that

$$\|\mathbf{r}\| = \|\mathbf{s} - \mathbf{u}\| \tag{1}$$

or considering range as the scalar $r = \|\mathbf{r}\|$

1. https://www.reddit.com/r/askscience/comments/1equw8/what_would_happen_if_i_took_a_gps_receiver_onto/

$$r = \|\mathbf{s} - \mathbf{u}\| \tag{2}$$

The objective is to find $\mathbf{u} = (x_u, y_u, z_u)$ using multiple satellite range measurements and of course knowledge of the satellite locations in ECEF coordinates.

Gold Codes

For commercial GPS use the coarse acquisition (CA) codes of period 1023 bits or *chips* are used for ranging. Each SV is assigned a unique CA code from the family of *Gold Codes* [1], [2]. The bit or chip rate of the ranging code is 1.023 Mcps (Mega chips per second). Note that the code period is exactly 1 ms. There are 37 Gold Codes available. In this project we assume that the code number is synonymous with the SV number (SVN). This is not actually case in practice, but is convenient here.

The Gold codes are special since they form a family of nearly orthogonal sequences. Orthogonal here means that when two SV signals are received by a user, correlation-based signal processing, the two signals do not interfere with each other. Consider $c_i(t)$ is the repeating code waveform of the CA code for the ith SV, the cross correlation between two code waveforms is

$$R_{ij}(\tau) = \lim_{T \to \infty} \frac{1}{2T} \int_{-T}^{T} c_i(t) c_j(t+\tau) dt \cong 0, \text{ for all } \tau \text{ and } i \neq j \tag{3}$$

In cellular telephony this is known as code division multiple access (CDMA), as multiple users can share the same radio frequency spectrum and nominally inflict minimal interference on each other. This is perfect for GPS too, as the user needs to receive multiple satellite signals (4 or more) in order to get a position fix. As with any ranging code system, the user needs to properly synchronize a *local replica* CA code with the received SV signal of interest. In this project the focus is on local replica synchronization. Two facets of the synchronization process are:

- Coarse alignment of the local code with the received signal, which is known as code acquisition
- Fine code tracking using a feedback control system, since the SV and likely the user are in motion, and the local clock is not perfectly synchronous with the transmit clock.

Later in the project description you will be reading about serial search for code acquisition and the noncoherent delay-locked loop (DLL) for fine code phase tracking. The DLL is very much like a phase-locked loop (PLL).

At baseband the CA code waveform consists of ± 1 amplitude pulses of duration $T_c = 1/1.023 = 0.9775$ µs. The transmitted CA code signal is placed on the L1 carrier frequency of $f_{L1} = 1.57542$ GHz as binary phase-shift keyed (BPSK) modulation

$$s_i(t) = \sqrt{2P_I} c_i(t) \cos[2\pi f_{L1} t], \; i = 1, \ldots, 37 \tag{4}$$

where P_I is the power placed on the in-phase carrier, which transmits the CA code, and the subscript i indicates which Gold Code pseudo-random sequence (PRN) is being transmitted. Not

present in this project, but part of the CA code signal, is a 50 bit/s data stream, $d(t)$, which contains:

- Satellite almanac data
- Satellite ephemeris data
- Signal timing data
- Ionospheric delay data
- A satellite health message

To be clear, in the project simulation code only $c_i(t)$, where as in reality $d(t)c_i(t)$ is present. For more information on the formatting of the 50 bps data stream, consult [1] or [2]. Note that the bit period is 20ms, so 20 CA code periods fit into one data bit period.

Pseudorange

I now jump back to the geometric range r given in (2), and write it in terms of time parameters

$$\text{Geometric Range} = r = c(T_u - T_s) = c\Delta t \tag{5}$$

where T_s is the time the signal leaves the satellite, T_u is the time the signal arrives at the receiver, and c is the velocity of propagation. The actual measurement process is depicted in Figure 3. The

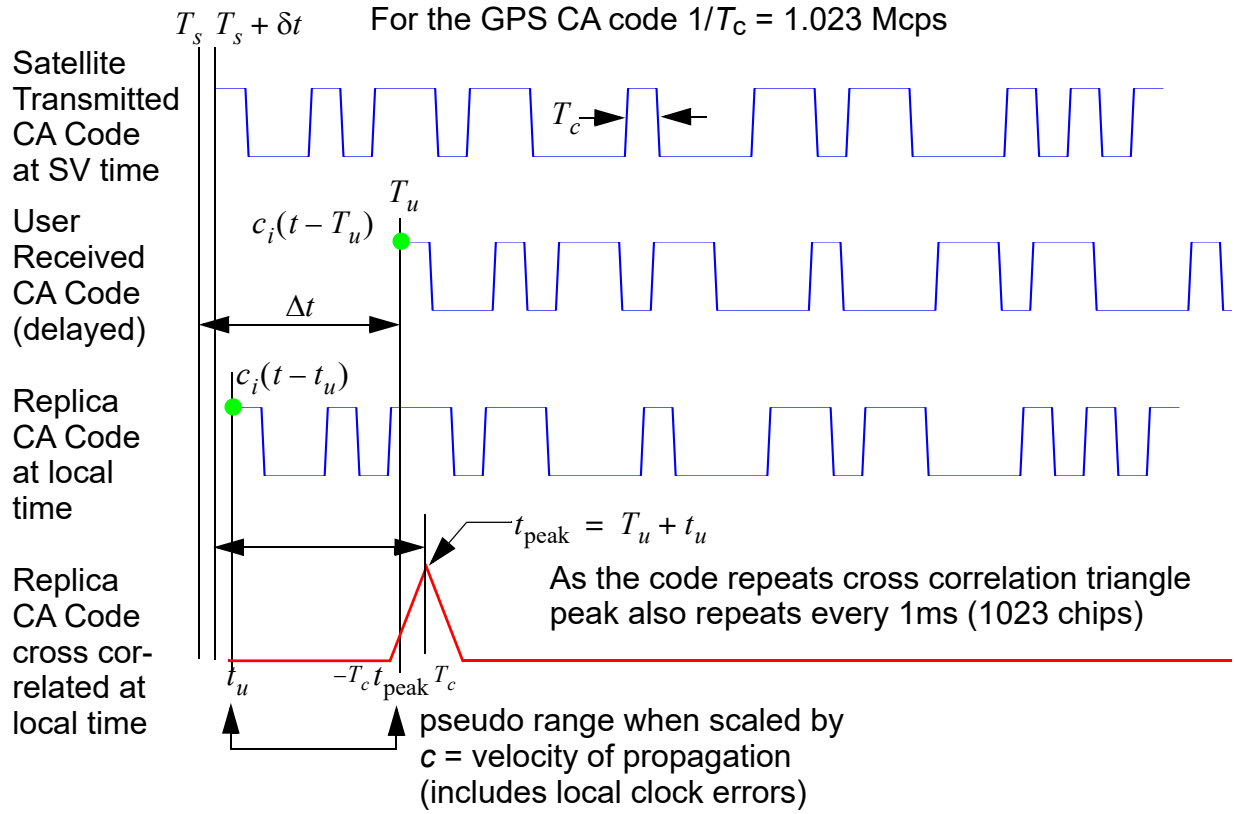


Figure 3: User time delay measurement using cross correlation with the local replica code.

pseudorange [1] is given by

$$\text{Pseudorange} \;=\; \rho \;=\; c[(T_u + t_u) - (T_s + \delta t)] \tag{6}$$

where t_u is the receiver clock offset or error relative to the system clock and δt is the offset of the system clock from the true system time. Rearranging (6) gives

$$\rho \;=\; c(T_u - T_s) + c(t_u - \delta t) \;=\; r + c(t_u - \delta t) \tag{7}$$

The GPS ground monitoring system determines δt and includes this information in the 50 bps data stream. So all that is left is t_u, meaning that to calculate the user position all you need to do is obtain four pseudoranges from received CA code waveforms and solve for the four unknowns (x_u, y_u, z_u) and t_u. The details of this are not part of this project, details can be found in [1] and [2]. Python code that uses a Kalman filter can be found at https://github.com/gps-helper/gps-helper.

Navigation Receiver Based on the RTL-SDR

This project is entirely computer simulation based, but the design of the simulation assumes that the receiver front-end utilizes the RTL-SDR software defined radio dongle [4]. A high level block diagram depicting this configuration is shown in Figure 4.In this figure you see highlighted a

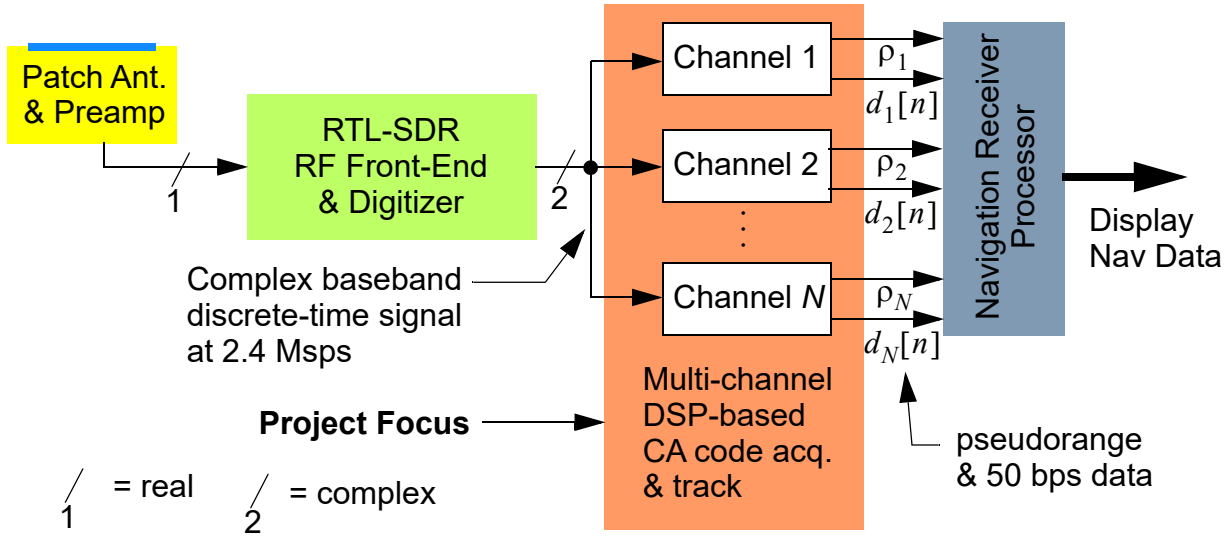


Figure 4: High level GPS receiver block diagram noting the multi-channel CA code acquisition and tracking subsystem.

multi-channel CA code acquisition and tracking subsystem. Each of these channels is responsible for processing the signal from a particular SV and hence uses a local replica CA code to correlate and track the code using a DLL. Serial search is used to find the proper code phase.

In the simulation for this project, the composite signal received signal at 2.4 Msps is formed using a collection of Python functions as shown in Figure 5. The signal is generated at 3 samples per chip making the effective sample rate in Hz be $3 \times 1.023\text{E}6$ MHz, then re-sampled at 2.4E6

MHz. The effective number of samples per chip is thus $2.4/1.023 = 2.3462$ or 0.4262 chips per sample. The top level signal generation function is `CA_rx_RTLSDR()`. Additive white Gaussian noise (AWGN) can also be modeled along with CA signal using the `digitalcom.py` function `cpx_AWGN()`. Once noise is added to one signal there is no need use this function again, as it will

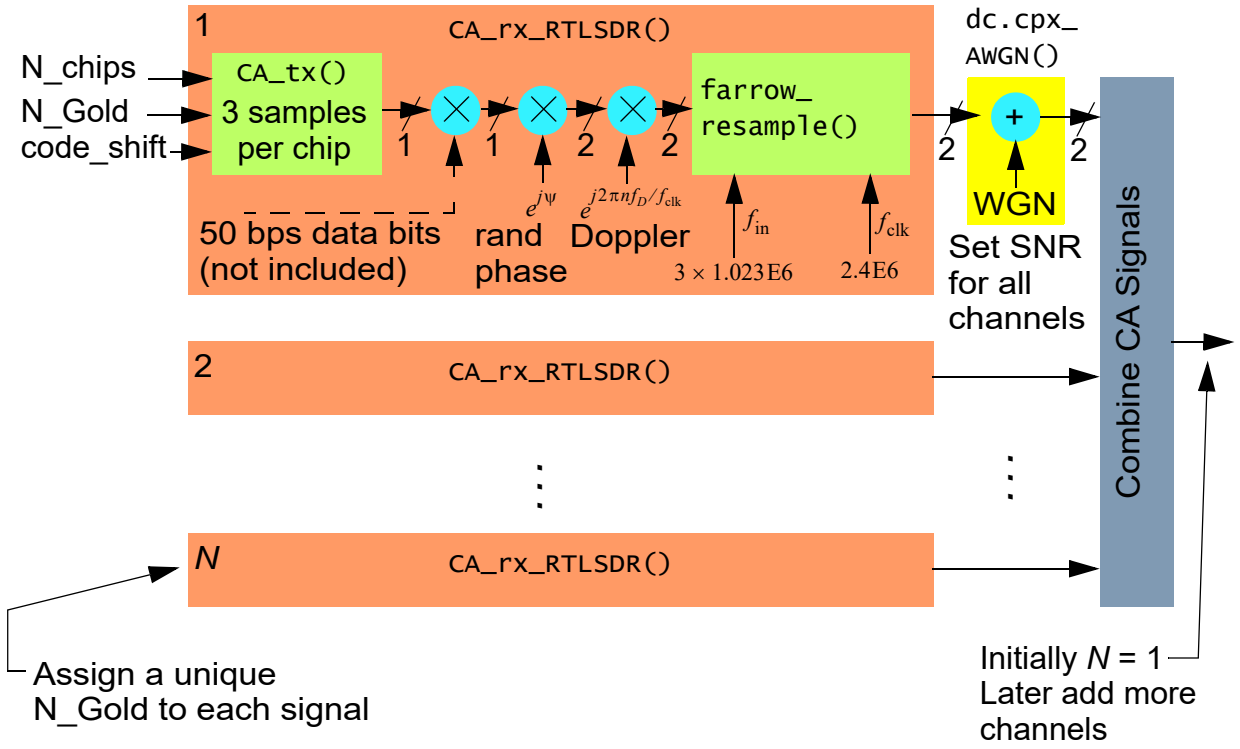


Figure 5: Python simulation model for *N* received CA code signals.

set the same background level for all signals. The doc string help for the top level function is given by:

```
def CA_rx_RTLSDR(Nchips,N_Gold,code_shift=0,
                 fDop = 0.0,f_clk=2.4e6,rphase=True):
    """
    Generate CA signal of given number of chips resampled to f_clk
    N_Gold = Gold code number: 1 - 37

    x_SDR24, x_SDR = CA_rx_RTLSDR(Nchips,N_Gold,code_shift=0,
                                  fDop = 0.0,f_clk=2.4e6,rphase=True)

        Nchips = Number of chips to simulate
        N_Gold = The Gold code number, 1 to 37
    code_shift = roll code starting point forward or backward relative to
                 1023 chip period
          fDop = Applied Doppler frequency shift in Hz
         f_clk = Default 2.4Msps. Output sample rate in sps (note the input to
                 the resampler is a CA code signal at 3 samples per chip or
                 fsamp = 3*1.023 Msps
        rphase = True/False to apply a random phase shift (rotation) to the
                 x_SDR24 signal. Note, if fDop 0 the rphase = False, then the
```

```
                output is real.
    ================================================================
       x_SDR24 = complex baseband GPS signal at f_clk Msps with a fixed
                 random phase and Doppler frequency shift. Normally
                 choose f_clk = 2.4e6.
         x_SDR = A real 3 sample per chip CA signal
    """
```

The function is contained in the module `GPS.py` and the complete code listing is given in the appendix of this document.

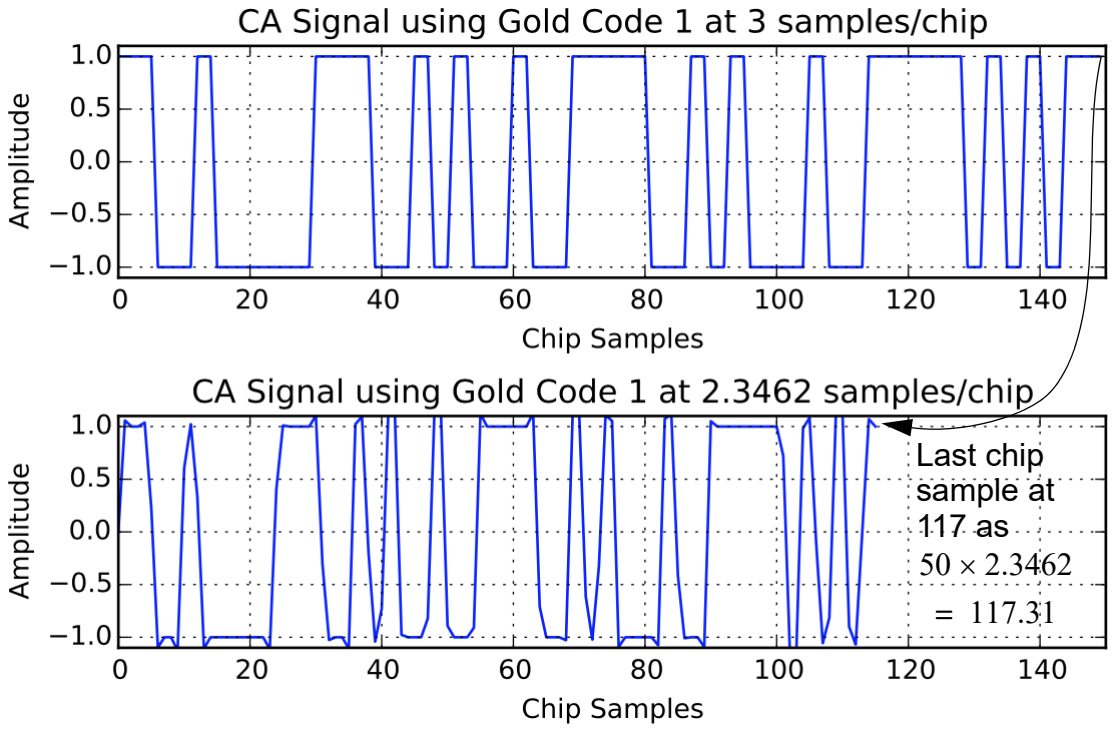
To demonstrate the use of `CA_rx_RTLSDR()`, consider generating 50 CA code chips using `N_Gold = 1`, which is the first Gold code.

```
import GPS
#reload(GPS)
```

```
x_SDR24, x_SDR = GPS.CA_rx_RTLSDR(50,1,0,0,rphase=False)
x_ref = GPS.CA_tx(1023,3,1,0)
```

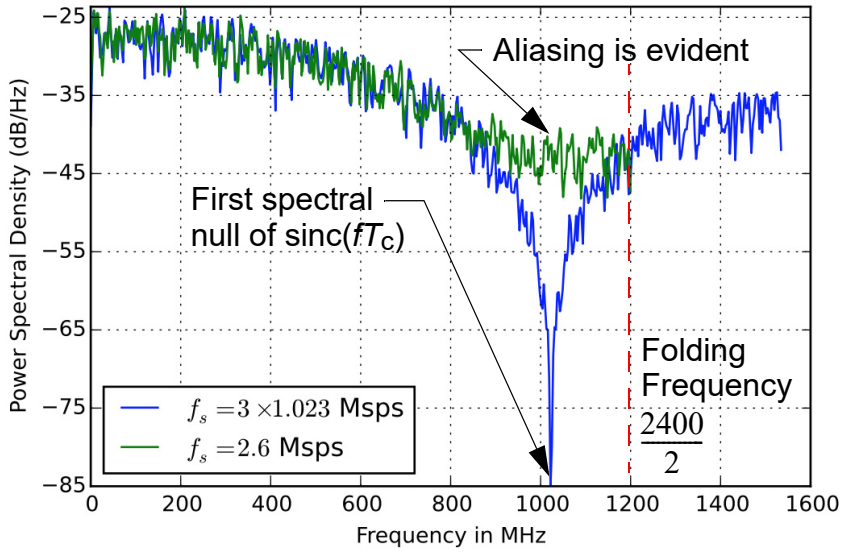
```
subplot(211)
plot(x_SDR)
xlim([0,3*50])
ylim([-1.1,1.1])
xlabel(r'Chip Samples')
ylabel(r'Amplitude')
title(r'CA Signal using Gold Code 1 at 3 samples/chip')
grid();
subplot(212)
plot(x_SDR24)
xlim([0,3*50])
ylim([-1.1,1.1])
xlabel(r'Chip Samples')
ylabel(r'Amplitude')
title(r'CA Signal using Gold Code 1 at 2.3462 samples/chip')
grid();
tight_layout()
```

The options chosen for CA_rx_RTLSDR() have the random phase turned off and the Doppler frequency is 0 Hz, so the output is a real signal. In general with a non-zero Doppler or a single random phase, uniform on $[0, 2\pi]$, drawn for the duration of the simulation, the output `x_SDR24` will be a complex signal with spectrum having center frequency at f_D. This constitutes what is known as a complex baseband signal, since it has real and imaginary parts and spectrum centered near DC. The time/sequence domain plot of the exact 3 samples per chip and the 2.3462 samples per chip waveforms are shown below.



The first plot uses precisely 3 samples per chip, so the waveform looks very nice. The second plot is following re-sampling from $3 \times 1.023\text{E}6$ Msps to 2.4E6 Msps, thus the sample values of the waveform are now asynchronous with respect to the chip rate. The waveform is still clean considering how low the sample rate is and the fact that the pulse shape is rectangular. Since a lowpass filter was not placed in front of the re-sampling operation you expect some aliasing to occur, as the theoretical power spectrum has an envelope of the form $\text{sinc}^2(fT_c)$, where T_c is the chip period. Using the Python `psd()` function this can be verified:

```
psd(x_SDR,2**10,3*1023); psd(x_SDR24,2**10,2400);
xlabel(r'Frequency in MHz')
legend((r'$f_s = 3\times 1.023$ Msps',r'$f_s = 2.6$ Msps'),loc='best');
```



Before leaving the discussion of the CA code waveform simulation function, I will consider the inclusion of AWGN and stack up two signals, one at `N_Gold = 1` and one at `N_Gold = 5`. To add noise to the received signal I adopt the use of C/N_0, the ratio of carrier power-to-noise spectral density at the receiver front end, as the signal quality figure-of-merit. The function `dc.cpx_AWGN()` has as input $(E_c/N_0)_{\text{dB}}$ which is related to $(C/N_0)_{\text{dB-Hz}}$ via

$$\left(\frac{C}{N_0}\right)_{\text{dB}} = \left(\frac{E_c}{N_0}\right)_{\text{dB}} + 10\log_{10}(R_c), \tag{8}$$

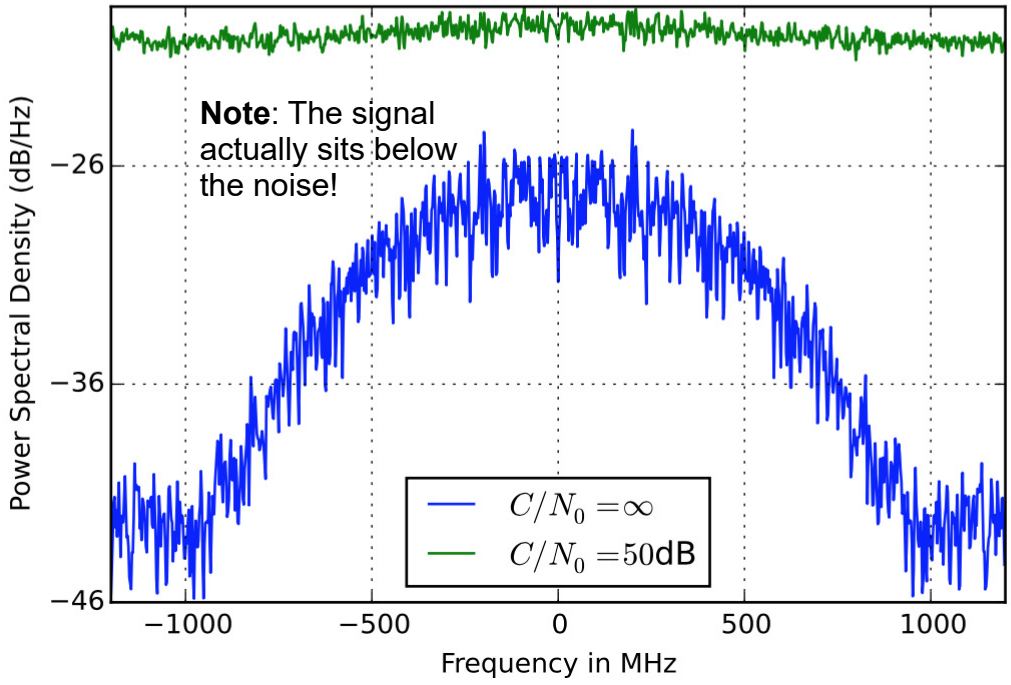
where here the chip rate $R_c = 1.023\times10^6$ chips/s and E_c/N_0 is the energy per chip-to-noise spectral density ratio at the receiver front end. Since R_c is close to 10^6, it is reasonable to approximate $10\log_{10}(R_c)$ as simply 60dB. To set C/N_0 to say 50 dB-Hz I use `cpx_AWGN()` as follows:

```
                                CN0dB-Hz - 60dB,samples per chip
r_SDR24 = dc.cpx_AWGN(x_SDR24,   50-60,  2.4/1.023)
```

The complete example is:

```
x_SDR241,x_SDR3 = GPS.CA_rx_RTLSDR(100000,1,0,10) #10 Hz Doppler
x_SDR245,x_SDR5 = GPS.CA_rx_RTLSDR(100000,5,0,100) # 100 Hz Doppler
r_SDR241 = dc.cpx_AWGN(x_SDR241,50-60,2.4/1.023)
x_SDR24_15 = x_SDR241 + x_SDR245 # two CA signals, no noise
r_SDR24_15 = r_SDR241 + x_SDR245
```

```
psd(x_SDR24_15,2**10,2400);
psd(r_SDR24_15,2**10,2400);
xlim([-1200,1200])
xlabel(r'Frequency in MHz')
legend((r'$C/N_0 = \infty$',r'$C/N_0 = 50$dB'),loc='best');
```



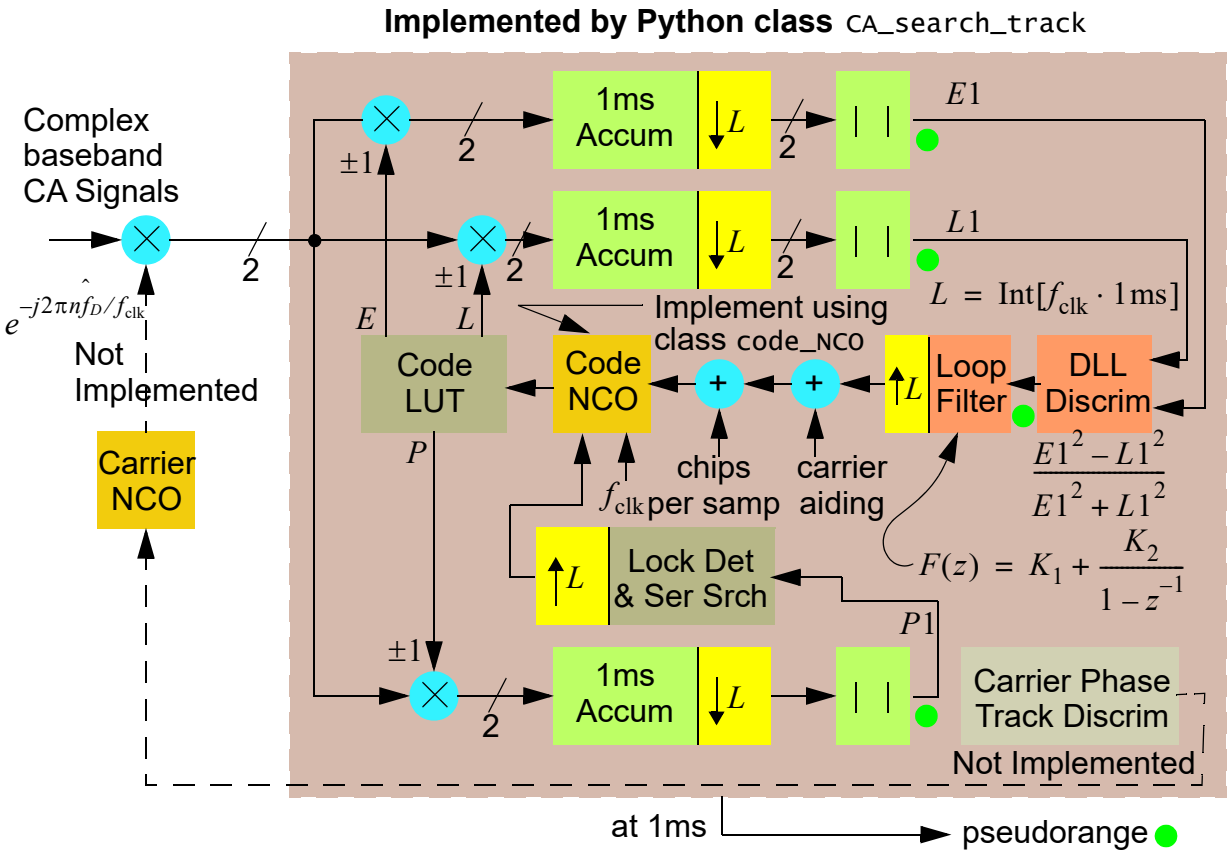


Figure 6: Serial search code acquisition and noncoherent DLL code tracking.

With some understanding of how to generate CA code signals, next up is an explanation of the serial search acquisition and DLL tracking receiver.

Delay-Locked Loop Code Phase Tracking

The block diagram of the serial search acquisition and noncoherent DLL CA code receiver subsystem, is shown in Figure 6. A portion of a carrier tracking system, which is often used to remove Doppler, is presently not implemented. The Doppler term, f_D, and its estimate $\hat{f}_D$, are thus cannot be moved far away from zero in the signal generation of Figure 5. All of the implemented functionality of Figure 6 is implemented by the Python class `CA_code_track` found in the module GPS.py (see the appendix of the code ZIP package for more details). The code tracking class also makes use of the `code_NCO` class. Of special note, each instance of the `CA_code_track` class is an **object** containing **three code correlators**. The *prompt* or *P* channel is used to detect coarse code alignment, and in a complete GPS receiver, not implemented here, recover the 50 bps data stream. The *early* or *E* and *late* or *L* channels are used to form an error signal that advances or retards the code Numerically controlled oscillator (NCO). The default time separation between these signals is 0.5 chip. The objective is to keep the replica code perfectly aligned with the input signal. When this happens the NCO code phase contains the pseudorange, which as you know from an earlier

discussion, is used to solve for the user location.

All of the signal points marked with a green dot are test points returned by the class attribute `trk_var`:

```
'''
+++++++++++++++++++++++++++++++++++++++++++++++++++++++
  trk_vars = a collection loop signals recorded during
             the simulation:
             row 0 = abs(early correlator output)
             row 1 =  abs(prompt correlator output)
             row 2 =  abs(late correlator output)
             row 3 =  DLL discriminator output
             row 4 =  pseudorange output in microseconds
+++++++++++++++++++++++++++++++++++++++++++++++++++++++
'''
```

Among the logged signals there is a computed signal that represents the pseudorange (row 4), scaled to have units of μs. The units are correct, but some work is needed to accurately calibrate the time offset. The raw pseudorange output is derived from the code NCO when differenced against a free running NCO having no error inputs, to advance or retard the chip phase/time skew.

At the input a Doppler frequency error frequency translation multiplier is shown. A means for driving this multiplier is not implemented, but is needed in a real receiver. You will find in your experimentation that the DLL code tracking subsystem cannot tolerate a very large Doppler frequency. Recall that the Doppler frequency, f_D, can be as high as 4.2 kHz just due to satellite motion relative to a user on the ground.

Code acquisition and tracking relies on the use of the replica code cross correlation waveform depicted at the bottom of Figure 3. In Figure 6 the cross correlation is implemented by multiplying the input CA code signal(s) by timed ± 1 samples of the Gold code pulled from a code lookup table (LUT) and then accumulating/integrating over a code period of 1ms. Since the phase of the input signal is unknown and not being tracked, the magnitude or magnitude squared output of the accumulator is used for operating the DLL and the serial search acquisition functions. Working with the magnitude makes the processing *noncoherent* with respect to the input signal phase. This makes sense when you think about the objective of the DLL being to bring the timing of the local replica into time alignment with a specific CA code signal present in the received signal. Recall more than one CA code signal may be present, but the Gold code orthogonality makes the correlator ignore the other signals that may be present.

The tracking operation with the code NCO also allows the local clock, f_{clk}, to be asynchronous relative to the received signal chip clock. What I mean by this is the code NCO not only has to implement time skew, it also has to be able to slow down or speed up. The NCO does this by updating both an integer chip delay/advance attribute and a fractional delay/advance attribute. The fractional value is updated every f_{clk} with a chips/sample constant, plus once every ms the loop filter output error signal is included. When the fractional value falls outside [0,1) a value of 1.0 is added or subtracted from the NCO integer attribute.

To test single channel acquisition and tracking you can use the function `CA_track()`:

```
def CA_Track(rec_sig,f_clk,N_Gold,T1,T2,ss_step=0.5,Bn=100,
             code_delta = 0.5,c_offset=0):
    """
    Delay-lock Loop Using 1ms Integration

    trk_vars = CA_Track(rec_sig,f_clk,code,Bn,chip_delta=0.5,c_offset=0)
    #+++++++++++++++++++++++++++++++++++++++++++++++++++++++
       rec_sig = received complex baseband signal
         f_clk = sampling clock in sps
        N_Gold = Gold code sequence number
            T1 = Threshold used to declare code acquisition (nominally 2000)
            T2 = Threshold used to return to code serial search
                 (nominally 1000)
       ss_step = Code serial search step size and direction in chips
            Bn = Tracking loop bandwidth in Hz
    chip_delta = early late chip delta, nominally +/- 0.25chip
      c_offset = local code offset at start of the simulation
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++
      trk_vars = a collection loop signals recorded during
                 the simulation:
                 row 0 = abs(early correlator output)
                 row 1 =  abs(prompt correlator output)
                 row 2 =  abs(late correlator output)
                 row 3 =  DLL discriminator output
                 row 4 =  pseudorange output in microseconds
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Mark Wickert November/December 2015
    """
```

As an example consider a 100,000 chip test signal using Gold code 1 with some delay added to the received signal using the `code_shift` argument of `CA_rx_RTLSDR()`. For the initial test C/N_0 will be set very high (> 100 dB) and the lock threshold `T1` will set to 2500 to **keep** the receiver is serial search mode. The serial search step size `ss_step` will be made very small so that you can see the response of the `E`, `P`, and `L` correlators as the timing alignment of the replica code steps past. As a second part of this example, the `T1` threshold will be lowered back 2000 to allow

serial search to stop and the DLL tracking function take over and pull the loop error down to zero.

```
# Delay by 1 whole chip and 3 samples at 0.42625 chips/sample
CN0 = 100
x_SDR241,x_SDR31 = GPS.CA_rx_RTLSDR(100000,1,1,0)
x_SDR241D = signal.lfilter([0,0,0,1],1,x_SDR241)
r_SDR241 = dc.cpx_AWGN(x_SDR241D,CN0-60,2.4/1.023)
```

Keep from locking by setting T1 high

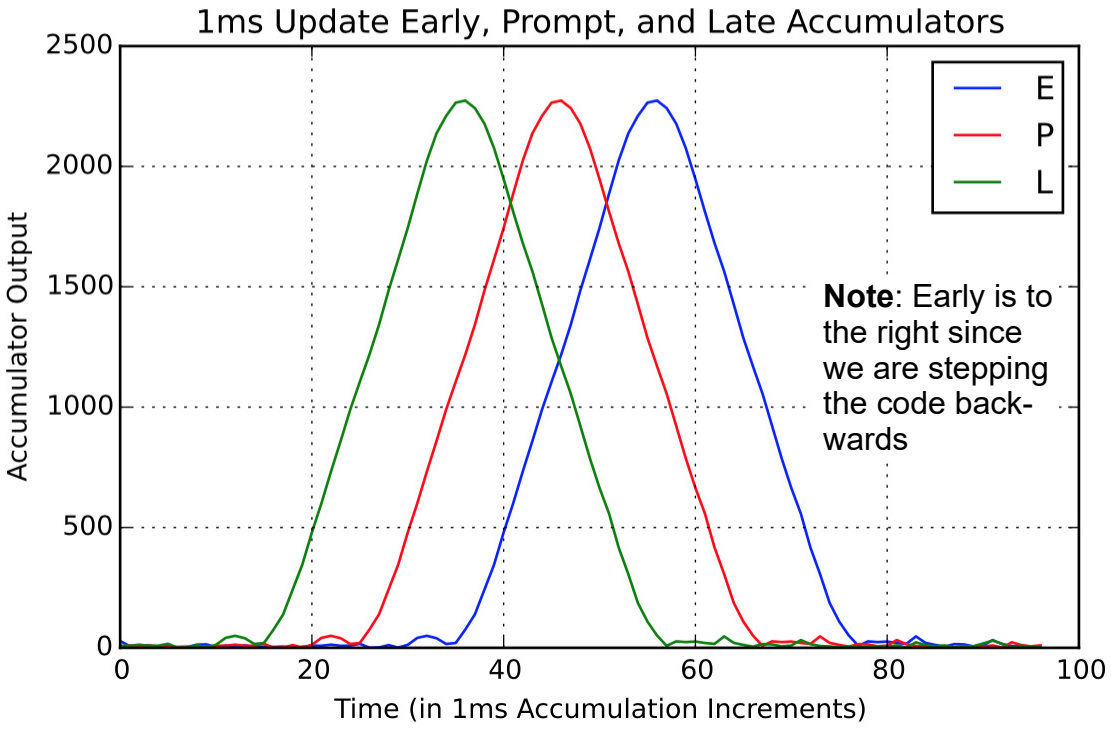
```
trk_vars_SV1 = GPS.CA_Track(r_SDR241,2.4e6,1,2500,1000,
                            -0.05,40,0.5,2.4) # cal factor 2.4
```

```
N_samples = 234598
N_1ms = 2400
K1 = 4.5463e-02, K2 = 2.4252e-03
Search n = 2400
Search n = 4800
Search n = 7200
Search n = 9600
Search n = 12000
Search n = 14400
Search n = 16800
Search n = 19200
Search n = 21600
Search n = 24000
```

Step replica back 0.05 chips every 1 ms; normal ss_step is 0.5 chip for a more rapid code search

```
GPS.plot_tracking_ELP(trk_vars_SV1)
ylim([0,2500])
```

```
(0, 2500)
```



Noting that the code steps 0.05 chips/ms, it is possible to rescale the x-axis to have units of chips.

Moving on, `T1` is now set back to 2000 and `ss_step` is set back -0.05 chips. Now coarse acquisition occurs and the track mode is entered. More plots will be shown as both the tracking

```
trk_vars_SV1 = GPS.CA_Track(r_SDR241,2.4e6,1,2000,1000,
                            -0.5,40,0.5,2.4) # cal factor 2.4
```

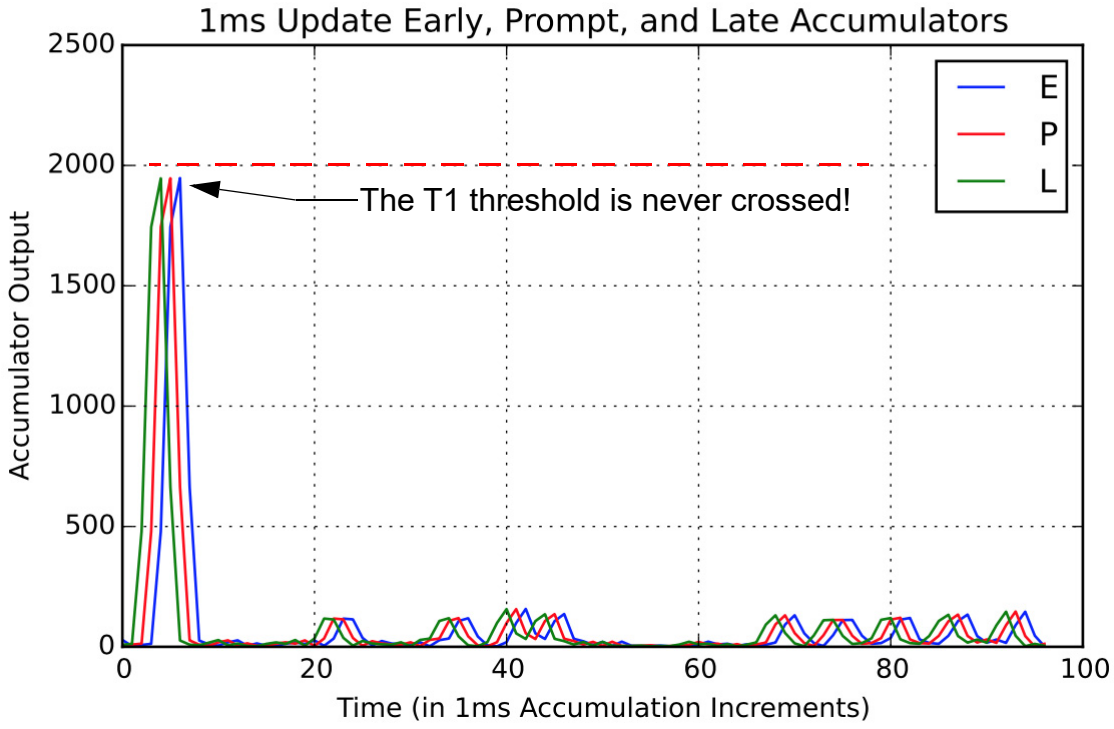
```
N_samples = 234598
N_1ms = 2400
K1 = 4.5463e-02, K2 = 2.4252e-03
Search n = 2400
Search n = 4800
Search n = 7200
Search n = 9600
Search n = 12000
Search n = 14400
Search n = 16800
Search n = 19200
Search n = 21600
Search n = 24000
Search n = 26400
Search n = 28800
Search n = 31200
```

Track mode is never entered during this 100,000 chip simulation! What's wrong?

error and the pseudorange are zeroed until track mode should be entered, but it is not. To troubleshoot take a look at the correlator outputs:

```
GPS.plot_tracking_ELP(trk_vars_SV1)
ylim([0,2500])
```

```
(0, 2500)
```



When stepping at 0.5 chips per ms the correlator output does not have time to settle to the maximum vale you see when stepping at 0.05 chips. I will lower `T1` down to 1500 and see what happens:

```
trk_vars_SV1 = GPS.CA_Track(r_SDR241,2.4e6,1,1500,1000,
                            -0.5,40,0.5,2.4) # cal factor 2.4
```

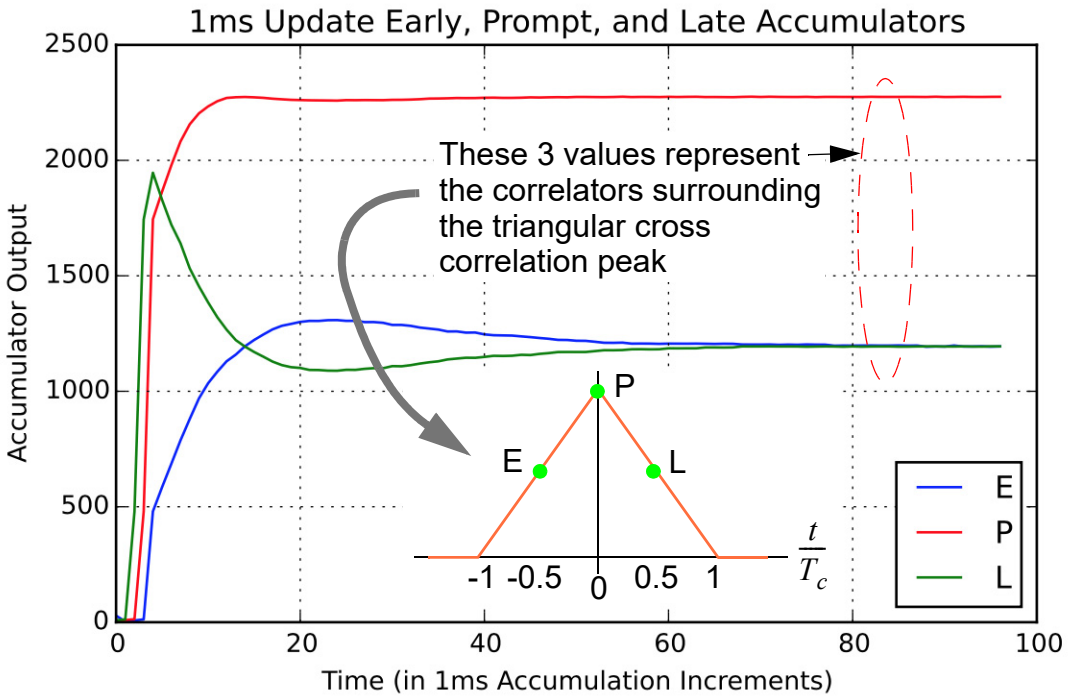
40 Hz closed-loop tracking bandwidth

```
N_samples = 234598
N_1ms = 2400
K1 = 4.5463e-02, K2 = 2.4252e-03
Search n = 2400
Search n = 4800
Search n = 7200
Search n = 9600
Fine Track n = 12000
Fine Track n = 14400
Fine Track n = 16800
Fine Track n = 19200
Fine Track n = 21600
Fine Track n = 24000
Fine Track n = 26400
```

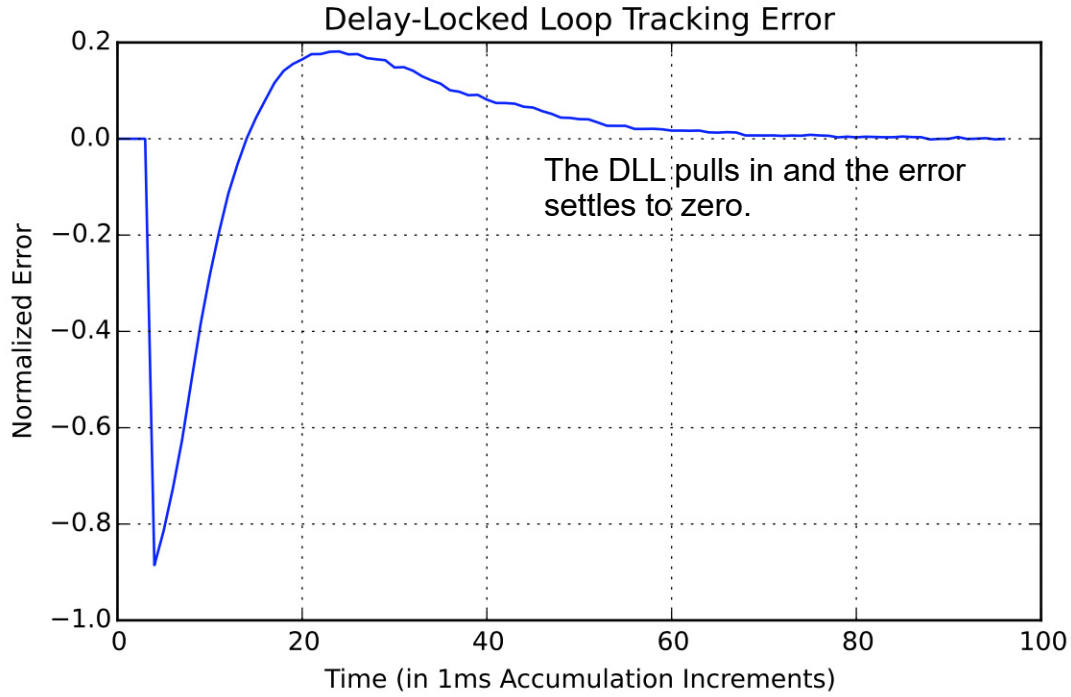
Now the system enters fine code tracking
The downside of lowering the threshold is that when noise is present a false lock make occur

```
GPS.plot_tracking_ELP(trk_vars_SV1)
ylim([0,2500])
```

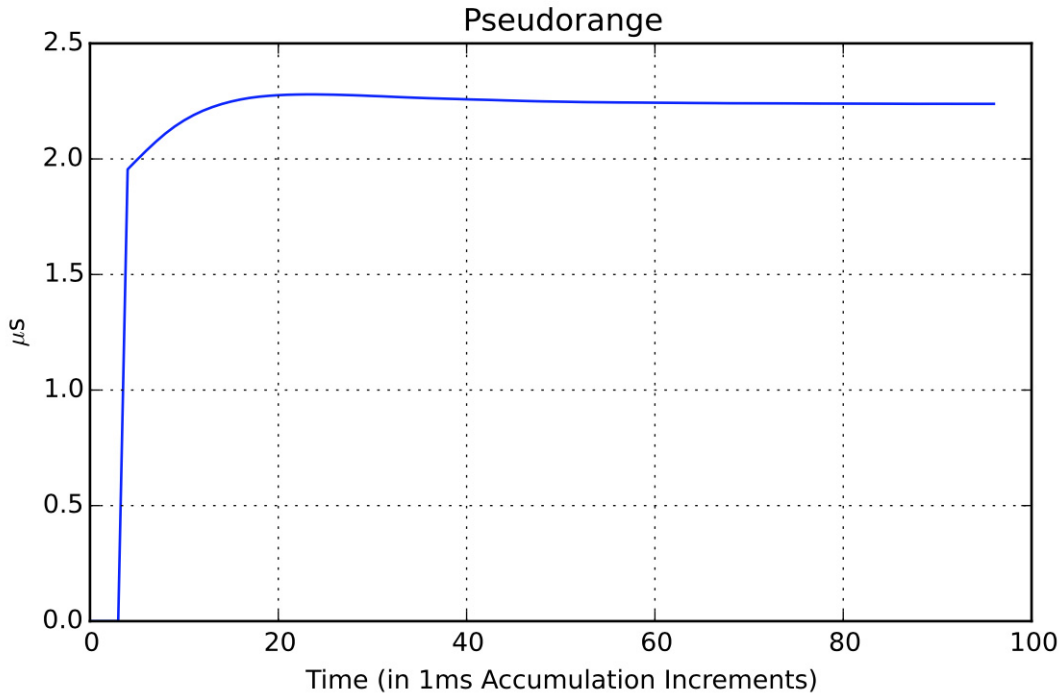
```
(0, 2500)
```




```
GPS.plot_tracking_error(trk_vars_SV1)
```



```
GPS.plot_pseudorange(trk_vars_SV1)
```




```
trk_vars_SV1[4,-1]
```

```
2.2898507204954512
```
◄—— steady-state pseudorange in us

Note in all of the above the DLL look bandwidth is set to 40 Hz. A faster response time can be obtained by increasing the bandwidth to say 100 Hz and a slower response time is obtained by reduce the bandwidth down to say 10 Hz. In a real system a loop bandwidth of 2 Hz is common.

As a final part of this example I will lower C/N_0 down to 50 dB-Hz and set $B_n = 100$ Hz:

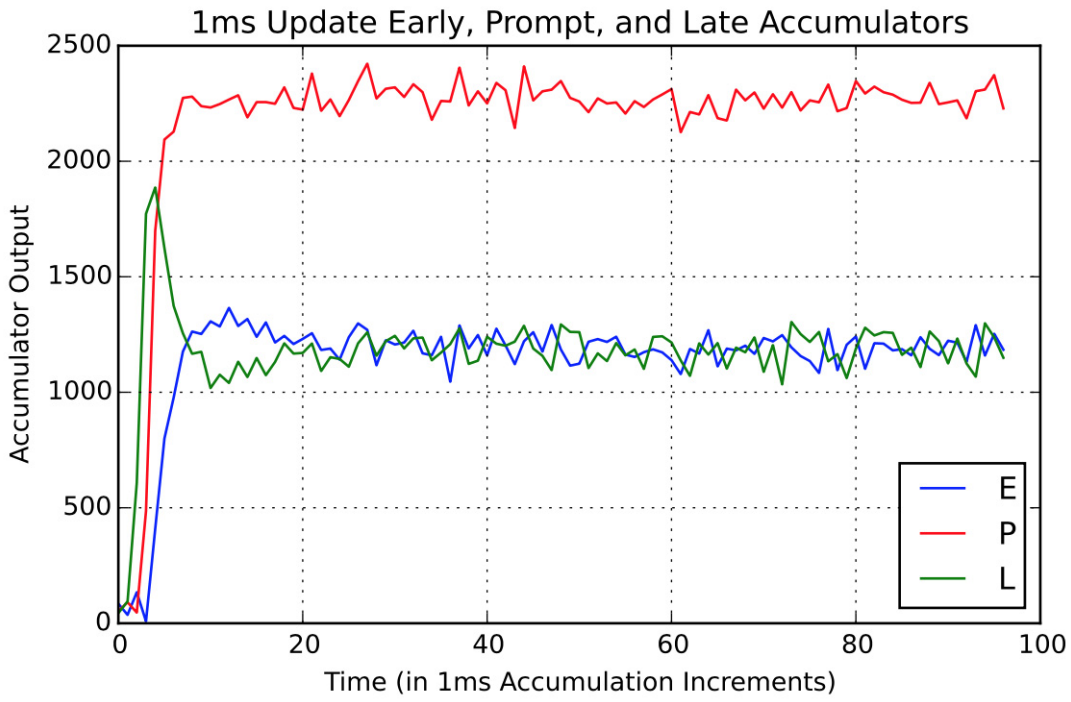
```
# Delay by 1 whole chip and 3 samples at 0.42625 chips/sample
CN0 = 60
x_SDR241,x_SDR31 = GPS.CA_rx_RTLSDR(100000,1,1,0)
x_SDR241D = signal.lfilter([0,0,0,1],1,x_SDR241)
r_SDR241 = dc.cpx_AWGN(x_SDR241D,CN0-60,2.4/1.023)
```

```
trk_vars_SV1 = GPS.CA_Track(r_SDR241,2.4e6,1,1500,1000,
                            -0.5,100,0.5,2.4) # cal factor 2.4
```

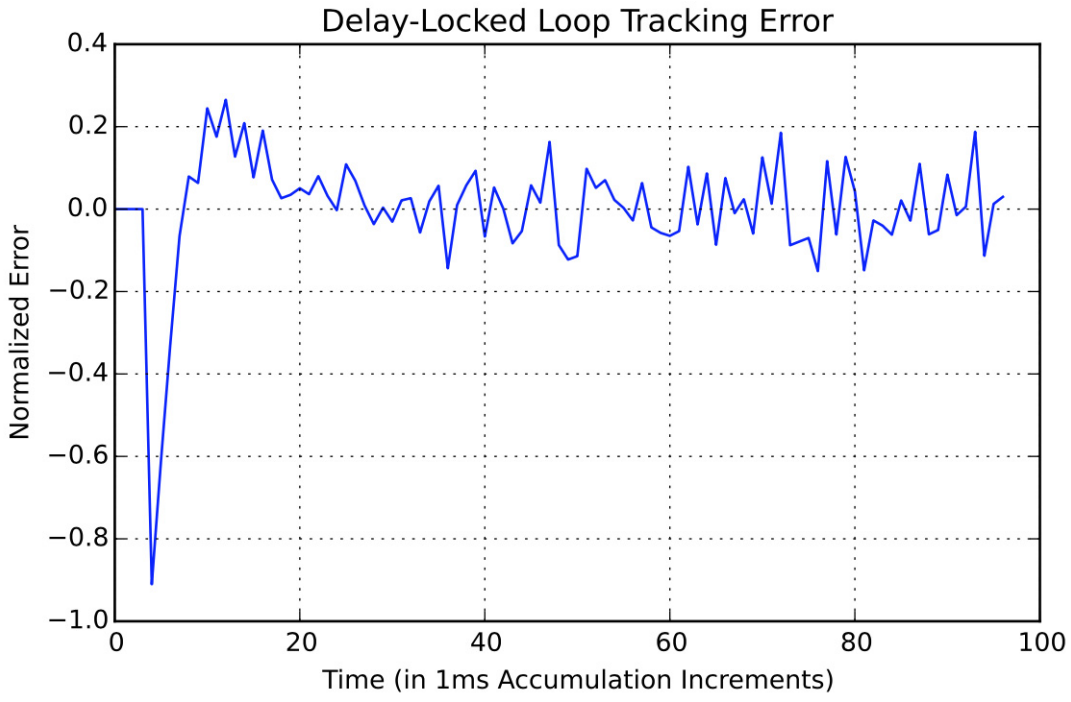
```
N_samples = 234598
N_1ms = 2400
K1 = 1.1366e-01, K2 = 1.5157e-02
Search n = 2400
Search n = 4800
Search n = 7200
Search n = 9600
Fine Track n = 12000
```

```
GPS.plot_tracking_ELP(trk_vars_SV1)
ylim([0,2500])
```

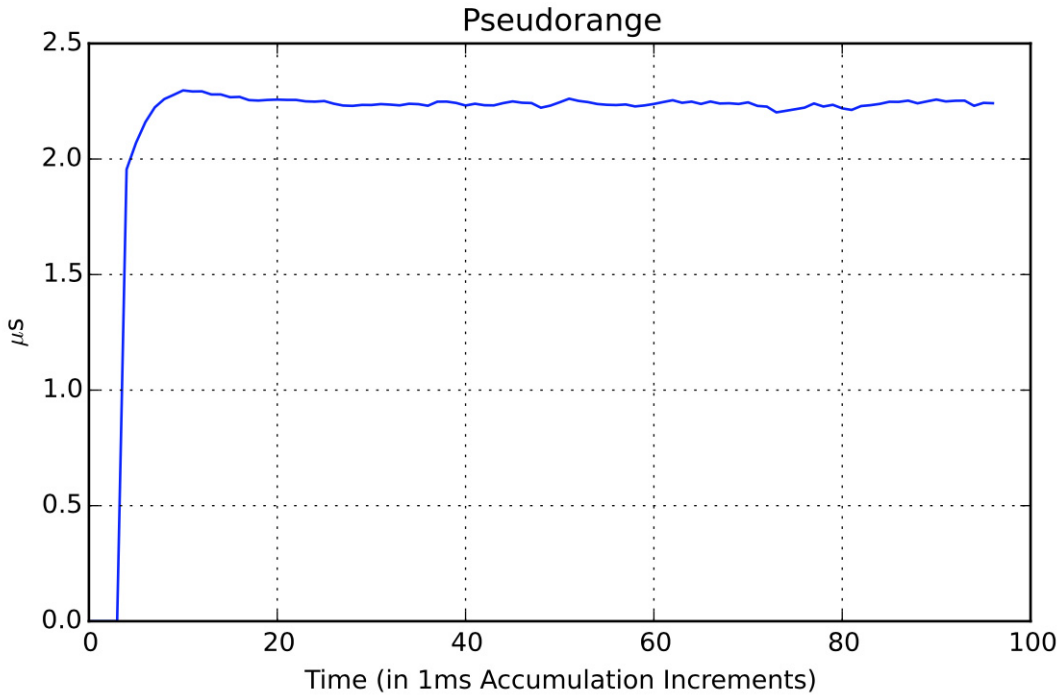
```
(0, 2500)
```




```
GPS.plot_tracking_error(trk_vars_SV1)
```



```
GPS.plot_pseudorange(trk_vars_SV1)
```




```
trk_vars_SV1[4,-1]
```

```
2.2929492165858392
```

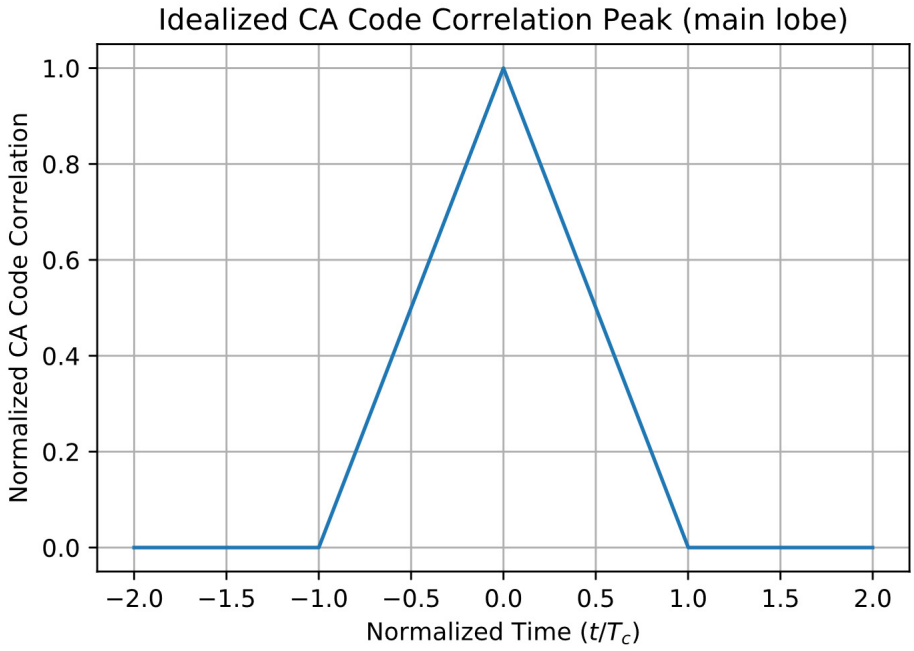
The results look reasonable, that is noise has impaired system performance. The DLL tracking error pseudorange measurements have non-zero variance. As a side note, increasing the loop bandwidth to 100 Hz has reduced the settling time of the DLL at the expense of noise variance being increased by a factor of 100/40.

Project Tasks

Overview of problems

1. Ideally we know that the code cross correlation function between the received CA code signal and the local replica is a triangle shape having base of $2T_c$ and normalized height of unity. In Python code we can describe this as:

```
t = arange(-2,2,.001)
R_CA = ss.tri(t,1.0)
plot(t,R_CA)
xlabel(r'Normalized Time ($t/T_c$)')
ylabel(r'Normalized CA Code Correlation')
title(r'Idealized CA Code Correlation Peak (main lobe)')
grid();
```



a) Plot the theoretical DLL error response characteristic for 0.5 chip correlator spacing using the above as the correlation shape, and the formula used in the code to form the DLL error, i.e.,

$$e(t) = \frac{R_{\mathrm{CA}}^2(t+0.5) - R_{\mathrm{CA}}^2(t-0.5)}{R_{\mathrm{CA}}^2(t+0.5) + R_{\mathrm{CA}}^2(t-0.5)} \tag{9}$$

The real system is discrete, but for modeling purposes the continuous-time is more convenient and is accurate assuming the sampling rate is high relative to the formation of the correlation function. The most interesting interval is $t \in [-0.5, 0.5]$, where I have assumed t has been normalized by the chip period T_c.

b) Using the data collected in the `E` and `L` rows of `trk_vars_SV1`, used in the plot at the top of p. 14, plot the experimental version of the DLL error characteristic of part (a). You will have to do some scaling and time axis shifting of the experimental data. Total agreement is not expected as the down sampling operation changes the shape of the correlation triangle.

2. Verify that the loop filter, $F(z)$, is properly implemented by studying the code found in the

`CA_search_track()` class method code_fine_track():

```
def code_fine_track(self,k):
    """
    Close the tracking loop
    """
    # Form the Code alignment error tracking signal
    E2 = np.abs(self.early_int)**2 # Find magnitude squared
    L2 = np.abs(self.late_int)**2 # Find magnitude squared
    if E2+L2 == 0:
        code_error = 0
    else:
        code_error = (E2-L2)/(E2+L2)

    # Fill instrumentation arrays
    self.trk_vars[0,k] = abs(self.early_int)
    self.trk_vars[1,k] = abs(self.prompt_int)
    self.trk_vars[2,k] = abs(self.late_int)
    self.trk_vars[3,k] = code_error
    self.trk_vars[4,k] = (self.st_code_NCO.c_int_ref+self.st_code_NCO.c_frac_ref)\
                         -(self.st_code_NCO.c_int_p + self.st_code_NCO.c_frac_p)

    # Code tracking second-order loop filter
    code_lead = self.K1 * code_error
    code_lag = self.K2 * code_error + self.code_LF_state
    self.code_filter_error = code_lead +  code_lag
    self.code_LF_state = code_lag
    #code_filter_error = 0
```

3. Build up a four channel receiver using the function `CA_track()` as a template.

   a) Develop the needed code to support a four channel code tracking receiver. Design it to track Gold codes 1, 5, 12, and 17. As part of the code testbed you will also need to generate an input consisting of four CA code signals using the corresponding Gold code numbers.

   b) Test the system at high C/N_0 (> 100 dB-Hz) using a 200,000 chip test array. Set the delays as follows: SV1 6 chips plus 0 samples, SV5 14 chips plus 3 samples, SV12 20 chips plus 2 samples, and SV17 27 chips plus 0 samples. Set the DLL loop bandwidth B_n of each tracking subsystem to 100 Hz and use a step size of -0.5 chips for `ss_step`. Report the final values of pseudorange for each of the channels.

   c) Repeat part (b) with some minor changes in delay for SV5 and SV17. Change the delay on SV5 to 13 chips and 2 samples and SV17 to 27 chips plus one sample. Compare the change in pseudorange for SV5 and SV17 and see if it agrees with your expectations.

   d) Repeat part (c) with C/N_0 decreased to 60 dB-Hz. Customize the loop bandwidth for SV1 to be 100 Hz and SV5 to be 50 Hz. Compare the DLL error variance for these two channels and see that the ratio of the variances is as expected, that is about 2:1. The loop error variable is stored in `trk_vars[3,:]`, but you will need to trim this array before computing the variance so as not to include loop settling transients.

   Note: Testing has revealed that the full 2:1 variance reduction is not achieved. For a linear phase-locked loop this is the expected behavior. I believe there is a calibration error

in setting up the loop. Something for me to work on later.

Thats all!

4. Bonus Problem: Compare the measured step response of the DLL to the theoretical second-order DLL response.

Bibliography/References

[1] Elliot Kaplan, editor, *Understanding GPS Principles and Applications*, Artech, Boston, 1996.

[2] M. Grewal, L. Weill, and A. Andrews, *Global Positioning Systems, Inertial Navigation, and Integration*, Wiley, New York, 2001.

[3] *GPS data processing: code and phase Algorithms, Techniques and Recipes*, available at http://gage.upc.edu/forum/gps-data-processing-code-and-phase-algorithms-techniques-and-recipes.

[4] http://sdr.osmocom.org/trac/wiki/rtl-sdr.

Appendix

Code listing for the project module `GPS.py`.

```
"""
GPS Function Module

Mark Wickert November 2015 - December 2015

Development continues!
"""

"""
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
"""

from matplotlib import pylab
import numpy as np
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm. digitalcom as dc
import matplotlib.pyplot as plt
from scipy import signal
```

```
#from sys import exit

# Load the CA code 37 x 1023 2D ndrray into the Python workspace
camat = np.loadtxt('ca1thru37.txt',dtype=np.int16,unpack=True)

def CA_tx(Nchips,Ns,N_Gold,code_shift=0):
    """
    Generate CA signal of given number of chips
    N_Gold = Gold code number: 1 - 37
    """
    N_GP = 1023; # Gold code period
    gold_code = camat[N_Gold-1,:]
    #gold_code = np.arange(1023,dtype=np.int16)
    gold_code_r = np.roll(gold_code,code_shift)
    x = np.zeros(Nchips)
    N_periods = Nchips//N_GP
    N_chip_rem = Nchips - N_GP*N_periods
    for n in xrange(N_periods):
        x[n*N_GP:(n+1)*N_GP] = gold_code_r
    if N_chip_rem > 0:
        x[N_periods*N_GP:] = gold_code_r[:N_chip_rem]
    x_up = ssd.upsample(2*x-1,Ns)
    x_up = signal.lfilter(np.ones(Ns),1,x_up)
    return x_up

def CA_rx_RTLSDR(Nchips,N_Gold,code_shift=0,
                 fDop = 0.0,f_clk=2.4e6,rphase=True):
    """
    Generate CA signal of given number of chips resampled to f_clk
    N_Gold = Gold code number: 1 - 37

    x_SDR24, x_SDR = CA_rx_RTLSDR(Nchips,N_Gold,code_shift=0,
                                  fDop = 0.0,f_clk=2.4e6,rphase=True)

        Nchips = Number of chips to simulate
        N_Gold = The Gold code number, 1 to 37
    code_shift = roll code starting point forward or backward relative to
                 1023 chip period
          fDop = Applied Doppler frequency shift in Hz
         f_clk = Default 2.4Msps. Output sample rate in sps (note the input to
                 the resampler is a CA code signal at 3 samples per chip or
                 fsamp = 3*1.023 Msps
        rphase = True/False to apply a random phase shift (rotation) to the
                 x_SDR24 signal. Note, if fDop 0 the rphase = False, then the
                 output is real.
    ================================================================
       x_SDR24 = complex baseband GPS signal at f_clk Msps with a fixed
                 random phase and Doppler frequency shift. Normally
                 choose f_clk = 2.4e6.
         x_SDR = A real 3 sample per chip CA signal
    """
    x_SDR = CA_tx(Nchips,3,N_Gold,code_shift)
    n = np.arange(len(x_SDR))
    if rphase:
        x_SDR_rot = x_SDR*np.exp(1j*2*np.pi*np.random.rand(1)) # random phase rotation
        x_SDR_rot = x_SDR_rot*np.exp(1j*2*np.pi*fDop/f_clk*n) # apply Doppler
    else:
        if fDop > 0:
            x_SDR_rot = x_SDR*np.exp(1j*2*np.pi*fDop/f_clk*n) # apply Doppler only
```

```
        else:
            x_SDR_rot = x_SDR
    x_SDR24 = dc.farrow_resample(x_SDR_rot,3*1.023e6,f_clk) # asynch resample
    return x_SDR24, x_SDR

# A class to manage the code NCO for early, prompt and late
# correlator timing control
class code_NCO(object):
    """
    A class that manages the code tracking loop numerically
    controlled oscillator for early, prompt, and late correlators.
    The attributes are code integer offset and the code fraction
    for all three correlators, all in chips (multiples of the chip
    period which is one over the chip rate)
    Mark Wickert November 2015
    """
    def __init__(self,R_chip,f_clk,DLL_delta = 0.5,c_period = 1023):
        # NCO center frequency
        self.NCO_CF = R_chip/f_clk #both must have the same units, e.g., MHz
        # Code period in chips
        self.c_period = c_period # code period in chips
        # Delay lock loop early & late time delta relative to prompt
        self.c_delta = DLL_delta
        # NCO accumulator integer and fractional values
        # The integer value is code chips and the fractional
        # part is fractions of a chip modulo 1
        self.c_int_p = 0
        self.c_frac_p = 0.0
        # Maintain a reference accumulator for easy pseudo-range calculation
        self.c_int_ref = 0
        self.c_frac_ref = 0


    def update(self,c_error):
        """
        Update NCO attributes
        following the loop error calculation
        """
        # Increment the code NCO
        # ******** prompt ***************************
        self.c_frac_p += c_error + self.NCO_CF
        # Check for code NCO overflows/underflows so
        if self.c_frac_p >= 1:
            self.c_int_p += 1
        elif self.c_frac_p < 0:
            self.c_int_p -= 1

        # Modulo 1 the c_frac_p value
        self.c_frac_p = np.mod(self.c_frac_p,1)

        #Update the reference NCO
        self.c_frac_ref += self.NCO_CF
        # Check for code NCO overflows/underflows so
        if self.c_frac_ref >= 1:
            self.c_int_ref += 1
        elif self.c_frac_ref < 0:
            self.c_int_ref -= 1

        # Modulo 1 the c_frac_p value
```

```
        self.c_frac_ref = np.mod(self.c_frac_ref,1)

    def step_code(self,step_frac, step_int = 0):
        """
        Step the code NCO by a chip fraction and integer
        as desired. The most likely use of the
        """
        # Increment the code NCO
        # ******** prompt ***************************
        self.c_int_p += step_int
        self.c_frac_p += step_frac
        # Check for code NCO overflows/underflows so
        if self.c_frac_p >= 1:
            self.c_int_p += 1
        elif self.c_frac_p < 0:
            self.c_int_p -= 1

        # Modulo 1 the c_frac_p value
        self.c_frac_p = np.mod(self.c_frac_p,1)

    def code_epl_chip_sample(self,shift,code):
        """
        Shift the code output sample used in the correlator taking into
        account the NCO integer and fractional offsets and an additional
        shift or offset factor. Note the shift might come from coarse
        acquisition.

        code_shift(shift)
        """

        # ****** shift and apply e, l, or p offset ******
        # Make sure chip samples repeat via the use of mod( ,c_period)
        # Early
        c_int_out = self.c_int_p + int(shift + self.c_delta)
        c_frac_out = self.c_frac_p + ((shift + self.c_delta) \
                                      - int(shift + self.c_delta))
        if c_frac_out >= 1:
            c_int_out += 1
        elif c_frac_out < 0:
            c_int_out -=  1
        cs_e = 2*code[np.mod(c_int_out,self.c_period)] - 1
        # Prompt
        c_int_out = self.c_int_p + int(shift)
        c_frac_out = self.c_frac_p + (shift - int(shift))
        if c_frac_out >= 1:
            c_int_out += 1
        elif c_frac_out < 0:
            c_int_out -=  1
        cs_p = 2*code[np.mod(c_int_out,self.c_period)] - 1
        # Late
        c_int_out = self.c_int_p + int(shift - self.c_delta)
        c_frac_out = self.c_frac_p + ((shift - self.c_delta) \
                                      - int(shift - self.c_delta))
        if c_frac_out >= 1:
            c_int_out += 1
        elif c_frac_out < 0:
            c_int_out -=  1
        cs_l = 2*code[np.mod(c_int_out,self.c_period)] - 1
        return cs_e, cs_p, cs_l
```

```
class CA_search_track(object):
    """
    An object for managing CA code search/acquisition and tracking
    to accomodate the unknown initial code phase for a particular SV.
    Note here the space vehicle number (SVN) is embodied in the CA_code
    index

    Mark Wickert November 2015
    """
    def __init__(self, N_samples, R_chip, f_clk, CA_code, Bn, code_delta = 0.5,
                 Ti = 1e-3, K0 = 1, Kp=2.346):
        """
        Initialize the object
        """
        self.R_chip = R_chip
        self.N_1ms = int(1e-3/(1/f_clk))
        # Store CA_code number
        self.CA_code = CA_code
        self.CA_code_seq = camat[CA_code-1,:] # Zero-based index

        # Initialize tracking loop variables 2D array
        self.trk_vars = np.zeros((5,int(N_samples/self.N_1ms)+10))

        # Tracking state
        self.tracking = False

        # Create Code NCO object
        self.st_code_NCO = code_NCO(self.R_chip,f_clk,code_delta)

        # Initialize correlation integrators (accumulators)
        self.early_int = 0
        self.prompt_int = 0
        self.late_int = 0

        # Code loop parameters
        self.zeta = 0.707 # Damping factor
        self.K0 = K0
        self.Kp = Kp # for Delta = 1/2 or 1.52 for Delta = 1/4
        self.K1 = 4*self.zeta/(self.zeta + 1/(4*self.zeta))*Bn*Ti/Kp/K0
        self.K2 = 4/(self.zeta + 1/(4*self.zeta))**2*(Bn*Ti)**2/Kp/K0
        # Loop filter state variable
        self.code_LF_state = 0
        self.code_filter_error = 0

    def accum_reset(self):
        """
        Reset the early, prompt, and late correlation
        integrators/accumulators
        """
        self.early_int = 0
        self.prompt_int = 0
        self.late_int = 0

    def corr_update(self,rec_sig,corr_chip_offset):
        """
        Update the correlator integrators/accumulators
        """
        e_replica,p_replica,l_replica = \
```

```
                        self.st_code_NCO.code_epl_chip_sample(\
                        corr_chip_offset,self.CA_code_seq)
        self.early_int += rec_sig*e_replica
        self.prompt_int += rec_sig*p_replica
        self.late_int += rec_sig*l_replica

    def code_search(self,k,T1,T2):
        """
        Search for coarse code alignment and verify that coarse
        code alignment is maintained. If the prompt accumulator
        fails to cross T2 re-enter code search.
        """
        if self.tracking == False:
            if np.abs(self.prompt_int) > T1:
                self.tracking = True
        else:
            if np.abs(self.prompt_int) < T2:
                self.tracking = False
                self.code_LF_state = 0.0

        # Fill instrumentation arrays
        if self.tracking == False:
            self.trk_vars[0,k] = abs(self.early_int);
            self.trk_vars[1,k] = abs(self.prompt_int);
            self.trk_vars[2,k] = abs(self.late_int);
            self.trk_vars[3,k] = 0

    def code_fine_track(self,k):
        """
        Close the tracking loop
        """
        # Form the Code alignment error tracking signal
        E2 = np.abs(self.early_int)**2 # Find magnitude squared
        L2 = np.abs(self.late_int)**2 # Find magnitude squared
        if E2+L2 == 0:
            code_error = 0
        else:
            code_error = (E2-L2)/(E2+L2)

        # Fill instrumentation arrays
        self.trk_vars[0,k] = abs(self.early_int)
        self.trk_vars[1,k] = abs(self.prompt_int)
        self.trk_vars[2,k] = abs(self.late_int)
        self.trk_vars[3,k] = code_error
        self.trk_vars[4,k] = (self.st_code_NCO.c_int_ref \
                            + self.st_code_NCO.c_frac_ref)\
                            -(self.st_code_NCO.c_int_p \
                            + self.st_code_NCO.c_frac_p)

        # Code tracking second-order loop filter
        code_lead = self.K1 * code_error
        code_lag = self.K2 * code_error + self.code_LF_state
        self.code_filter_error = code_lead +  code_lag
        self.code_LF_state = code_lag
        #code_filter_error = 0

    def NCO_update(self,n,ss_step = 0.5):
        """
        Update the code phase NCO and apply code error adjustment
```

```
        if needed.
        """
        if int(np.mod(n,self.N_1ms)) == 0 and self.tracking == True:
            print('SV%02d: Fine Track n = %d' % n)
            self.st_code_NCO.update(self.code_filter_error)
        elif int(np.mod(n,self.N_1ms)) == 0 and self.tracking == False:
            print('SV%02d: Search n = %d' % n)
            self.st_code_NCO.step_code(ss_step) # Serial search: Advance code by 1/2
chip
            self.st_code_NCO.update(0.0)
        else:
            #print('No NCO Updates n = %d' % n)
            self.st_code_NCO.update(0.0)

def CA_Track(rec_sig,f_clk,N_Gold,T1,T2,ss_step=0.5,Bn=100,
             code_delta = 0.5,c_offset=0):
    """
    Delay-lock Loop Using 1ms Integration

    trk_vars = CA_Track(rec_sig,f_clk,N_Gold,Bn,chip_delta=0.5,c_offset=0)
    #+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
       rec_sig = received complex baseband signal
         f_clk = sampling clock in sps
        N_Gold = Gold code sequence number
            Bn = Tracking loop bandwidth in Hz
            T1 = Threshold used to declare code acquisition (nominally 2000)
            T2 = Threshold used to return to code serial search
                 (nominally 1000)
       ss_step = Code serial search step size and direction in chips
    chip_delta = early late chip delta, nominally +/- 0.25chip
      c_offset = local code offset at start of the simulation
     #+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      trk_vars = a collection loop signals recorded during
                 the simulation:
                 row 0 = abs(early correlator output)
                 row 1 =  abs(prompt correlator output)
                 row 2 =  abs(late correlator output)
                 row 3 =  DLL discriminator output
                 row 4 =  pseudorange output in microseconds

     #+++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    Mark Wickert November 2015
    """
    R_chip = 1.023e6 # Chips/s
    # 1023 CA-code chips <--> 1ms

    rec_sig = rec_sig[5:]
    N_samples = len(rec_sig)
    N_1ms = int(1e-3/(1/f_clk))
    print('N_samples = %d' % N_samples)
    print('N_1ms = %d' % N_1ms)

    # Starting location in chips of the replica CA code generator.
    # This value is relative the prompt correlator.
    corr_chip_offset = c_offset

    # Initialize signal index counters
    n = 1 # current time index
```

```
    k = 0 # 1ms integration time cycle counter

    # Create CA_search_track object for SV 1
    CA_sv1 = CA_search_track(N_samples,R_chip,f_clk,N_Gold,Bn)
    #CA_sv1.tracking = True

    print('K1 = %6.4e, K2 = %6.4e' % (CA_sv1.K1,CA_sv1.K2))

    while n <= N_samples-1000: # sample-by-sample loop at f_clk rate

        # Process I/Q signals
        # Create 1ms buffers for code correlation integration
        if n > 30*N_1ms:
            CA_sv1.corr_update(rec_sig[n],corr_chip_offset + 0.0)
        else:
            CA_sv1.corr_update(rec_sig[n],corr_chip_offset)

        #////////////////////////////////////////////////////////////////////
        # BEGIN 1ms code correlation (integrator output) processing event loop
        #////////////////////////////////////////////////////////////////////
        if int(np.mod(n,N_1ms)) == 0:
            # Search for proper code phase or continue to verify that
            # code is still coarse aligned
            CA_sv1.code_search(k,T1,T2)
            # Enter fine code tracking if coarse alignment found
            if CA_sv1.tracking == True:
                #print('Doing fine track at k = %d' % k)
                CA_sv1.code_fine_track(k)
            # Reset integrator/accumulator regardless of search or fine track
            CA_sv1.accum_reset()
            k += 1
        #////////////////////////////////////////////////////////////////////
        # END 1ms code correlation (integrator output) processing event loop
        #////////////////////////////////////////////////////////////////////

        #////////////////// Code NCO1 Updating /////////////////
        # Increment code NCO each f_clk period, but input loop error
        # updates every integration period Ti = 1ms
        CA_sv1.NCO_update(n,ss_step)

        n += 1
    print(k)
    return CA_sv1.trk_vars

def plot_tracking_vars(trk_var,x_upper_limit=97):
    """
    Input the trk_var 2D array
    Plot the early, prompt, and late integrator/accumulator
    outputs on one subplot. Plot the tracking loop error
    on a second subplot.

    """
    plt.figure(figsize=(6,6)) # Figure width and height in inches
    plt.subplot(211)
    plt.plot(trk_var[0,:x_upper_limit],'b')
    plt.plot(trk_var[1,:x_upper_limit],'r')
    plt.plot(trk_var[2,:x_upper_limit],'g')
    plt.title(r'1ms Update Early, Prompt, and Late Accumulators')
    plt.ylabel(r'Accumulator Output')
```

```
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    plt.legend((r'E',r'P','L'),loc='best')
    plt.grid();
    plt.subplot(212)
    plt.plot(trk_var[3,:x_upper_limit])
    plt.title(r'Delay-Locked Loop Tracking Error')
    plt.ylabel(r'Normalized Error')
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    #plt.legend((r'Error',),loc='best')
    plt.grid();
    plt.tight_layout()

def plot_tracking_ELP(trk_var,x_upper_limit=97):
    """
    Input the trk_var 2D array
    Plot the early, prompt, and late integrator/accumulator
    outputs.

    """
    plt.figure(figsize=(6,4)) # Figure width and height in inches
    plt.plot(trk_var[0,:x_upper_limit],'b')
    plt.plot(trk_var[1,:x_upper_limit],'r')
    plt.plot(trk_var[2,:x_upper_limit],'g')
    plt.title(r'1ms Update Early, Prompt, and Late Accumulators')
    plt.ylabel(r'Accumulator Output')
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    plt.legend((r'E',r'P','L'),loc='best')
    plt.grid();
    plt.tight_layout()

def plot_tracking_error(trk_var,x_upper_limit=97):
    """
    Input the trk_var 2D array
    Plot the tracking loop error. The max and min error displayed is
    one by design. The actual error in chips is a function of the
    parameter chip_delta. For chip_delta = 0.5 the error +1/-1 error swing
    corresponds to 1/2 chip.

    """
    plt.figure(figsize=(6,4)) # Figure width and height in inches
    plt.plot(trk_var[3,:x_upper_limit])
    plt.title(r'Delay-Locked Loop Tracking Error')
    plt.ylabel(r'Normalized Error')
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    #plt.legend((r'Error',),loc='best')
    plt.grid();
    plt.tight_layout()

def plot_pseudorange(trk_var,x_upper_limit=97):
    """
    Input the trk_var 2D array
    Plot the tracking loop error. The max and min error displayed is
    one by design. The actual error in chips is a function of the
    parameter chip_delta. For chip_delta = 0.5 the error +1/-1 error swing
    corresponds to 1/2 chip.

    """
    plt.figure(figsize=(6,4)) # Figure width and height in inches
    plt.plot(trk_var[4,:x_upper_limit]/1.023)
```

```
    plt.title(r'Pseudorange')
    plt.ylabel(r'$\mu$s')
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    #plt.legend((r'Error',),loc='best')
    plt.grid();
    plt.tight_layout()"""
GPS Function Module

Mark Wickert November 2015 - December 2015

Development continues!
"""
from __future__ import division

"""
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
"""

from matplotlib import pylab
#from matplotlib import mlab
import numpy as np
import ssd
import digitalcom as dc
#from numpy import fft
import matplotlib.pyplot as plt
from scipy import signal
#from sys import exit

# Load the CA code 37 x 1023 2D ndrray into the Python workspace
camat = np.loadtxt('ca1thru37.txt',dtype=np.int16,unpack=True)

def CA_tx(Nchips,Ns,N_Gold,code_shift=0):
    """
    Generate CA signal of given number of chips
    N_Gold = Gold code number: 1 - 37
    """
    N_GP = 1023; # Gold code period
    gold_code = camat[N_Gold-1,:]
    #gold_code = np.arange(1023,dtype=np.int16)
    gold_code_r = np.roll(gold_code,code_shift)
    x = np.zeros(Nchips)
    N_periods = Nchips//N_GP
    N_chip_rem = Nchips - N_GP*N_periods
    for n in xrange(N_periods):
        x[n*N_GP:(n+1)*N_GP] = gold_code_r
    if N_chip_rem > 0:
        x[N_periods*N_GP:] = gold_code_r[:N_chip_rem]
    x_up = ssd.upsample(2*x-1,Ns)
```

```
    x_up = signal.lfilter(np.ones(Ns),1,x_up)
    return x_up

def CA_rx_RTLSDR(Nchips,N_Gold,code_shift=0,
                 fDop = 0.0,f_clk=2.4e6,rphase=True):
    """
    Generate CA signal of given number of chips resampled to f_clk
    N_Gold = Gold code number: 1 - 37

    x_SDR24, x_SDR = CA_rx_RTLSDR(Nchips,N_Gold,code_shift=0,
                                  fDop = 0.0,f_clk=2.4e6,rphase=True)

        Nchips = Number of chips to simulate
        N_Gold = The Gold code number, 1 to 37
    code_shift = roll code starting point forward or backward relative to
                 1023 chip period
          fDop = Applied Doppler frequency shift in Hz
         f_clk = Default 2.4Msps. Output sample rate in sps (note the input to
                 the resampler is a CA code signal at 3 samples per chip or
                 fsamp = 3*1.023 Msps
        rphase = True/False to apply a random phase shift (rotation) to the
                 x_SDR24 signal. Note, if fDop 0 the rphase = False, then the
                 output is real.
    ================================================================
       x_SDR24 = complex baseband GPS signal at f_clk Msps with a fixed
                 random phase and Doppler frequency shift. Normally
                 choose f_clk = 2.4e6.
         x_SDR = A real 3 sample per chip CA signal
    """
    x_SDR = CA_tx(Nchips,3,N_Gold,code_shift)
    n = np.arange(len(x_SDR))
    if rphase:
        x_SDR_rot = x_SDR*np.exp(1j*2*np.pi*np.random.rand(1)) # random phase rotation
        x_SDR_rot = x_SDR_rot*np.exp(1j*2*np.pi*fDop/f_clk*n) # apply Doppler
    else:
        if fDop > 0:
            x_SDR_rot = x_SDR*np.exp(1j*2*np.pi*fDop/f_clk*n) # apply Doppler only
        else:
            x_SDR_rot = x_SDR
    x_SDR24 = dc.farrow_resample(x_SDR_rot,3*1.023e6,f_clk) # asynch resample
    return x_SDR24, x_SDR

# A class to manage the code NCO for early, prompt and late
# correlator timing control
class code_NCO(object):
    """
    A class that manages the code tracking loop numerically
    controlled oscillator for early, prompt, and late correlators.
    The attributes are code integer offset and the code fraction
    for all three correlators, all in chips (multiples of the chip
    period which is one over the chip rate)
    Mark Wickert November 2015
    """
    def __init__(self,R_chip,f_clk,DLL_delta = 0.5,c_period = 1023):
        # NCO center frequency
        self.NCO_CF = R_chip/f_clk #both must have the same units, e.g., MHz
        # Code period in chips
        self.c_period = c_period # code period in chips
        # Delay lock loop early & late time delta relative to prompt
```

```
        self.c_delta = DLL_delta
        # NCO accumulator integer and fractional values
        # The integer value is code chips and the fractional
        # part is fractions of a chip modulo 1
        self.c_int_p = 0
        self.c_frac_p = 0.0
        # Maintain a reference accumulator for easy pseudo-range calculation
        self.c_int_ref = 0
        self.c_frac_ref = 0


    def update(self,c_error):
        """
        Update NCO attributes
        following the loop error calculation
        """
        # Increment the code NCO
        # ******** prompt ***************************
        self.c_frac_p += c_error + self.NCO_CF
        # Check for code NCO overflows/underflows so
        if self.c_frac_p >= 1:
            self.c_int_p += 1
        elif self.c_frac_p < 0:
            self.c_int_p -= 1

        # Modulo 1 the c_frac_p value
        self.c_frac_p = np.mod(self.c_frac_p,1)

        #Update the reference NCO
        self.c_frac_ref += self.NCO_CF
        # Check for code NCO overflows/underflows so
        if self.c_frac_ref >= 1:
            self.c_int_ref += 1
        elif self.c_frac_ref < 0:
            self.c_int_ref -= 1

        # Modulo 1 the c_frac_p value
        self.c_frac_ref = np.mod(self.c_frac_ref,1)

    def step_code(self,step_frac, step_int = 0):
        """
        Step the code NCO by a chip fraction and integer
        as desired. The most likely use of the
        """
        # Increment the code NCO
        # ******** prompt ***************************
        self.c_int_p += step_int
        self.c_frac_p += step_frac
        # Check for code NCO overflows/underflows so
        if self.c_frac_p >= 1:
            self.c_int_p += 1
        elif self.c_frac_p < 0:
            self.c_int_p -= 1

        # Modulo 1 the c_frac_p value
        self.c_frac_p = np.mod(self.c_frac_p,1)

    def code_epl_chip_sample(self,shift,code):
        """
```

```
        Shift the code output sample used in the correlator taking into
        account the NCO integer and fractional offsets and an additional
        shift or offset factor. Note the shift might come from coarse
        acquisition.

        code_shift(shift)
        """

        # ****** shift and apply e, l, or p offset ******
        # Make sure chip samples repeat via the use of mod( ,c_period)
        # Early
        c_int_out = self.c_int_p + int(shift + self.c_delta)
        c_frac_out = self.c_frac_p + ((shift + self.c_delta) \
                                    - int(shift + self.c_delta))
        if c_frac_out >= 1:
            c_int_out += 1
        elif c_frac_out < 0:
            c_int_out -=  1
        cs_e = 2*code[np.mod(c_int_out,self.c_period)] - 1
        # Prompt
        c_int_out = self.c_int_p + int(shift)
        c_frac_out = self.c_frac_p + (shift - int(shift))
        if c_frac_out >= 1:
            c_int_out += 1
        elif c_frac_out < 0:
            c_int_out -=  1
        cs_p = 2*code[np.mod(c_int_out,self.c_period)] - 1
        # Late
        c_int_out = self.c_int_p + int(shift - self.c_delta)
        c_frac_out = self.c_frac_p + ((shift - self.c_delta) \
                                    - int(shift - self.c_delta))
        if c_frac_out >= 1:
            c_int_out += 1
        elif c_frac_out < 0:
            c_int_out -=  1
        cs_l = 2*code[np.mod(c_int_out,self.c_period)] - 1
        return cs_e, cs_p, cs_l

class CA_search_track(object):
    """
    An object for managing CA code search/acquisition and tracking
    to accomodate the unknown initial code phase for a particular SV.
    Note here the space vehicle number (SVN) is embodied in the CA_code
    index

    Mark Wickert November 2015
    """
    def __init__(self, N_samples, R_chip, f_clk, CA_code, Bn, code_delta = 0.5,
                 Ti = 1e-3, K0 = 1, Kp=2.346):
        """
        Initialize the object
        """
        self.R_chip = R_chip
        self.N_1ms = int(1e-3/(1/f_clk))
        # Store CA_code number
        self.CA_code = CA_code
        self.CA_code_seq = camat[CA_code-1,:] # Zero-based index

        # Initialize tracking loop variables 2D array
```

```
        self.trk_vars = np.zeros((5,int(N_samples/self.N_1ms)+10))

        # Tracking state
        self.tracking = False

        # Create Code NCO object
        self.st_code_NCO = code_NCO(self.R_chip,f_clk,code_delta)

        # Initialize correlation integrators (accumulators)
        self.early_int = 0
        self.prompt_int = 0
        self.late_int = 0

        # Code loop parameters
        self.zeta = 0.707 # Damping factor
        self.K0 = K0
        self.Kp = Kp # for Delta = 1/2 or 1.52 for Delta = 1/4
        self.K1 = 4*self.zeta/(self.zeta + 1/(4*self.zeta))*Bn*Ti/Kp/K0
        self.K2 = 4/(self.zeta + 1/(4*self.zeta))**2*(Bn*Ti)**2/Kp/K0
        # Loop filter state variable
        self.code_LF_state = 0
        self.code_filter_error = 0

    def accum_reset(self):
        """
        Reset the early, prompt, and late correlation
        integrators/accumulators
        """
        self.early_int = 0
        self.prompt_int = 0
        self.late_int = 0

    def corr_update(self,rec_sig,corr_chip_offset):
        """
        Update the correlator integrators/accumulators
        """
        e_replica,p_replica,l_replica = \
                            self.st_code_NCO.code_epl_chip_sample(\
                            corr_chip_offset,self.CA_code_seq)
        self.early_int += rec_sig*e_replica
        self.prompt_int += rec_sig*p_replica
        self.late_int += rec_sig*l_replica

    def code_search(self,k,T1,T2):
        """
        Search for coarse code alignment and verify that coarse
        code alignment is maintained. If the prompt accumulator
        fails to cross T2 re-enter code search.
        """
        if self.tracking == False:
            if np.abs(self.prompt_int) > T1:
                self.tracking = True
        else:
            if np.abs(self.prompt_int) < T2:
                self.tracking = False
                self.code_LF_state = 0.0

        # Fill instrumentation arrays
        if self.tracking == False:
```

```
            self.trk_vars[0,k] = abs(self.early_int);
            self.trk_vars[1,k] = abs(self.prompt_int);
            self.trk_vars[2,k] = abs(self.late_int);
            self.trk_vars[3,k] = 0

    def code_fine_track(self,k):
        """
        Close the tracking loop
        """
        # Form the Code alignment error tracking signal
        E2 = np.abs(self.early_int)**2 # Find magnitude squared
        L2 = np.abs(self.late_int)**2 # Find magnitude squared
        if E2+L2 == 0:
            code_error = 0
        else:
            code_error = (E2-L2)/(E2+L2)

        # Fill instrumentation arrays
        self.trk_vars[0,k] = abs(self.early_int)
        self.trk_vars[1,k] = abs(self.prompt_int)
        self.trk_vars[2,k] = abs(self.late_int)
        self.trk_vars[3,k] = code_error
        self.trk_vars[4,k] = (self.st_code_NCO.c_int_ref \
                            + self.st_code_NCO.c_frac_ref)\
                            -(self.st_code_NCO.c_int_p \
                            + self.st_code_NCO.c_frac_p)

        # Code tracking second-order loop filter
        code_lead = self.K1 * code_error
        code_lag = self.K2 * code_error + self.code_LF_state
        self.code_filter_error = code_lead +  code_lag
        self.code_LF_state = code_lag
        #code_filter_error = 0

    def NCO_update(self,n,ss_step = 0.5):
        """
        Update the code phase NCO and apply code error adjustment
        if needed.
        """
        if int(np.mod(n,self.N_1ms)) == 0 and self.tracking == True:
            print('SV%02d: Fine Track n = %d' % n)
            self.st_code_NCO.update(self.code_filter_error)
        elif int(np.mod(n,self.N_1ms)) == 0 and self.tracking == False:
            print('SV%02d: Search n = %d' % n)
            self.st_code_NCO.step_code(ss_step) # Serial search: Advance code by 1/2
chip
            self.st_code_NCO.update(0.0)
        else:
            #print('No NCO Updates n = %d' % n)
            self.st_code_NCO.update(0.0)

def CA_Track(rec_sig,f_clk,N_Gold,T1,T2,ss_step=0.5,Bn=100,
             code_delta = 0.5,c_offset=0):
    """
    Delay-lock Loop Using 1ms Integration

    trk_vars = CA_Track(rec_sig,f_clk,N_Gold,Bn,chip_delta=0.5,c_offset=0)
    #+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
       rec_sig = received complex baseband signal
```

```
        f_clk = sampling clock in sps
      N_Gold = Gold code sequence number
          Bn = Tracking loop bandwidth in Hz
          T1 = Threshold used to declare code acquisition (nominally 2000)
          T2 = Threshold used to return to code serial search
               (nominally 1000)
     ss_step = Code serial search step size and direction in chips
  chip_delta = early late chip delta, nominally +/- 0.25chip
    c_offset = local code offset at start of the simulation
  #+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    trk_vars = a collection loop signals recorded during
               the simulation:
               row 0 = abs(early correlator output)
               row 1 =  abs(prompt correlator output)
               row 2 =  abs(late correlator output)
               row 3 =  DLL discriminator output
               row 4 =  pseudorange output in microseconds

  #+++++++++++++++++++++++++++++++++++++++++++++++++++++++++

  Mark Wickert November 2015
  """
  R_chip = 1.023e6 # Chips/s
  # 1023 CA-code chips <--> 1ms

  rec_sig = rec_sig[5:]
  N_samples = len(rec_sig)
  N_1ms = int(1e-3/(1/f_clk))
  print('N_samples = %d' % N_samples)
  print('N_1ms = %d' % N_1ms)

  # Starting location in chips of the replica CA code generator.
  # This value is relative the prompt correlator.
  corr_chip_offset = c_offset

  # Initialize signal index counters
  n = 1 # current time index
  k = 0 # 1ms integration time cycle counter

  # Create CA_search_track object for SV 1
  CA_sv1 = CA_search_track(N_samples,R_chip,f_clk,N_Gold,Bn)
  #CA_sv1.tracking = True

  print('K1 = %6.4e, K2 = %6.4e' % (CA_sv1.K1,CA_sv1.K2))

  while n <= N_samples-1000: # sample-by-sample loop at f_clk rate

      # Process I/Q signals
      # Create 1ms buffers for code correlation integration
      if n > 30*N_1ms:
          CA_sv1.corr_update(rec_sig[n],corr_chip_offset + 0.0)
      else:
          CA_sv1.corr_update(rec_sig[n],corr_chip_offset)

      #////////////////////////////////////////////////////////////////
      # BEGIN 1ms code correlation (integrator output) processing event loop
      #////////////////////////////////////////////////////////////////
      if int(np.mod(n,N_1ms)) == 0:
          # Search for proper code phase or continue to verify that
```

```
            # code is still coarse aligned
            CA_sv1.code_search(k,T1,T2)
            # Enter fine code tracking if coarse alignment found
            if CA_sv1.tracking == True:
                #print('Doing fine track at k = %d' % k)
                CA_sv1.code_fine_track(k)
            # Reset integrator/accumulator regardless of search or fine track
            CA_sv1.accum_reset()
            k += 1
        #////////////////////////////////////////////////////////////////////
        # END 1ms code correlation (integrator output) processing event loop
        #////////////////////////////////////////////////////////////////////

        #////////////////// Code NCO1 Updating /////////////////
        # Increment code NCO each f_clk period, but input loop error
        # updates every integration period Ti = 1ms
        CA_sv1.NCO_update(n,ss_step)

        n += 1
    print(k)
    return CA_sv1.trk_vars

def plot_tracking_vars(trk_var,x_upper_limit=97):
    """
    Input the trk_var 2D array
    Plot the early, prompt, and late integrator/accumulator
    outputs on one subplot. Plot the tracking loop error
    on a second subplot.

    """
    plt.figure(figsize=(6,6)) # Figure width and height in inches
    plt.subplot(211)
    plt.plot(trk_var[0,:x_upper_limit],'b')
    plt.plot(trk_var[1,:x_upper_limit],'r')
    plt.plot(trk_var[2,:x_upper_limit],'g')
    plt.title(r'1ms Update Early, Prompt, and Late Accumulators')
    plt.ylabel(r'Accumulator Output')
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    plt.legend((r'E',r'P','L'),loc='best')
    plt.grid();
    plt.subplot(212)
    plt.plot(trk_var[3,:x_upper_limit])
    plt.title(r'Delay-Locked Loop Tracking Error')
    plt.ylabel(r'Normalized Error')
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    #plt.legend((r'Error',),loc='best')
    plt.grid();
    plt.tight_layout()

def plot_tracking_ELP(trk_var,x_upper_limit=97):
    """
    Input the trk_var 2D array
    Plot the early, prompt, and late integrator/accumulator
    outputs.

    """
    plt.figure(figsize=(6,4)) # Figure width and height in inches
    plt.plot(trk_var[0,:x_upper_limit],'b')
    plt.plot(trk_var[1,:x_upper_limit],'r')
```

```
    plt.plot(trk_var[2,:x_upper_limit],'g')
    plt.title(r'1ms Update Early, Prompt, and Late Accumulators')
    plt.ylabel(r'Accumulator Output')
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    plt.legend((r'E',r'P','L'),loc='best')
    plt.grid();
    plt.tight_layout()

def plot_tracking_error(trk_var,x_upper_limit=97):
    """
    Input the trk_var 2D array
    Plot the tracking loop error. The max and min error displayed is
    one by design. The actual error in chips is a function of the
    parameter chip_delta. For chip_delta = 0.5 the error +1/-1 error swing
    corresponds to 1/2 chip.

    """
    plt.figure(figsize=(6,4)) # Figure width and height in inches
    plt.plot(trk_var[3,:x_upper_limit])
    plt.title(r'Delay-Locked Loop Tracking Error')
    plt.ylabel(r'Normalized Error')
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    #plt.legend((r'Error',),loc='best')
    plt.grid();
    plt.tight_layout()

def plot_pseudorange(trk_var,x_upper_limit=97):
    """
    Input the trk_var 2D array
    Plot the tracking loop error. The max and min error displayed is
    one by design. The actual error in chips is a function of the
    parameter chip_delta. For chip_delta = 0.5 the error +1/-1 error swing
    corresponds to 1/2 chip.

    """
    plt.figure(figsize=(6,4)) # Figure width and height in inches
    plt.plot(trk_var[4,:x_upper_limit]/1.023)
    plt.title(r'Pseudorange')
    plt.ylabel(r'$\mu$s')
    plt.xlabel(r'Time (in 1ms Accumulation Increments)')
    #plt.legend((r'Error',),loc='best')
    plt.grid();
    plt.tight_layout()
```

Hints

Task 1

Part a

Here you calculate the error signal directly from the given equation. Given the way I have defined the error characteristic and the fact that negative feedback is used in the DLL, please plot $-e(t)$. Note in synchronization theory this characteristic is known the *s-curve*. The slope of the plotted error characteristic will have a positive slope that crosses through $t = 0$. The actual slope value is used as a gain constant in the control system modeling of the DLL.

Part b

The plot you obtain in this part should almost perfectly match the theoretical *s-curve* of part (a). The starting point is data held in the array variable `trk_vars_sv1`, as found in the code cell used to generate the plot at the top of page 14 of this document. The early and late waveforms in this plot are used here to replace the ideal waveforms of part (a). To obtain a properly scaled time axis you need to find the array index corresponding to the peak of the prompt correlator output and taking into account the fact that the time step is 0.05 chips. To find the index of the peak you can enter a logical expression into the function below and have it search over an `ndarray`:

```
def find_idx_array(arg_cond):
    idx_array = np.nonzero(np.ravel(arg_cond))[0]
    return idx_array
```

Quick example of usage:

```
n_test = arange(100)
z_test = 50 - (n_test - 35)**2
# Show numerically that the max value of z_test is at 35
idx = find_idx_array(z_test == max(z_test))
idx
```

Simple parabola

```
array([35], dtype=int64)
```

Task 2

No additional comments at this time.

Task 3

To save paper in your documented results of (3) or even in (1b) please edit the logs that result when running a long simulation. If you are using Typora for example you can later trim the logged information down to just time intervals where interesting activity is occurring, in particular when code track commences for a particular SV signal.