Exploring the Extended Kalman Filter for GPS Positioning Using Simulated User and Satellite Track Data

Mark Wickert‡*, Chiranth Siddappa‡

Abstract—This paper describes a Python computational tool for exploring the use of the extended Kalman filter (EKF) for position estimation using the Global Positioning System (GPS) pseudorange measurements. The development was motivated by the need for an example generator in a training class on Kalman filtering, with emphasis on GPS. In operation of the simulation framework both user and satellite trajectories are played through the simulation. The User trajectory is input in local east-north-up (ENU) coordinates and satellites tracks, specified by the C/A code PRN number, are propagated using the Python package SGP4 using two-line element (TLE) data available from [Celestrak].

Index Terms—Global positioning system, Kalman filter, Extended Kalman filter,

Introduction

The Global Positioning System (GPS) allows user position estimation using time difference of arrival (TDOA) measurements from signals received from a constellation of 24 medium earth orbit satellites of space vehicles (SVs). The Kalman filter is a popular optimal *state estimation* algorithm [Simon2006] used by a variety of engineering and science disciplines. In particular the extended Kalman filter (EKF) is able to deal with nonlinearities related to both the measurement equations and state vector process update model. The EKF used in GPS has a linear process model, but a nonlinear measurement model [Brown2012]. This paper describes a Python computational tool for exploring the use of the EKF for GPS position estimation using pseudorange measurements. The development was motivated by the need for an example generator in a training class on Kalman filtering, with emphasis on GPS. What is special about the tool created here is that both *User* and satellite trajectories are custom generated for input to a Kalman filter implemented in a Jupyter notebook. The steps followed are logical and clear. You first enter a desired *User* trajectory/route, then choose appropriate *in-view* GPS satellites, and then using actual GPS satellite orbital mechanics information, create a simulated receiver measurement stream. A 3D plot shows you the satellite tracks in space and the User trajectory on the surface of the earth, over time. The Kalman filter code, also defined in the Jupyter notebook, uses the matrix math commonly found in textbooks, but it is easy to follow as we make use of the PEP 465 @ infix operator for matrix multiplication. As the final step, the data set is played through the Kalman filter in earth-centered earth-fixed (ECEF) coordinates. The User trajectory is input in local east-north-up (ENU) coordinates, and the SVs in view by the User to form the location estimate, are specified by the coarse acquisition (C/A) code pseudo-random noise (PRN) number. The ECEF coordinates of the SVs are then propagated using [SGP4] using the two-line element (TLE) data available from [Celestrak], in time step with the User trajectory. The relationship between ECEF and ENU is explained in Figure 1. For convenience, this computational tool, is housed in a Jupyter notebook. Data set generation and 3D trajectory plotting is provided with the assistance of a single module, [GPS_helper].

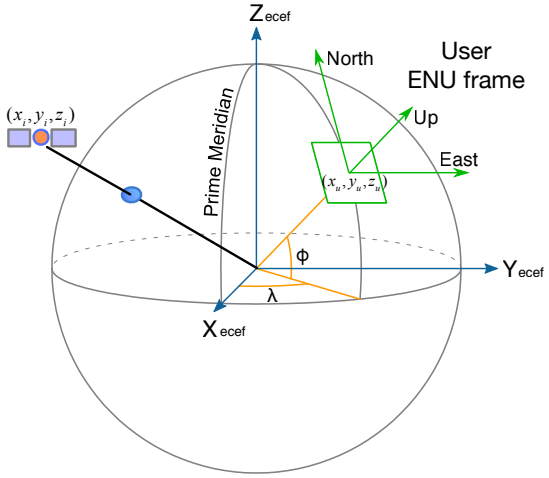

Fig. 1: *The earth centric earth fixed (ECEF) coordinate system compared with the local east-north-up (ENU) coordinate system.*

GPS Background

GPS was started in 1973 with the first block of satellites launched over the 1978 to 1985 time interval [GPS]. 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 original design called for 24 satellites. The satellites 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

* *Corresponding author: mwickert@uccs.edu*
‡ *University of Colorado Colorado Springs*

(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 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 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.

Waveform Design and Pseudorange Measurements

Time difference of arrival (TDOA) is the key to forming the User position estimates. This starts by assigning a unique repeating code of 1023 bits to each SV and corresponds to the L1 carrier waveform it transmits. As the User receives the superposition of all the *in-view* satellites, the code known by its PRN number assigned to a particular satellite, is discernable by cross-correlating the composite received L1 signal and a locally generated PRN waveform. The correlation peak and its associated TDOA, become the *pseudorange* or approximate radial distance between the User and SV when multipled by c, the speed of light.

The pseudorange contains error due to the receiver clock offset from the satellite time and other error components [Brown2012]. The noise-free pseudorange takes the form

$$\rho_i = \sqrt{(x_i - x_u)^2 + (y_i - y_u)^2 + (z_i - z_u)^2} + c\Delta t \qquad (1)$$

where (x_i, y_i, z_i), $i = 1, \ldots 4$, is the satellite ECEF location and (x_u, y_u, z_u) is the user ECEF location, c is the speed of light, and Δt is the receiver offset from satellite time. The product $c\Delta t$ can be thought of as the *range equivalent* timing error. There are three geometry unknowns and time offset, thus at minimum there are four non-linear equations of (1) are what must be solved to obtain the User location.

Solving the Nonlinear Position Equations

Two techniques are widely discussed in the literature and applied in practice [GPS] and [Kaplan]: (1) nonlinear least squares and (2) the extended Kalman filter (EKF). In this paper we focus on the use of the EKF. The EKF is an extension to the linear Kalman filter, so we start by briefly describing the linear model case and move quickly to the nonlinear case.

Kalman Filter and State Estimation

It was back in 1960 that R. E. Kalman introduced his filter [Kalman]. It immediately became popular in guidance, navigation, and control applications. The Kalman filter is an optimal, in the minimum mean-squared error sense, as means to estimate the *state* of a dynamical system [Simon2006]. By state we mean a vector of variables that adequately describes the dynamical behavior of a system over time. For the GPS problem a simplifying assumption regarding the state model is to assume that the User has approximately constant velocity, so a position-velocity (PV) only state model is adequate. The Kalman filter is recursive, meaning that the estimate of the state is refined with each new input measurement and without the need to store all of the past measurements.

Within the Kalman filter we have a *process model* and a *measurement model*. The *process equation* associated with the process model, describes how the state is updated through a state transition matrix plus a process noise vector having covariance matrix $\mathbf{Q}$. The *measurement model* contains the *measurement equation* that abstractly produces the measurement vector as a matrix times the state vector plus a measurement noise vector having covariance matrix $\mathbf{R}$. The optimal recursive filter algorithm is formed using the quantities that make up the process and measurement models. For details the reader is referred to the references.

For readers wanting a hands-on beginners introduction to the Kalman filter, a good starting point is the book by Kim [Kim2011]. In Kim's book the Kalman filter is neatly represented input/output block diagram form as shown in Figure 2, with the input being the vector of measurements $\mathbf{z}_k$, at time k, and the output $\hat{\mathbf{x}}_k$ an updated estimate of the state vector. The Kalman filter variables are defined in Table 1. Note the dimensions seen in Table 1 are $n =$ number of state variables and $m =$ number of measurements.

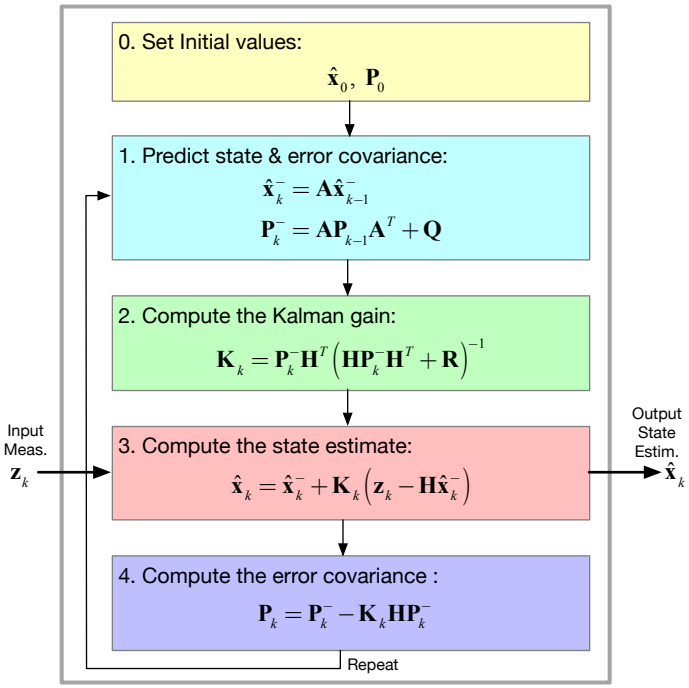

Fig. 2: *General Kalman filter block diagram.*

State Vector for the GPS Problem

For a PV model the User state vector position and velocity in x, y, z and clock equivalent range and range velocity error [Brown2012]:

$$\begin{aligned}\mathbf{x} &= \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \end{bmatrix} \\ &= \begin{bmatrix} x & \dot{x} & y & \dot{y} & z & \dot{z} & c\Delta t & c\dot{\Delta} t \end{bmatrix} \end{aligned} \qquad (2)$$

where ECEF coordinates are assumed and the over dots denote the time derivative, e.g., $\dot{x} = dx/dt$. We further assume that there is no coupling between $x, y, z, c\Delta t$, thus the state transition matrix $\mathbf{A}$ is a 4×4 block diagonal matrix of the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{cv} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{cv} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{cv} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_{cv} \end{bmatrix} \qquad (3)$$

State Estimate (output)	
$\hat{\mathbf{x}}_k$ $(n \times 1)$	State estimate at time k
Measurement (input)	
$\mathbf{z}_k$ $(m \times 1)$	Measurement at time k
System Model	
$\mathbf{A}$ $(n \times n)$	State transition matrix
$\mathbf{H}$ $(m \times n)$	Measurement matrix
$\mathbf{Q}$ $(n \times n)$	State error autocovariance matrix
$\mathbf{R}$ $(m \times m)$	Measurement error autocovariance matrix
Internal Comp. Quant.	
$\mathbf{K}_k$ $(n \times m)$	Kalman gain
$\mathbf{P}_k$ $(n \times n)$	Estimate of error covariance matrix
$\hat{\mathbf{x}}_k^-$ $(n \times 1)$	Prediction of the state estimate
$\mathbf{P}_k^-$ $(n \times n)$	Prediction of error covariance matrix

TABLE 1: *The Kalman filter variables and a brief description.*

where

$$\mathbf{A}_{cv} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{4}$$

Process Model Covariance Matrix

The process covariance matrix for the GPS problem is a block diagonal Matrix, with three identical blocks for the position-velocity pairs and one matrix for the clock-clock drift pair. The block diagonal form means that the states are assumed be statistically coupled only in pairs and outside of the pairs uncorrelated. In the model of [Brown2012] each position-velocity state-pair has two variance terms and one covariance term describing an upper triangle 2×2 submatrix

$$\mathbf{Q}_{xyz} = \sigma_{xyz}^2 \begin{bmatrix} \frac{\Delta t^3}{3} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^2}{2} & \Delta t \end{bmatrix} \tag{5}$$

where σ_{xyz}^2 is a white noise spectral density representing random walk velocity error. The clock state variable pair has a 2×2 covariance matrix governed by S_p, the white noise spectral density leading to random walk velocity error. The clock and clock drift has a more complex 2×2 covariance submatrix, $\mathbf{Q}_b$, with S_g the white noise spectral density leading to a random walk clock frequency error plus white noise clock drift, thus two components of clock phase error

$$\mathbf{Q}_b = \begin{bmatrix} S_f \Delta t + \frac{S_g \Delta t^3}{3} & \frac{S_g \Delta t^2}{2} \\ \frac{S_g \Delta t^2}{2} & S_g \Delta t \end{bmatrix} \tag{6}$$

In final form $\mathbf{Q}$ is a 4×4 block covariance matrix

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{xyz} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_{xyz} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}_{xyz} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{Q}_b \end{bmatrix} \tag{7}$$

Measurement Model Covariance Matrix

The covariance matrix of the pseudorange measurement error is assumed to be diagonal with equal variance σ_r^2, thus we have

$$\mathbf{R} = \begin{bmatrix} \sigma_r^2 & 0 & 0 & 0 \\ 0 & \sigma_r^2 & 0 & 0 \\ 0 & 0 & \sigma_r^2 & 0 \\ 0 & 0 & 0 & \sigma_r^2 \end{bmatrix} \tag{8}$$

for the case of $m = 4$ measurements. Being diagonal means that all measurements are assumed statistically uncorrelated, which is reasonable.

Extended Kalman Filter

The extended Kalman filter (EKF) allows both the state update equation, Step 1 in Figure 2, to be a nonlinear function of the state, and the measurement model, Step 3 in Figure 2, to be a nonlinear function of the state. Thus the EKF block diagram replaces two expressions in Figure 2 as follows:

$$\mathbf{A}\hat{\mathbf{x}}_{k-1} \longrightarrow \mathbf{f}(\hat{\mathbf{x}}_{k-1}) \tag{9}$$

$$\mathbf{H}\hat{\mathbf{x}}_{k-1}^- \longrightarrow \mathbf{h}(\hat{\mathbf{x}}_{k-1}^-) \tag{10}$$

For the case of the GPS problem we have already seen that the state transition model is linear, thus the first calculation of **Step 1**, *predicted state update expression*, is the same as that found in the standard linear Kalman filter. For **Step 3**, the state estimate, we need to linearize the equations $\mathbf{h}(\hat{\mathbf{x}}_k^-)$. This is done by forming a matrix of partials or Jacobian matrix, which then generates an equivalent $\mathbf{H}$ matrix as found in the linear Kalman filter, but in the EKF is updated at each iteration of the algorithm.

$$\mathbf{H} = \left.\frac{\partial \mathbf{h}}{\mathbf{x}}\right|_{\mathbf{x}=\hat{\mathbf{x}}_k^-} \tag{11}$$

$$= \begin{bmatrix} \frac{\partial \rho_1}{\partial x} & 0 & \frac{\partial \rho_1}{\partial y} & 0 & \frac{\partial \rho_1}{\partial z} & 0 & 1 & 0 \\ \frac{\partial \rho_2}{\partial x} & 0 & \frac{\partial \rho_2}{\partial y} & 0 & \frac{\partial \rho_2}{\partial z} & 0 & 1 & 0 \\ \frac{\partial \rho_3}{\partial x} & 0 & \frac{\partial \rho_3}{\partial y} & 0 & \frac{\partial \rho_3}{\partial z} & 0 & 1 & 0 \\ \frac{\partial \rho_4}{\partial x} & 0 & \frac{\partial \rho_4}{\partial y} & 0 & \frac{\partial \rho_4}{\partial z} & 0 & 1 & 0 \end{bmatrix} \tag{12}$$

where

$$\frac{\partial \rho_i}{\partial x} = \frac{-(x_i - \hat{x}_1^-)}{\sqrt{(x_i - \hat{x}_1^-)^2 + (y_i - \hat{x}_3^-)^2 + (z_i - \hat{x}_5^-)^2}} \tag{13}$$

$$\frac{\partial \rho_i}{\partial y} = \frac{-(y_i - \hat{x}_3^-)}{\sqrt{(x_i - \hat{x}_1^-)^2 + (y_i - \hat{x}_3^-)^2 + (z_i - \hat{x}_5^-)^2}} \tag{14}$$

$$\frac{\partial \rho_i}{\partial z} = \frac{-(z_i - \hat{x}_5^-)}{\sqrt{(x_i - \hat{x}_1^-)^2 + (y_i - \hat{x}_3^-)^2 + (z_i - \hat{x}_5^-)^2}} \tag{15}$$

for $i = 1, 2, 3$ and 4.

Computational Tool

The Python computational tool is composed of a Jupyter notebook and a helper module `GPS_helper.py`. The key elements of the helper are described in Figure 3. Here we see that the class `GPS_data_source` is responsible for propagating the SVs in view by the User in time-step with a constant velocity *line segment* User trajectory. The end result is a collection of matrices (ndarrays) that contain the ECEF User coordinates as the triples (x_u, y_u, z_u) versus times (also the ENU version) and for each SV indexed as $i = 1, 2, 3, 4$, the ECEF triples (x_i, y_i, z_i), also as a function of time. The time step value is T_ss.

It is important to note that in creating a data set the developer must choose satellite PRNs that place the SVs in view of the user for the given start time and date. One approach is by trial and error. Pick a particular time and date, choose four PRNs, and produce the data set and create a 3D plot using `GPS_helper.SV_User_Traj_3D()`. This is quite tedious! A better approach is to use a GPS cell phone app, or better yet a

Module: GPS_helper.py	
Class: GPS_data_source	**Inputs/Outputs**
Constructor( ):	(0) GPS TLE text file from Celestrak as 'GPS_tle.txt' (1) List of SVs in view by User as 'PRN #' (2) User Reference Location as LAT, LONG, ALT (3) Sampling Period (default = 1s)
user_traj_gen( ):	(0) Route (a list of 2D nodes in ENU mi) (1) User velocity in mph GMT trajectory start time (2-6): (2) Year (2k year, i.e., 2018 -> 18) (3) Month (4) Day (5) Hour (6) Minute
returns:	(0) User position in ENU (ndarray) vs time (1) User position in ECEF (ndarray) vs time (2) SV position (ndarray) vs time (3) SV velocity (ndarray) vs time
Functions:	**Inputs/Outputs**
SV_User_Traj_3D( ): (displays 3D plot)	(0) GPS data source object (1) SV position ndarray (2) User position ndarray (3) 3D plot ALT = 20 (4) 3D plot AZIM = 20
returns:	none

Fig. 3: *Of significance the helper module, `GPS_helper.py`, contains a class and a 3D plotting function that supports time-varying data set generation of satellite positions and the corresponding* User *trajectory.*

Kalman GPS Jupyter Notebook Classes	
Class: GetPseudoRange	**Inputs/Outputs**
Constructor( ):	(0) Pseudorange std. dev. (default 0) (1) Pseudo range bias CDt (default 0) (2) Number of satellites in view (default 4)
measurement( ):	(0) User position in ECEF (ndarray) vs time (1) Satellite (SV) position in ECEF (ndarray) vs time
returns:	none, but USER_SR (ndarray) is filled
Class: GPS_EKF	**Inputs/Outputs**
Constructor( ):	(0) User initial position in ECEF (1) Time step (default 1s) (2) Process model diagonal covariance (3) Clock drift random phase walk (default 36) (4) Clock drift random frequency walk (default 0.01) (5) Pseudorange measurement variance (default 36) (6) Number of satellites in view (default 4)
next_sample( )	(0) User position ECEF at time step *k* (1) Satellite (SV) positions ECEF at time step *k*
returns:	none, none but all EKF attributes updated

Fig. 5: *Jupyter notebook classes that synthesize pseudorange test vectors from the time-varying data set created by `GPS_helper.py`, and implement the extended Kalman filter for estimating the time-varying User position.*

stand-alone GPS that displays a map with PRN numbers of what SVs are in view and their signal strengths. An example from a Garmin GPSmap 60CSx [Garmin] is shown in Figure 4 The time and date used in the simulation then corresponds to the time and date of the actual app measurements. A current TLE set should also be obtained from Celestrak.

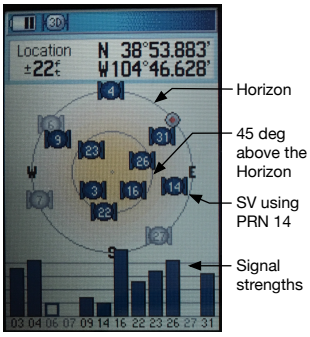

Fig. 4: *SV map of satellites in use on a commercial GPS receiver.*

With a data set generated the next step is to generate pseudorange measurements, as the real GPS receiver would obtain TDOAs via waveform cross-correlation with a local version of the SVs PRN sequence. Finally, we estimate the user position using the EKF. Classes for both these calculations are contained the Jupyter notebook `Kalman_GPS_practice`. A brief description of the two classes in given in Figure 5.

The mathematical details of the EKF were discussed earlier, the Python code implementation is found in the public and private methods of the `GPS_EKF` class. The essence of Figure 2 is the code in the `update()` method:

```
def next_sample(self, z, SV_Pos):
    """
    Update the Kalman filter state by inputting a
    new set of pseudorange measurements.
    Return the state array as a tuple.
    Update all other Kalman filter quantities
    Input SV ephemeris at one time step, e.g.,
    SV_Pos[:,:,i]
    """
    # H = Matrix of partials dh/dx
    H = self.Hjacob(self.x, SV_Pos)

    xp = self.A @ self.x
    Pp = self.A @ self.P @ self.A.T + self.Q

    self.K = Pp @ H.T @ inv(H @ Pp @ H.T + self.R)

    # zp = h(xp), the predicted pseudorange
    zp = self.hx(xp, SV_Pos)

    self.x = xp + self.K @ (z - zp)
    self.P = Pp - self.K @ H @ Pp
    # Return the x,y,z position
    return self.x[0,0], self.x[2,0], self.x[4,0]
```

Note the above code uses the Python 3.5+ matrix multiplication operator, @, to make the code nearly match the matrix algebra expressions of Figure 2.

Simulation Examples

In this section we consider two examples of using the Python framework to estimate a time-varying User trajectory using a time-varying set of GPS satellites. In the code snippets that follow were extracted from a Jupyter notebook that begins with the magic `%pylab inline`, hence the namespace is filled with `numpy` and `matplotlib`.

We start by creating a line segment user trajectory with ENU tagging, followed by a GPS data source using TLEs date 1/10/2018, and finally, populate User and satellite (SV) `ndarrays` using the `user_traj_gen()` method:

```
# Line segment User Trajectory
rl1 = [('e',.2),('n',.4),('e',-0.1),('n',-0.2),
       ('e',-0.1),('n',-0.1)]
```

```
# Create a GPS data source
GPS_ds1 = GPS.GPS_data_source('GPS_tle_1_10_2018.txt',
          Rx_sv_list = \
          ('PRN 32','PRN 21','PRN 10','PRN 18'),
          ref_lla=(38.8454167, -104.7215556, 1903.0),
          Ts = 1)
# Populate User and SV trajectory matrices
# Populate User and SV trajectory matrices
USER_vel = 5 # mph
USER_Pos_enu, USER_Pos_ecf, SV_Pos, SV_Vel = \
   GPS_ds1.user_traj_gen(route_list=rl1,
                         Vmph=USER_vel,
                         yr2=18,
                         mon=1,
                         day=15,
                         hr=8+7,     # 1/18/2018
                         minute=45) # 8:45 AM MDT
```

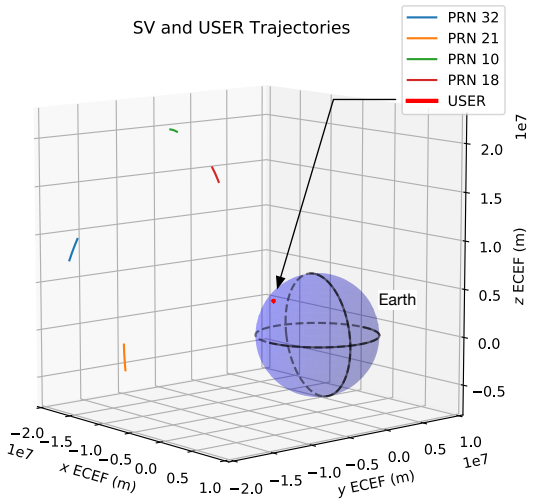

Fig. 6: *A 3D plot of the SV trajectories using* `PRN 32`*,* `PRN 21`*,* `PRN 10`*, and* `PRN 18`*, and the User trajectory over 13.2 min in ECEF, dated 8:45 AM MDT on 1/18/2018.*

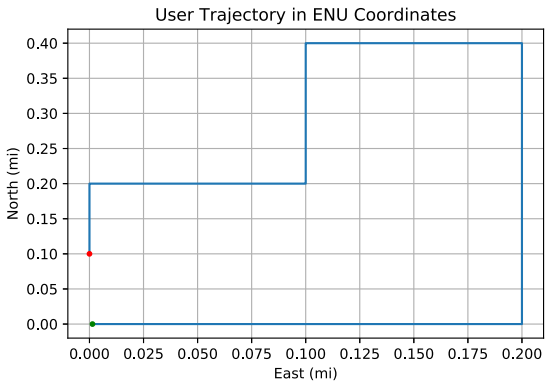

Fig. 7: *The ideal user trajectory as defined by* `rl1` *in the above code snippet.*

The 3D plot 6 shows clearly the motion of the SVs, even though the simulation run-time is only 13.2 min. The User trajectory on the earth, in this case a location in Colorado Springs, CO appears as a red blob, unless the plot is zoomed in. From the ENU User trajectory we now have a clear view of the route taken by the user. The velocity is only 5 mph in straight line segments.

Case #1

With the data set created we now construct an EKF simulation for estimating the User trajectory from the measured pseudoranges for four SVs. Specifically we consider high quality satellite signals, with measurement update period $T_s = 1$s, and constant velocity $V_{\text{User}} = 5$ mph. The simulation code, as taken from a Jupyter notebook cell, is given below:

```
Nsamples = SV_Pos.shape[2]
print('Sim Seconds = %d' % Nsamples)
dt = 1
# Save user position history
Pos_KF = zeros((Nsamples,3))
# Save history of error covariance matrix diagonal
P_diag = zeros((Nsamples,8))

Pseudo_ranges1 = GetPseudoRange(PR_std=0.1,
                                CDt=0,
                                N_SV=4)
GPS_EKF1 = GPS_EKF(USER_xyz_init=USER_Pos_ecf[0,:]
                   + 5*randn(3),
                   dt=1,
                   sigma_xyz=5,
                   Sf=36,
                   Sg=0.01,
                   Rhoerror=36,
                   N_SV=4)
for k in range(Nsamples):
    Pseudo_ranges1.measurement(USER_Pos_ecf[k,:],
                               SV_Pos[:,:,k])
    GPS_EKF1.next_sample(Pseudo_ranges1.USER_PR,
                         SV_Pos[:,:,k])
    Pos_KF[k,:] = GPS_EKF1.x[0:6:2,0]
    P_diag[k,:] = GPS_EKF1.P.diagonal()
```

With the simulation complete, we now consider the ECEF errors in m in Figure 8 for m for `(x,y,z)` components. The initial position *guess* in this example has a standard deviation of 5 m (or variance of 25 meters-squared), so we see that from the start of the tracking the errors are relatively rather small and then settle down to peak errors of *pm*1 m, or so.

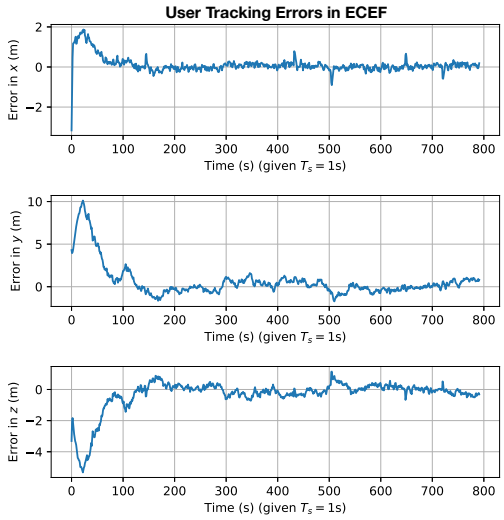

Fig. 8: *ECEF errors in position estimation for Case #1.*

Figure 9 shows selected error covariance matrix terms from $\mathbf{P}_k$ throughout the simulation. The terms displayed are the position diagonal terms, that is σ_x^2, σ_y^2, and σ_z^2. The initial conditions of the EKF make these variance terms initially large. Settling begins about 50s into the simulation, and the decay continues as the 13.2 m simulation comes to an end. The EKF is behaving as expected.

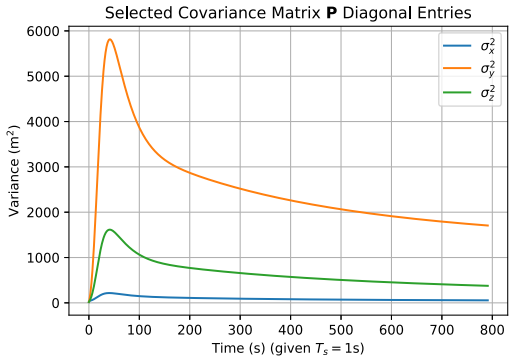

Fig. 9: *Selected error covariance matrix terms, in particular the diagonal elements* σ_x^2, σ_y^2, σ_z^2.

Finally, in Figure 10 we have a plot of the User trajectory estimate in ENU, as a map-like 2D plot showing just the east-west and north-south axes. The units are tenths of miles, so with the User moving along linear line segments at just 5 mph, the trajectory looks perfect.

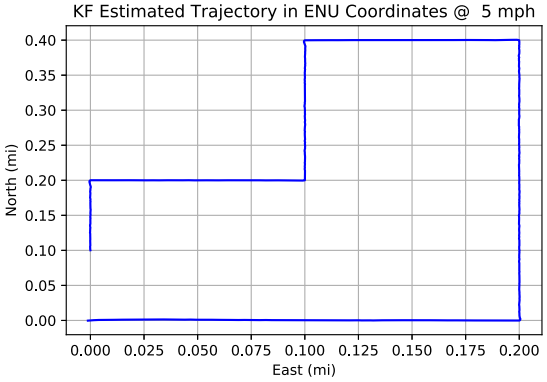

Fig. 10: *The estimated user trajectory in ENU coordinates and the same scale as Figure 7.*

In the next example parameters will be varied to see the impact.

Case #2

In this case we still consider high quality satellite signals and a 1s update period, but now the user velocity is increased to 30 mph, so the time to traverse the User trajectory is reduced from 13.2 min down to 2.2 min. The random initial (xyz) position is set to a error standard deviation of 50 m compared with 5 m in the first case. We expect to see some difference in performance.

In Figure 11 we again plot the ECEF errors in m. The large initial position error variance forces the plot axes scale to change from Case #1. The initial errors are now very large, but do settle to small values with the exception of *blips* that occur every time the user changes direction by making a 90° turn. The blips are somewhat artificial, since making a perfect right-angle turn without slowing or *rounding* the corner is more practical. Still it is interesting to see this behavior and also see that the EKF recovers from these errors.

Figure 12 again shows the error covariance terms for σ_x^2, σ_y^2, and σ_z^2. The results here are very similar to Case #1. The variance peaks at about 50 s into the simulation and then rapidly decays. This is not too surprising as the EKF tuning has changed from

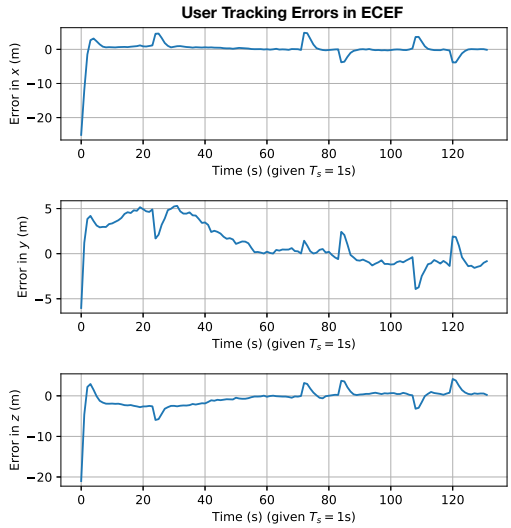

Fig. 11: *ECEF errors in position estimation for Case #1.*

Case #1, with the exception of the initial position error. Since the simulation only runs for 2.2 min which is 132 s, we have to compare the variances at this time to the Case #2 end results. They appear to be about the same, once again the EKF appears to be working correctly.

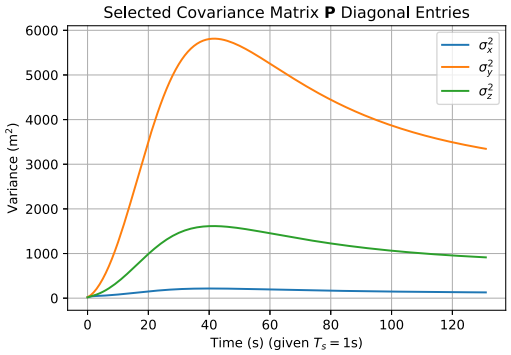

Fig. 12: *Selected error covariance matrix terms, in particular the diagonal elements* σ_x^2, σ_y^2, σ_z^2.

Finally, Figure 13 plots the ENU trajectory estimate in the plane EN (ignoring the UP coordinate as before). The speed is upped by a factor six compared to case #1. The most notable change is trajectory overshoot at each of the right-angle turns. No surprise here as the EKF is asked to handle very abrupt (and impractical) position changes. The EKF recovers quickly.

Overall the results for both cases are very good. There a lot of *knobs* to turn in this framework, so many options to explore.

It is worthy of note at this point that the *Unscented Kalman Filter* (UKF) [Wan2006], and the more general class of algorithms known as *Sigma-Point Kalman Filters* (SPKF), are today much preferred to the EKF of the past. The EKF is sub-optimal, and the linearization approach makes it sensitive to initial conditions. The EKF requires the Jacobian matrix, which may be hard to obtain, and may not converge without carefully chosen initial conditions. In this paper the EKF was chosen for use in a training scenario because it is the next logical step from the linear Kalman filter, and its development is simple to follow. The UKF is harder to get explain. In the end, the UKF is of similar complexity to the EKF,

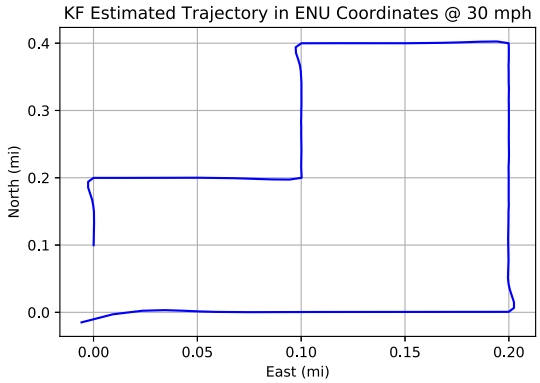

Fig. 13: *The estimated user trajectory in ENU coordinates and the same scale as Figure 7.*

can offer large performance benefits, and does not require the use of a Jacobian.

Conclusions and Future Work

The objective of creating a Jupyter notebook-based simulation tool for studying the use of the EKF in GPS position estimation has been met. There are many tuning options to explore, which provides a very nice environment for studying a large variety scenarios. The performance results are consistent with expectations.

There are several improvements under consideration. The first is to develop a more realistic user trajectory generator. The second is to make measurement quality a function of the SV range, which would also make the measurement quality SV specific, rather than identical as it is now. A third desire is to move to the UKF to avoid the use of the Jacobian, reduce the sensitivity to initial conditions, and improve performance.

References

[Celestrak] *CelesTrack*, (2017, January 26). Retrieved June 26, 2018, from https://celestrak.com.

[SGP4] *Python implementation of most recent SGP4 satellite tracking*, (2018, May 24). Retrieved June 26, 2018, from https://github.com/brandon-rhodes/python-sgp4.

[GPS_helper] *Tools and Examples for GPS*, (2018, June 24), Retrieved from https://github.com/chiranthsiddappa/gps_helper.

[GPS] *Global Positioning System*, (2018, June 24). Retrieved June 26, 2018, from https://en.wikipedia.org/wiki/Global_Positioning_System.

[Garmin] *GPSMAP® 60CSx with sensors and maps owner's manual*, (2007), Retrieved June 26, 2018, from https://static.garmincdn.com/pumac/GPSMAP60CSx_OwnersManual.pdf.

[Kalman] Kalman, R. (1960). A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 35–45.

[Brown2012] Brown, R. and Hwang, P. (2012). *Introduction to Random Signals and Applied Kalman Filtering with MATLAB Exercises*, 4th edition. New York: Wiley.

[Kaplan] Kaplan, E. and Hegarty, C., editors (2017). *Understanding GPS/GNSS: Principles and Applications*, third edition. Boston: Artech House.

[Kim2011] Phil Kim, P. (2011). *Kalman Filtering for Beginners with MATLAB Examples*. CreateSpace Independent Publishing Platform.

[Simon2006] Simon, D. (2006). *Optimal State Estimation*. New York: Wiley-Interscience.

[Wan2006] Wan, E. (2006). Sigma-Point Filters: An Overview with Applications to Integrated Navigation and Vision Assisted Control. *IEEE Nonlinear Statistical Signal Processing Workshop*. doi:10.1109/NSSPW.2006.4378854.