UNIVERSITY OF PADOVA

Satellite Communication Systems
Report: PVT computation block

Figaro Mattia - 2089060
Francesco Rossato - 2082121
Sara Levorin - 2096409
Fattokh - 2085453
Simone Piccirilli

Academic Year 2022-2023

Contents

0.1 Contributions

This table lists all the contributions to the project for each group member.

Author	**Contribution**
Francesco Rossato	• MATLAB code GNSS and LNSS: completion and improvement, results, commenting and code description. • MATLAB code kalman filter: code writing, commenting and testing. • Report writing.
Mattia Figaro	• MATLAB code GNSS and LNSS: completion and improvement, results, commenting and code description. • MATLAB code kalman filter: commenting and testing. • Report writing.
Sara Levorin	• Report writing
Fattokh	• MATLAB code GNSS and LNSS: initial draft, commenting and improvements
Simone Piccirilli	

Chapter 1

Project overview

1.1 The general view

The high level goal of the project carried out by the whole class of the Satellite Communication Systems course is to develop a MATLAB simulator for a system analogue to the modern days' GPS but working on the Moon rather than on the Earth.
From the determination of the orbital parameters down to the user movements on the Moon, the simulator should be able to compute the position of the GPS receiver while simulating the whole scenario.
The broad challenge was divided into four working groups:

- **Orbital propagator team**: defining the Keplerian parameters of the satellites' orbits, the design of the constellation and the number of satellites to ensure a good coverage and provide the data to the other groups.
- **Satellite transmitter**: implement the behaviour of the transmitter on the satellite, define the navigation message structure, develop a channel model and implement some modulation schemes. Simulating the transmission of the navigation message through the channel and delivering the resulting signal to the *user receiver* group.
- **User receiver**: implement the satellite tracking on the receiver side, decoding the signal received from the *satellite transmitter* group in order to recover the navigation message. From the navigation message, compute and provide pseudoranges, pseudorange rates and navigation message to the *PVT block design* team.
- **PVT block design**: receiving the pseudoranges, the pseudorange rates and the navigation messages from the *user receiver* group and compute the PVT (Position, Velocity and Time) of the receiver starting from the received pseudorange measurements and navigation messages.

1.2 PVT block design

Our task consists in designing the block capable of estimating the PVT of the receiver, that is, its Position, Velocity and Time. In order to do this, we should receive pseudoranges, pseudorange rates and the navigation messages of the satellites from the *user receiver* team. Moreover, in order to target more realistic scenarios we implemented some possible integrations to the PVT computational block. Those include the possibility of equipping the user exploring the Moon with an inertial measurement unit *IMU* in order to exploit the inertial information provided, i.e. acceleration, to feed a Kalman filter. The other alternative implementation consists in equipping the receiver with an altimeter which determines the altitude through sensing air pressure. For more details please refer to next sections.

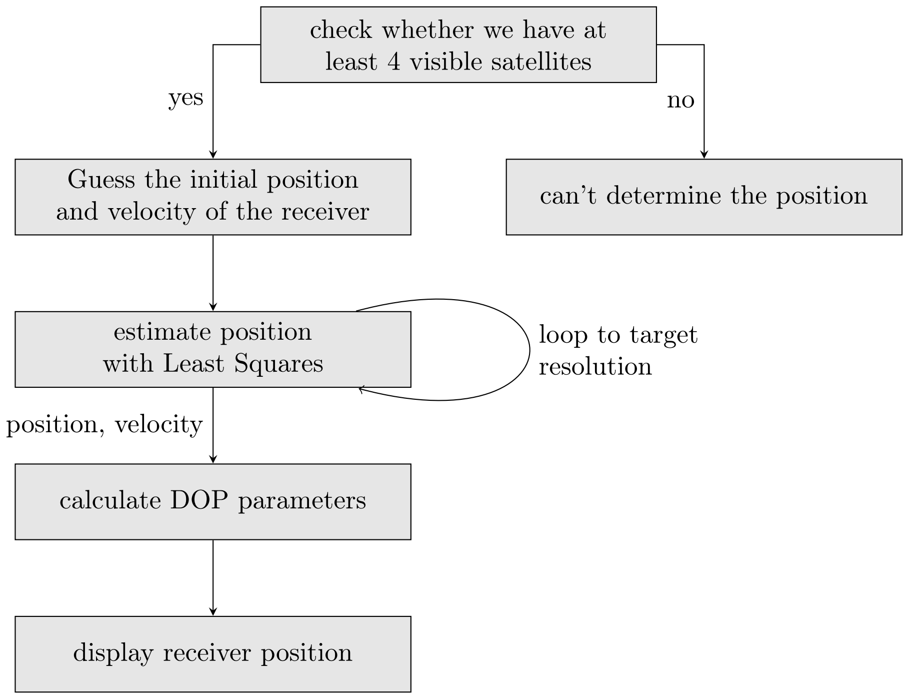
check whether we have at least 4 visible satellites
yes
no
Guess the initial position and velocity of the receiver
can't determine the position
estimate position with Least Squares
loop to target resolution
position, velocity
calculate DOP parameters
display receiver position

Figure 1.1: PVT block diagram

Block diagram The behaviour of our block can be easily visualized in the block diagram 1.1. First of all, we check whether we have at least 4 visible satellites by examining the data provided by the *user receiver* group. Then, if the condition holds true, we make a guess of the user position in order to have a faster convergence of the least square algorythm (it does not matter if the guess is rough, picking a random point on the Moon surface is still better than starting from `[0,0,0]`), iterate the least square until we either are able to get to the desired precision or we arrive to a threshold number of iterations. This provides the position and velocity estimate. Finally, we compute the DOP parameters and we are now able to display the position of the receiver.
A similar procedure is used when dealing with the addition of a reference ground station and an altimeter placed at the receiver.

File structure Our work is divided into two main folders:

- **LNSS**: it contains several files, but the the two main ones are "GNSS_simulation.m" and "LNSS_simulation.m". The first one simulates a GPS receiver on the Earth. We have used the last version of the file "data_Rx_group.csv" we received from the Receiver Gruop and a real GPS data file that we have downloaded on the Internet because we were not provided with the navigation message, which is necessary to compute the orbits. The purpose of this script was to test our implementation in an easier environment like the Earth, before adapting it to the Moon. It provides several results like satellite positions, position of the receiver, position (ECEF) error for each axis, accelerometer readings,position (ECEF) error, dilution of precision (DOP), number of visible satellites and satellite visibility.
  The second main file simulates a GPS receiver on the Moon. We have used the file "data_Orbit_group.csv" we received from the orbital group because we were not provided with the navigation message, which is necessary to compute the orbits. Furthermore, we have discovered that the satellite constellation is designed in such a way that we almost never have four or more visible satellites. For this reason we are not able to provide a consistent position fix for each instant. We are not able to provide the same output plots as for GNSS_simulation.m because we would obtain meaning-

less results since for the most part of the time we are not able to see at least four satellites. The code structure is very similar to GNSS_simulation.m but it is adapted for the Moon environment and different orbital parameters. It provides several plots like position of the receiver, satellite orbits, accelerometer readings along X, Y, Z axis, number of visible satellites and satellite visibility.

- **Extended Kalman Filter**: also here there are two main files: "kalman_filter_main.m" and "kalman_filter_test.m". The first one is a function that uses the Kalman filter for solving the GPS positioning problem. It makes use of the Extended_KF and least square functions, and its input parameters are the pseudoranges and the satellites positions. As output it provides the comparison between the error of the Kalman Filter and the Least Square when used to compute the position.
  The second one performs some sanity checks on kalman_filter_main to ensure that the code is working properly. As output it provides the computed position along X, Y and Z directions, the computed velocity along X, Y and Z directions and the computed clock bias.

Dependencies In order to be able to run all of these codes you need to have the following extensions:

1. aerospace_toolbox
2. navigation_toolbox
3. communication_toolbox
4. signal_blocks
5. signal_toolbox

Chapter 2

Code description

2.1 GNSS implementation on the Earth

Why Earth One of the main challenges that we faced during the development of the Moon GPS project was the lack of an available test bed with some real data, since a positioning system akin to the terrestrial GPS does not exist on the Moon yet. We therefore had to design a way to test the correctness of the results of our implementation. The idea we came up with consisted in starting by developing a receiver placed on the Earth, so that we were able to gather some real-world data to test out software. Then, once we were confident with the provided results, we proceeded to adapt the code for the Moon scenario.

2.1.1 Parameters

In order for the simulation to be configurable as desired, we defined a set of parameters:

- `start_time` initial time of the simulation
- `mask_angle` specify the mask angle to decide the satellites which should be excluded from the position computation
- `dt` time interval between subsequent simulation instants
- `target_res` the accuracy of the position that we aim to achieve
- `max_iter` the maximum number of iterations for the least square algorithm

2.1.2 Input files

We begin by reading the data provided from the *orbital propagator* team in the file `data_Rx_group.csv` and the *user receiver* team in the file `data_Orbit_group.csv`, since, even though this script is built to work on the Earth, we still want the majority of the parameters to be similar in both scenarios.
However, we soon discovered some problems with the data: we are not provided with any navigation message and the contents of the file are not sufficient to determine the user position.
We therefore decided to move to a more realistic scenario by considering a real world GPS RINEX file `real_gps_data.rnx` containing the navigation message with all the data we need to compute the simulation.

2.1.3 Code implementation

Since we were lacking the pseudorange information, we needed to calculate them based on the data contained into the RINEX file. This calculation was performed twice: once for the standard scenario, then considering the ground station for differential positioning. For each simulation step, we performed the least square method until we reached the desired precision or until we reached a specified number of iterations which we set to 200. During each iteration, the position estimate is updated using equation 2.1 and the whole process restart.

$$\Delta x = (H^T H^{-1})^{-1} H^T \Delta\rho \tag{2.1}$$

```
% we begin the computation of the position: initially, the
    resolution
% is infinite (max uncertainty) and we want to achieve the
    resolution
% specified in the variable target_res with the successive
    iterations.
res_pos = Inf;
res_pos_diff = Inf;
target_res = 1;
max_iter = 200;
iter = 0;
  [...]
while (res_pos > target_res) && (iter < max_iter)
    [...]
```

We then compute the pseudoranges and the so called line of sight vectors, which point from the receiver to the direction of the satellites.

For the differential GNSS computation, we place a ground station at a fixed position on the Earth near the user and perform similar computations as the ones described above. The main difference is that here we need to factor into account the measurements that we get for the station as follows:

```
while (res_pos_diff > target_res) && (iter < max_iter)
    % dgnss position accounting for the ground station
    [pEst_dgnss, ~,~] = nav.internal.gnss.calculatePseudoranges
        (single_sat_pos(is_visible,:), single_sat_vel(is_visible
        ,:), station_position, [0,0,0]);
    delta_rho = pEst_dgnss-pseudo_sat_dgnss(is_visible,idx);
    % pseudoranges with previous estimated position
    [p_est_diff, ~, los_vector_diff] = nav.internal.gnss.
        calculatePseudoranges(single_sat_pos(is_visible,:),
        single_sat_vel(is_visible,:), pos_prev_diff(1:3).',
        [0,0,0]);
    % new orbit estimate adding Least Square deviation (lecture
         21, slide 27, point 3)
    pPred_1 = p_est_diff + delta_rho;
    H_pos_diff(:,1:3) = -los_vector_diff;
      [...]
```

Functions Our code makes use of a MATLAB function available in the navigation toolbox called `calculatePseudoranges` which returns the pseudoranges and the line of signt vectors pointing from the receiver to the satellites.
Also, we have designed the function
`function [hdop, vdop, tdop] = calculate_DOP(dop_matrix, refFrame, lla_0)`
capable of computing the DOP parameters with the equations as follows:

$$GDOP = \sqrt{D_{11} + D_{22} + D_{33} + D_{44}} \qquad PDOP = \sqrt{D_{11} + D_{22} + D_{33}}$$
$$HDOP = \sqrt{D_{11} + D_{22}} \qquad VDOP = \sqrt{D_{33}}$$
$$TDOP = \sqrt{D_{44}}/c$$

2.2 LNSS implementation on the Moon

After testing our ideas and implementations on a easier environment like the Earth, we have decided to pass to our target aim: a GNSS system on the Moon. As we will see, the main structure remains quite the same, as we have seen that GNSS code worked as intended, but with some adaptations for the Moon system. Doing our job, however, we ran into some difficulties that did not allow us to achieve the desired results. This was due to some shortcomings in the input data from the orbit and the *receiver group*. Anyway, we will go into more specific details in next sections.

2.2.1 Parameters

In order for the simulation to be configurable as desired, we defined the same set of parameters as we did for GNSS code. For example parameters like `start_time`, `max_iter` and `target_res` remained untouched. On the other hand, some parameters have changed: the `mask_angle`, since on the Moon we assume to have less obstacles than down on the Earth so it had a smaller value than GNSS one, and `dt`, since we have set it equals to 300 sec that was the data sample period used by *orbit group*.
One last very important parameter was the `num_sats`, because in this situation we had not available 32 satellites as in GNSS case, but only four. This will be the biggest issue we faced, and one that will undermine many of the proposed results.

2.2.2 Input files

In this case, the only input file used was `data_Orbit_group.csv`, that contains, for each satellite, the corresponding position in x, y, z axis, its distance with respect to the receiver, its pseudorange values and a boolean variable corresponding to whether the satellite was visible or not at every timestamp instance. There were also the receiver position in ECEF coordinates, the mask angle, the eccentricity and the satellite antenna cone angle.

2.2.3 Code implementation

As we have already said, the main structure of the code is practically identical to the one proposed in GNSS one. For that reason in the following section we will analyze only the main features.
Since we have not received any consistent files from the *receiver group*, we had to used the only data that we had available, namely those contained in the input file. Starting from that, we computed some important parameters that we missed, like the distance between the receiver positions in two successive time instants and the velocities along x, y, z for each one of the four satellites.
For the first ones, we used a very simple cycle like the following:

```matlab
% calculate the distance between the receiver positions in two
% successive time instants
dist_receiver_x = zeros(steps-1,1);
dist_receiver_y=  zeros(steps-1,1);
dist_receiver_z = zeros(steps-1,1);
for ii = 1:steps-1
    dist_receiver_x(ii)= receiver_ECEF(ii+1,1) - receiver_ECEF(
       ii,1);
    dist_receiver_y(ii) = receiver_ECEF(ii+1,2) - receiver_ECEF
        (ii,2);
    dist_receiver_z(ii) = receiver_ECEF(ii+1,3) - receiver_ECEF
        (ii,3);
end
distance = [dist_receiver_x, dist_receiver_y, dist_receiver_z];
```

Also a similar ideas for the velocities can be used, where, for simplicity, we will report the code for only one satellite, but the actual implementation includes the same computation for each of the four ones within the for loop.

```
% calculating the velocities along x, y, z for each one of the
   four
% satellites. Since we do not have the navigation message, we
   need to do
% this by hand by looking at their positions in time
for idx = 1:steps-1
    sat1_difference = [initial_data(idx+1,3) - initial_data(idx
        ,3), initial_data(idx+1,7) - initial_data(idx,7),
        initial_data(idx+1,11) - initial_data(idx,11)];
    sat1_vel = sat1_difference/dt;
    sat_velocities(:, :, idx) = [sat1_vel];
end
```

The last new feature that we had added, in order to have consistent results, was a check to verify that, at the current time instance, we were working with at least four satellite, otherwise we would obtain meaningless outputs. It was collocated as one of the first starting commands inside the main for loop: the one that contains the two while loop for the standard and altimeter computation of PVT.

```
for idx = 1:steps
    % copy pseudorange at step idx
    pseudoranges_idx = all_pseudoranges(:,idx);
    % take only the current values at instant idx
    pdot = all_pseudorange_rates(:,idx);
    is_visible = all_are_visible(:,idx);
    single_sat_pos = sat_positions(:,:,idx);
    single_sat_vel = sat_velocities(:,:,idx);
    % we had to put a condition on the number of visible
        satellites,
    % because otherwise the code throws many warnings of "
        matrix is singular"
    % since, because of the way the satellite constellation is
        designed, we almost
    % never have 4 or more visible satellites, which is the
        minimum number to be
    % able to perform the position fix. For that reason we take
         in consideration
    % data only when we can see at least four satellites.
    if sum(is_visible(:)) >= 4
```

Functions In `LNSS_simulation.m` we used two extra functions that can be found as last things at the end of the code. The first one was `calculateVel` that calculates the instantaneous velocity starting from a vector of positions and time interval between positions and `calculate_DOP` that computes dilution of precision (DOP) values in local frame from Earth-Center-Earth-Fixed (ECEF) cofactor matrix.

2.3 Kalman Filter

In this section the reader can find the detailed explanation of how we chose to implement the Kalman filter code.

2.3.1 Functions

- `kalman_filter_main.m`: this is the main function. It uses the extended Kalman filtering to solve the GPS positioning problem taking as inputs the pseudoranges and the satellite positions. Since we didn't have these data, we resorted to some real world data example as we did for the GNSS and LNSS simulation scripts.
- `kalman_filter_test.m`: this function has the same behaviour as the main one, but it serves a test purpose. The data that we pass as inputs are toy examples and manually calculated so we are able to reliably check the results.
- `Extended_KF.m`: this function contains the actual implementation of a generic extended Kalman filter. The difference between extended and standard KF is that the extended one is able to deal with nonlinear functions by linearizing before applying the standard KF. We chose to separate the generic implementation from the GPS one in order to have better code readability and so that modifications could be made without the risk of having two different versions instead of one.
- `Rcv_Pos_Compute.m`: this function computes the position by usung the least square approximation instead of the Kalman filter in order to compare the results.
- `PseudorangeEquation.m`: simple function to compute the pseudoranges as from Equation 2.2.
$$\rho_j = \sqrt{(x_j - x_u)^2 + (y_j - y_u)^2 + (z_j - z_u)^2} + ct_u \tag{2.2}$$
- `ConstantVelocity.m`: this function computes the matrix expressing how the state evolves in time. We assumed a model similar to the one studied during the lectures with constant velocity.

2.3.2 Code implementation

The general procedure that our KF algorithm implements is the following:

1. Load the inputs: satellite positions and pseudoranges and set the Gaussian noise parameters. The two files are `sat_pos.mat` and `pseudoranges.mat`.
2. Set the simulation parameters: the time interval between readings and the number of iterations to be performed per reading
3. Guess a random position and clock bias for the user, since this is better than having to start from all zero values.
4. Initialize the prediction matrix that will contain the information about the prediction step.
5. Repeatedly iterate the computation of the position with the Kalman filter and with the least square in order to compare the two solutions.
   The extended Kalman filter (`Extended_KF.m`) is used to deal with the non linearity, and it proceeds as following:
   - linearize the two input functions $f(x)$ of the state transition and $g(x)$ of the measurement.

    - compute the state transition matrix and the observation matrix
    - apply an ordinary (non extended) Kalman filter

6. Save and display the results.

Note that the initial guessing of the position is pretty accurate, indeed. This is not a realistic scenario, since any GPS would be useless if it required the knowledge of the user position to operate properly. However, since we are interested in plotting the evolution of the position error as the iterations go by, setting a random, potentially really coarse initial guess would have caused a much higher initial error spike such that the following data points would appear as a straight line with no appreciable variations unless we started zooming in, so ultimately this was set for ease of reading of the output plots. It can, however, be changed to an arbitrary value to show that the algorithm works properly even then.

```
% initial position guessing
X([1 3 5]) = [-2.168816181271560e+006, 4.386648549091666e+006,
    4.077161596428751e+006];
```

2.3.3 Test implementation

In order to verify whether we were obtaining some meaningful results, effort was put in designing a testing script for out Kalman filter implementation.

Satellite scenario The scenario consists of a receiver standing still in position `[0,0,0]` and four satellites placed along $\hat{x}$ and $\hat{y}$ axis as shown in Fig. 2.1, so that the pseudoranges are easy to compute.

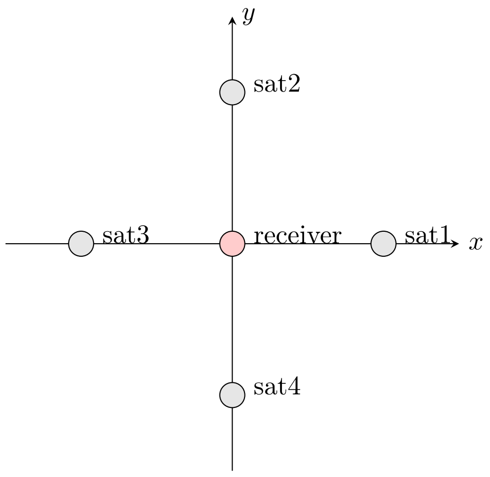

Figure 2.1: Satellite and receiver positioning

Being a static scenario, the pseudoranges are constant as well as the satellites. We are aware that this is not physically possible, but this is a test scenario created ad-hoc. The insertion of the data regarding the satellite positions and pseudoranges is performed by the following code snippet, where:

- `dist` is the parameter containing the distance from the receiver to the satellite, which we suppose to be the same for each of our satellites.
- `iterations` is the number of iterations that the Kalman filter and least square should perform
- `noise` is the random noise that we add to the pseudoranges

- `bias` is the error on the pseudoranges caused by the clock bias.

```
for i = 1:iterations
    sat_pos(i) = {[0, 0, dist; 0, dist, 0;-dist, 0, 0; 0, -dist
        , 0]};
    pseudoranges(i) = {[dist+noise(1, i)+bias, dist+noise(2, i)
        +bias, dist+noise(3, i)+bias, dist+noise(4, i)+bias]};
end
```

Noise modeling Real world implementation is always different from ideal conditions, and the measurements are affected by both noise and clock misalignment between the satellites and the receiver. In order to account for this, we added two parameters to the pseudoranges: a zero average Gaussian noise and a bias representing the error on the calculation caused by the clock misalignment. The first test was conducted with both these parameter set to zero, in order to see whether it converged under ideal conditions.
Since it worked correctly, we increased both the noise and the clock bias as shown in the following code snippet.

```
bias = 25;
  [...]
noise = zeros(4, iterations);
noise = wgn(4, iterations, 0.5);
```

We then proceeded by feeding the data into our extended Kalman filter and least squares. The results are described in Section 3.3.

Chapter 3

Results evaluation and Conclusions

3.1 GNSS

Satellite positions Basing on the satellite positions provided in the navigation message and the position that we computed using the pseudoranges, we were able to calculate the direction of the satellites with respect to the user. In particular, we extracted the vectors pointing from the user to each of the visible satellites. The obtained plot is depicted in Fig. 3.1. We set the mask angle to 10°, which is a fairly typical value, to exclude the satellites too close to the horizon. Those satellites can worsen our position estimation because they suffer from more atmospheric noise and fading. Moreover, it is unlikely that we have a perfect 180° unobstructed view of the sky unless we were for example in the middle of a big field. Setting a mask angle makes for a more realistic scenario. The mask angle is represented on the plot as a gray band placed at the margin of the circle.

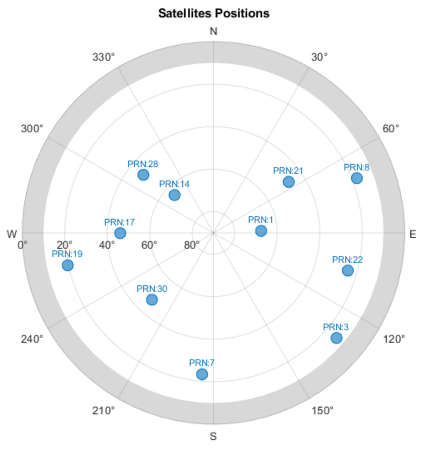

Figure 3.1: Satellites with respect to the receiver.

Receiver position The main focus was to be able to estimate the receiver position. This is represented in the plot of Fig. 3.2, where the trajectory of the user is depicted.

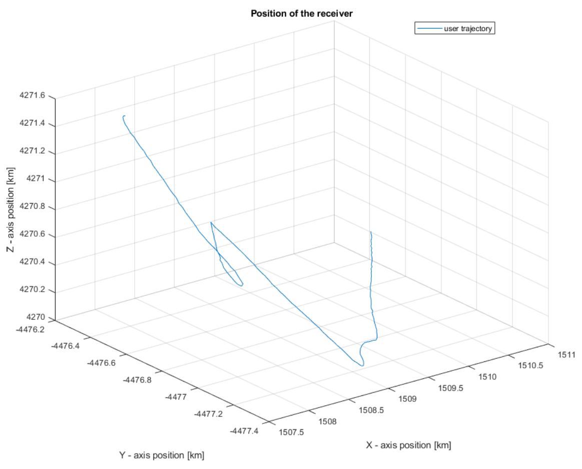

Figure 3.2: Receiver trajectory on Earth.

Position error with altimeter Since the altimeter is modeled as a random variable, we can't derive any meaningful consideration regarding how the measurement improves. This is certainly a point of discussion for future improvements.
This error is calculated as the difference between the user real position, assumed to be known for the sake of this simulation) and the computed position with our GPS system. The error in ECEF coordinates for the measurements with the altimeter and the measurement without it is depicted in Fig. 3.3. As we can see, the inclusion of the altimeter data does not lead to any meaningful improvement

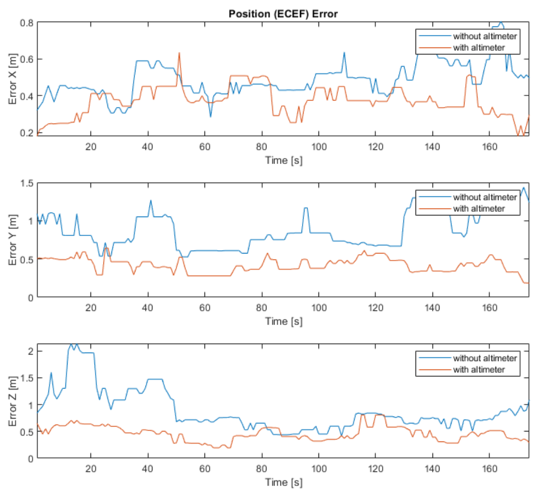

Figure 3.3: Receiver ECEF error with and without altimeter.

ECEF error, no altimeter In Fig. 3.4 we can see the error along each of the three axis as time evolves. Note that the values are overall small and compatible with a standard GPS receiver, while exhibiting a bigger peak about 40 seconds into the simulation.
Cross-checking with the user trajectory, this time instant correspond to the abrupt change in

direction of the user that happens about in the middle of the long straight segment on the left in Figure 3.2. This behaviour happens because in our model we assume that the receiver follows a simple trajectory with a slowly varying velocity, and the data is received from the satellites at some predetermined intervals, hence some time passes between successive position estimations.
As a consequence we have that a sudden change in direction and velocity leads to a spike in the position error, which fades away once the system catches up with the changed velocity and direction, updating its status. We can see this in action because, given a few seconds, the error returns back to lower values.

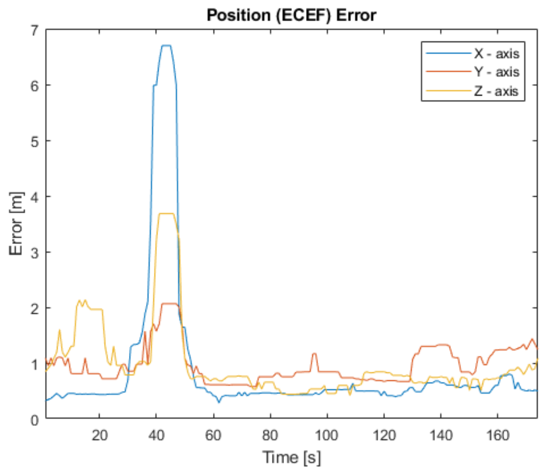

Figure 3.4: Receiver ECEF error.

DOP parameters Figure 3.5 shows the dilution of precision parameters, both HDOP and VDOP and both when considering and when not considering the altimeter readings. Since the altimeter is modeled as a zero-average random variable with a small variance, we see that the two HDOP curves are almost perfectly superimposed, and the same occurs for the two VDOP curves. Increasing the variance tends to spread the curves apart. However, in order to see an improvement we would need to model the altimeter using real data.

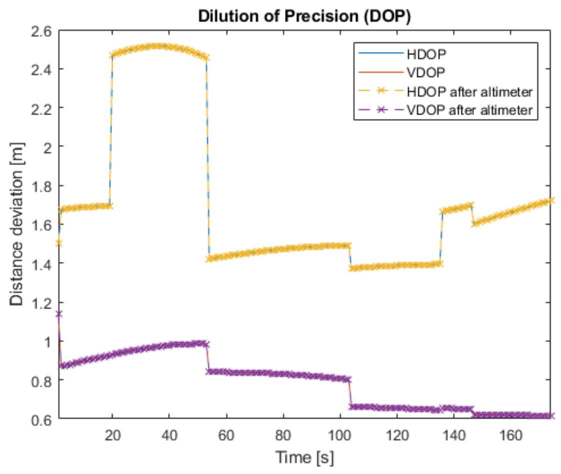

Figure 3.5: HDOP and VDOP with and without altimeter.

Satellite visibility Since the satellites move with time, their visibility changes as time goes by. This behaviour is depicted in Figures 3.6 and 3.7 where we can see the instants when each one of the 32 satellites is visible and the cumulative number of visible satellites. An important observation to be made is that we can always see at least 4 satellites (as a matter of fact, we never dip below the figure of 9 visible satellites), so we are always able to calculate the receiver position.

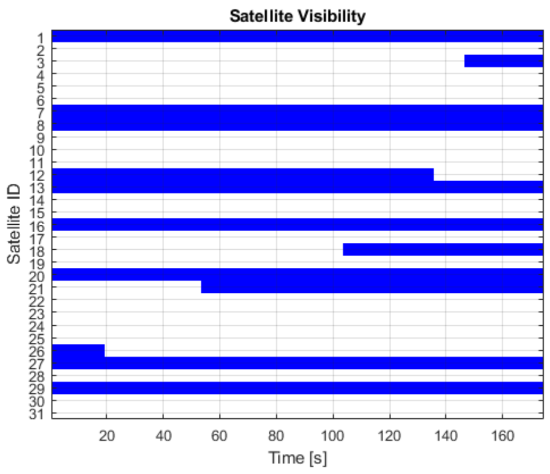

Figure 3.6: Visibility for each satellite.

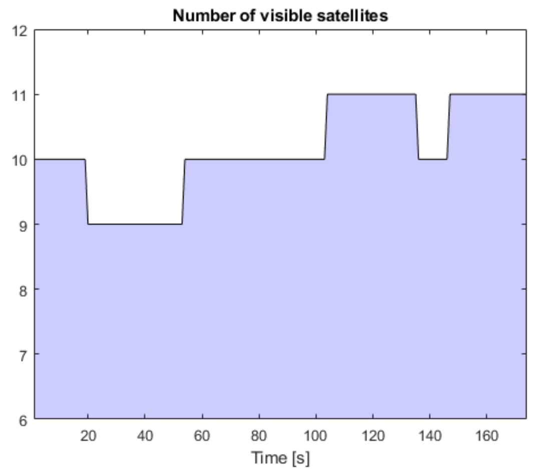

Figure 3.7: Number of visible satellites.

3.2 LNSS

Receiver Position In Fig. (3.8) is possible to observe the receiver position in every consistent time instant, in other words, only in the instant in which we can view at least four satellite. If we compare this plot with what we have obtained for GNSS code, we can easily say that this result is much more inaccurate. The number of plotted point is very low, furthermore often the points are thickened in small range of the trajectory. This can

be explained simply observing the result of Fig. (3.10) or (3.11), in which is shown that we have all the satellites in view only for short and distant intervals.
Instead, the outlier point, at the bottom of the graph, is only an initial guess for the user position.

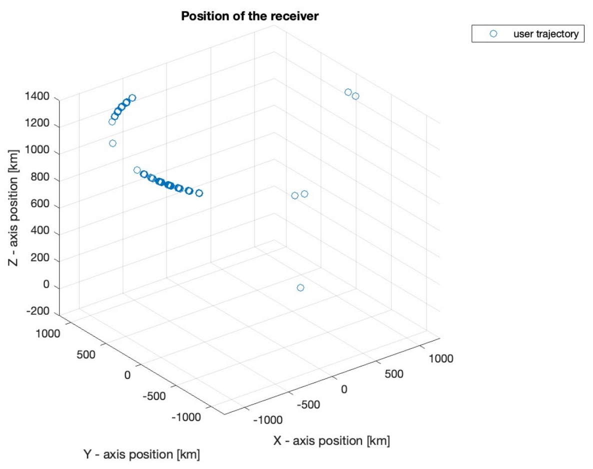

Figure 3.8: Position of the receiver.

Satellites orbits This is only a didactic plot to show orbits of each satellite around the moon, however, it is not possible to clearly appreciate when the satellites are visible from the receiver; the next figures will be needed for this.

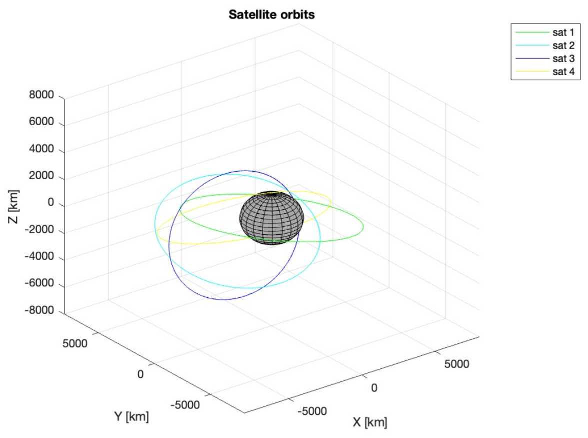

Figure 3.9: Satellite orbits.

Satellites visibility Watching the Fig. (3.10) is possible to notice that we almost never have four satellite in view. It happens only after time 2200 sec, and occurs only every 100 sec, between these intervals we are completely blind.

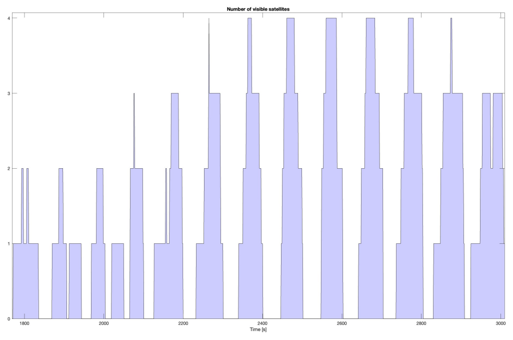

Figure 3.10: Number of visible satellites.

Here instead, is possible to see when each satellite, with its own ID, is visible at each time.

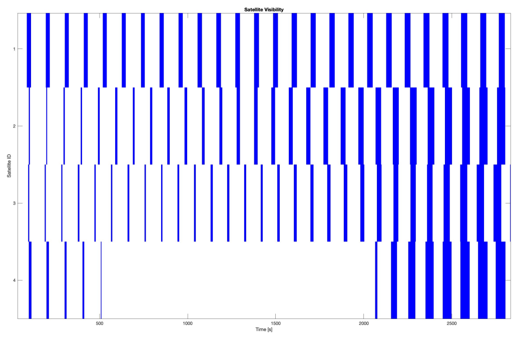

Figure 3.11: Satellite visibility.

At this point, it is possible to make clear why it was not possible to do an accurate job as in GNSS case. We know that for correctly computing a PVT operation we need at most four observations, corresponding to the four satellites in view. In a more general case, we need a number of equations equals to $dim + 1$, where dim is the number of dimensions in which we work; so in our case, in which we deal with a 3-D space, this means that we need four equations.
All this talk because we should compute the position of the receiver, so there are the three obvious unknown, its x, y and z coordinates, but also a fourth one, that is the misalignment between reference system and receiver clock, since the one between each satellite and receiver should be know by the navigation message. But, as we already said, we are working with data coming from four satellite, which are hardly ever visible at the same time, and also we do not have the navigation message, since *receiver group* did not provide it to us.
For these reasons, we had to work only with the data from the *orbit group*, even if, in principle, we should not have had all the perfect position data of each satellite.
Futhermore, we are not able to provide the same output plots as for `GNSS_simulation.m`, where we resorted to use a real world GPS file, because we would obtain meaningless results since for the most part of the time we not able to see at least four satellites.

3.3 Kalman

3.3.1 Testing implementation

The test implementation of the Kalman filter presents the following behaviour:

Receiver position We initially set the receiver position to be at the center of our system and calculated the pseudoranges based on this assumption. We therefore know the exact values of the coordinates the Kalman filter and the least squares algorithms should converge to (Figure 3.12).

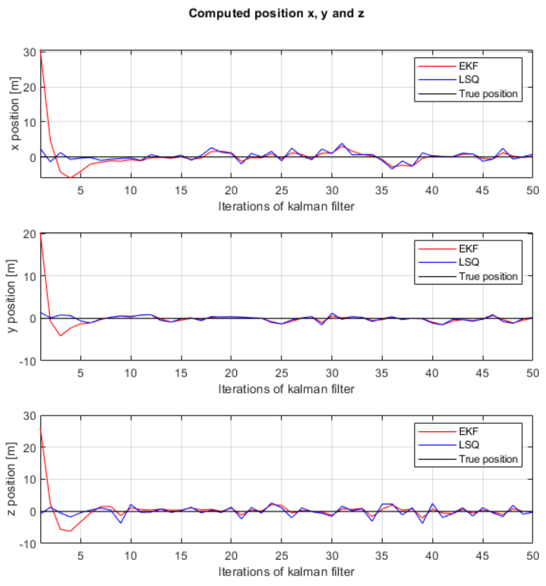

Figure 3.12: Testing receiver position.

As we can see, after the initial spike due to the first iterations being poorly accurate, all the three coordinates converge to zero, which is the true position of the receiver, exactly what we wanted.
In terms of performances, not much difference is observed between KF and LSQ algorithms. This happens because the Kalman filter is often used to combine measurements from different sources such as accelerometers, rotational sensors placed on the wheels in the case of a travelling car, ... where in this case our implementation was designed to only compute the position starting from the pseudorange measurements. Tha Kalman filter would have been more helpful if we had access to a proper inertial measurement unit.

Increasing the noise by ten times, we get that the error increases as shown in Figure 3.13. Once again, we see that both algorithms tend to converge to the correct position, albeit with a bigger swing.

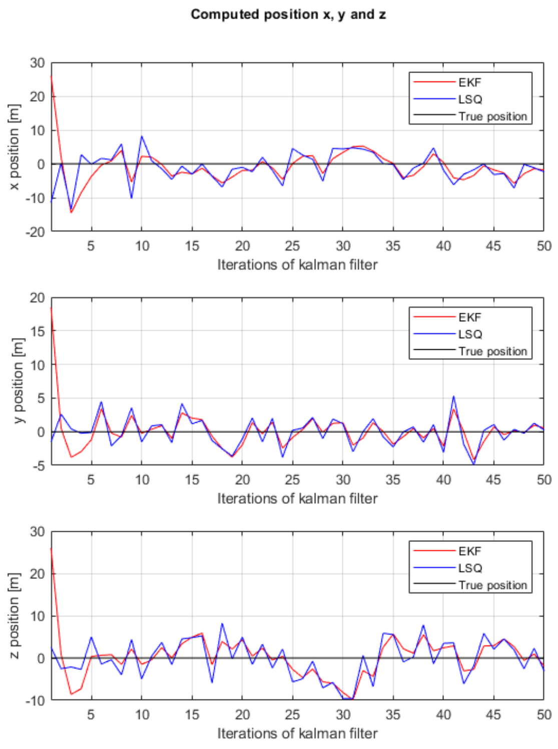

Figure 3.13: Testing receiver position with increased noise.

Receiver velocity We initially set the receiver velocity to be zero, so the Kalman filter should converge to zero in all directions, even though the pseudoranges are varying due to the noise that we add to them. In Figure 3.14 we can see this exact behaviour: the velocities are converging to zero, which is what we expected.

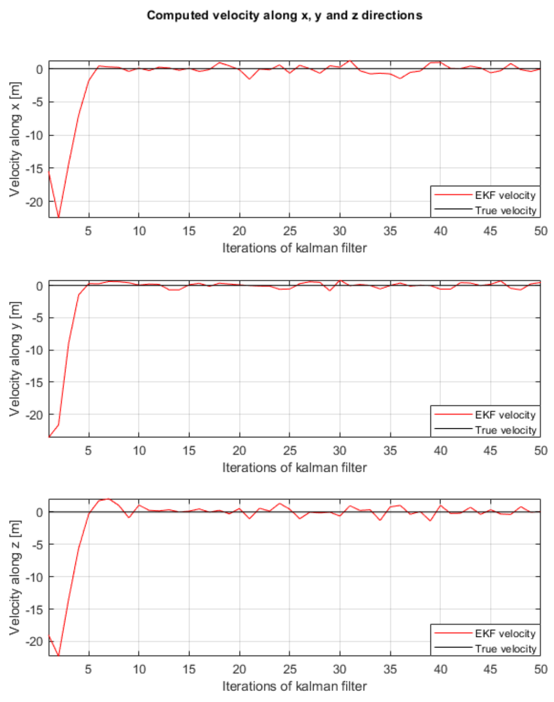

Figure 3.14: Testing receiver velocity.

Clock bias We initially set the error on the pseudorange measurements due to the clock bias to be of 25 meters. This constant component is picked up by the Kalman filter as shown in Figure 3.15. The black line represents the correct value that we manually set.

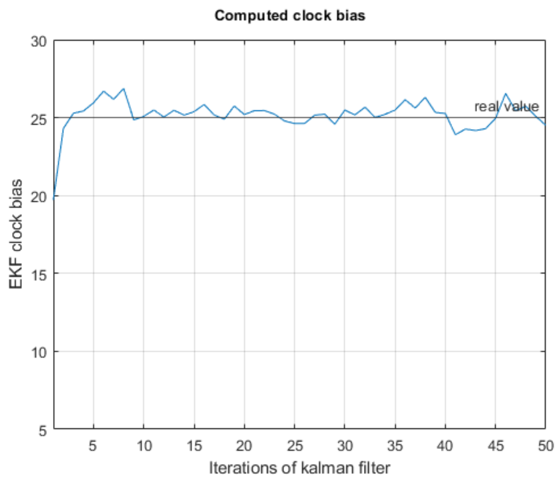

Figure 3.15: Testing receiver clock bias.

3.3.2 Real implementation

After the testing phase, we loaded a set of real world measurements and ran both the Kalman filter algorithm and the least squares algorithm. The results can be seen in Figure 3.16. The

errors are worse than the ones observed in our test scenario, but this was expected since we are now dealing with real orbiting satellites and a moving receiver with non constant velocity, which is different from our simple model where velocity was assumed constant.
Note that the least square algorithm performs noticeably worse than the Kalman filter in the $\hat{z}$ position. this could be due to the fact that the user is moving faster along $\hat{z}$ rather than the other two directions, and the LSQ needed more iterations to converge to a better estimate. Nonetheless, both systems are working and the errors in the last iterations are under a reasonable margin.

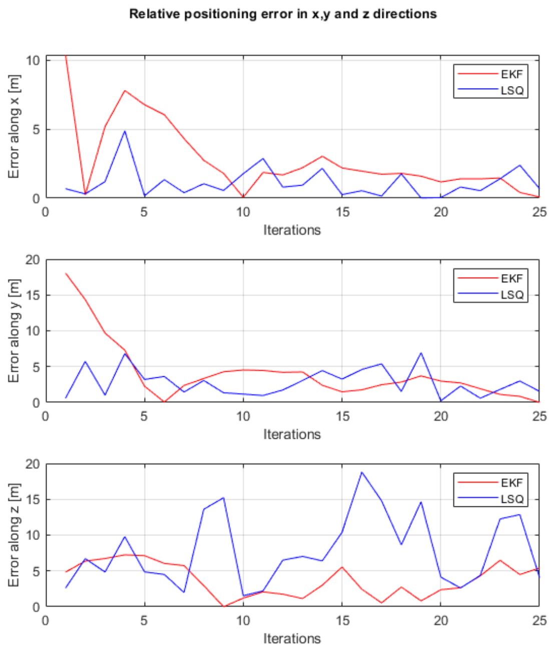

Figure 3.16: Receiver position error with KF and LSQ

3.3.3 Performance

The code is quite slow in terms of execution: during our tests running on an Intel Core i7 6700K CPU at 4GHz the execution took 6 seconds, dominated by the loops for the calculation of position. It also uses many variables, increasing its memory needs. It is most definitely not suitable for position computation on a mobile device. This can surely be improved by decreasing the number of parameters that we are calculating: many variables and plots are computed and displayed only to check the validity of the results and for the sake of getting a better understanding about how the GPS system works.
Looking at it from an end-user standpoint, all of these parameters are of no use and resource-wasteful.
Surely, more thought can be put on optimization, by either getting rid of the unnecessary debug variables, settling for a lower precision, avoid displaying graphical results and even rewriting the code in a programming language tailored for efficiency. Modern days GPS receivers are highly integrated Systems On a Chip (SOCs), where many computations are performed at the hardware level, resulting in unmatched execution speed and energy efficiency.

Chapter 4

Improvements

IMU and Kalman filter This is surely one key aspect of future improvements: the integration of the LNSS simulation code with the Kalman filter and a proper inertial measurement unit like an accelerometer, in order to provide a better position estimation and guarantee that the system will continue to work even in the case of a brief disruption of the satellite signals. IMUs can provide good performances in this scenario in the short term, but tend to accumulate errors over time, so satellites are needed to guarantee the precision of the position in the long run.

Better user experience As for now, the software lacks a polished and well made user interface where one can specify all the parameters and the inputs, see the updated results as the parameters vary and export them, all of this without having to modify the underlying MATLAB code. Being a settings file or a full graphical user interface, this would most certainly benefit the end user experience.

Altimetric data In our code the altimeter is simulated as a random variable. However, it would be interesting to be able to test this with real altimetric data and perform sensor fusion with the Kalman filter. We thought about this and we came up with two possible implementations:

- having a complete map of the Moon with the altitude, but this is probably not feasible since the receiver could also be not placed at ground level. Moreover, we would need to extract the altimetric data from the map and to do so we would need the receiver position, defeating the whole purpose of the GPS system.
- drawing a carefully designed user trajectory and assume that the altitude is known by the receiver. Then add some noise to the measurements and use that as altimeter data. Whilst possible, this would have required a thorough work with the other teams, which has proven to be difficult. It would be a nice future addition.

Integration with other groups data Right now, as we previously discussed, part of our data was taken from real world measurements due to necessity. The downside of this is that the code does not integrate well with the rest of the simulator, so the behaviour of our block is in part agnostic of the other ones. A better integration in particular with the *user receiver* would fix this broken link.

Performance As we have discussed, the code is not particularly efficient and performs checks and redundant calculations that increase both CPU and memory usage. While having some debug data is useful in a testing scenario, a complete revision of the code with a focus on removing the parts that are not strictly necessary and a general optimization would surely improve the execution time, the necessary memory and the overall efficiency also in terms of energy.