COMP0130 Robot Vision and Navigation
Coursework 1: Integrated Navigation for a Robotic Lawnmower

Angelina Magal, Jeffrey Li, Sara Karati
12039874, 20005893, 23196581
angelina.magal.23@ucl.ac.uk, zcabbli@ucl.ac.uk, sara.karati.23@ucl.ac.uk

Contents

1 Context

In this coursework we compute the horizontal position, horizontal velocity and heading of a robotic lawnmower in London. As input, we utilised simulated data from the lawnmower's navigation sensors: a GNSS receiver, wheel speed sensors, a magnetic compass and a low-cost MEMS gyroscope.

2 Methodology

We undertook the following high-level steps to compute the lawnmower's position, velocity and heading (the "solution") from the sensor data:

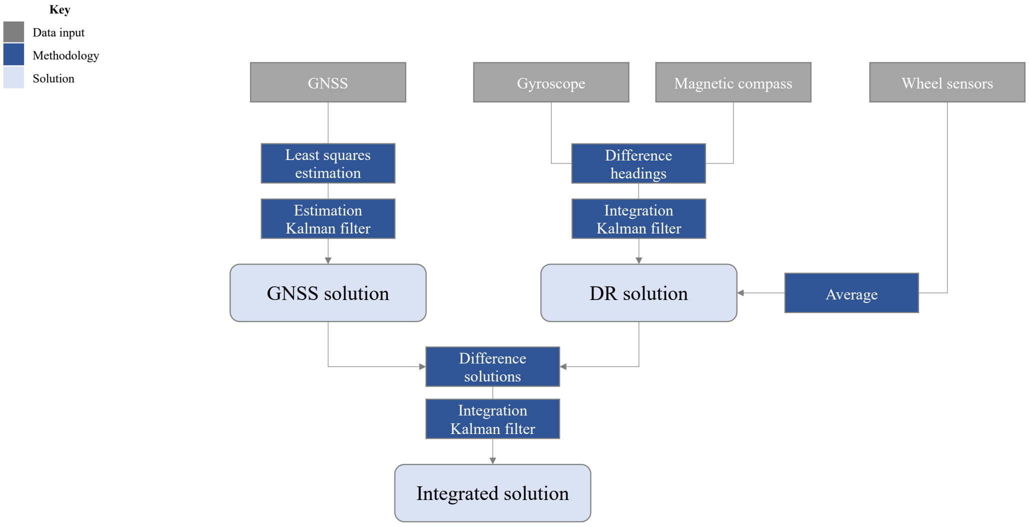


Figure 1: Methodology flowchart

As evident in the flowchart above, we first developed an independent GNSS and dead reckoning (DR) solution. We compared these solutions to identify any obvious errors, before combining all sensor data to compute our final solution.

Given that we are estimating states changing over time and incorporating measurements over time, we used the 10-step Kalman filter algorithm as described in [1] to compute our solutions. We detail the tailoring made to this approach for the specific tasks below. We refer to the standard Kalman filter algorithm steps in [1] throughout these sections as KF Step #.

We used MATLAB to implement our algorithms. We use the mathematical notation used in the COMP0130 lecture materials throughout this coursework. In our code, we use a modified version of Define_constants.m to define the constants used throughout, including most assumptions provided in the coursework instructions (e.g. error specifications). The following error specifications were not incorporated in our solutions: the wheel speed sensor specifications and quantisation, and the gyroscope scale factor, cross-coupling and quantisation. For the GNSS error specifications, we used the "reasonable, though not optimal" performance assumptions.

2.1 Initial positioning and navigation solutions

2.1.1 GNSS solution

We calculated an initial solution based solely on the provided GNSS data using the GNSS_KF.m function.

Initialisation: We initialised the state vector and error covariance matrix using Initialise_GNSS_KF.m. As our initial state vector was unknown, we initialized it using an unweighted least squares approach in line with [2]. The state vector here represented the ECEF position of the lawnmower, $\mathbf{r}^{\mathbf{e}}_{\mathbf{ea}}$ (m), its horizontal velocity, $\mathbf{v}^{\mathbf{e}}_{\mathbf{ea}}$ (m/s), the receiver clock offset, $\delta\rho^{\mathbf{a}}_{\mathbf{c}}$ (s), and the receiver clock drift , $\delta\hat{\dot{\rho}}^{\mathbf{a}}_{\mathbf{c}}$ (s/s).

System-propagation: We propagated the state estimates and error covariance matrix forward based on our known system dynamics and unknown uncertainties. We assumed that the system dynamics and system noise covariance is consistent over time, so defined the transition matrix Φ_{k-1} (KF Step 1) and system noise covariance matrix Q_{k-1} (KF Step 2) outside of our recursive loop. We completed the system-propagation phase within our recursive loop, for each epoch k, following the standard implementation of KF Steps 3-4.

Measurement update: Within our recursive loop, we corrected the state vector estimate and error covariance for each epoch to incorporate the new measurement information. Given we were working with GNSS data, we first used the weighted least squares velocity and positioning approach methodology in [2]. Namely, we corrected our estimates for each satellite, in each epoch for the Sagnac effect, C^I_e (s), using CalcSagnacMatrix.m, calculated the line of sight unit vector $\mathbf{u}^{\mathbf{e}}_{\mathbf{as}}$, and used these to update the predicted pseudo-ranges $\hat{r}^-_{as}$ (m), range rates $\hat{\dot{r}}^-_{as}$ (m/s), and measurement matrix $\mathbf{H_k}$ (KF Step 5). We then returned to the standard KF Steps 6-10 before saving our results and ending our calculation for the respective epoch.

Outlier removal: We use the function Outlier_Detection.m to inspect whether there are any outliers in our predicted velocity and position states, removing any outliers from the first epoch calculations, and stored an outlier matrix indicating whether the data (pseudo-range and pseudo-range-rate) is an outlier or not. We used the outlier detection methodology described in [3]. We used a detection threshold of 5 which was obtained through testing. Then in the GNSS Kalman Filter, we used Outlier_Data_Correction.m along with the calculated outlier matrix to ignore any outlier data.

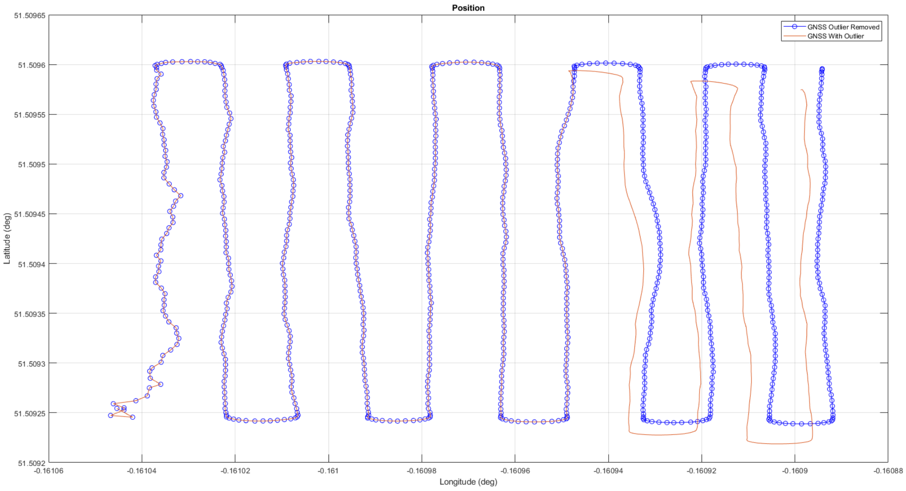


Figure 2: GNSS position outliers

Figure 2 shows the positioning before and after the outliers were removed. We noticed that both position solutions match initially, but the one with outliers would start to drift down later in the epoch. This is most likely due to the outlier measurements being propagated into following epochs, which resulted in snow-balling of errors. The removal of outliers significantly improved the positional solution.

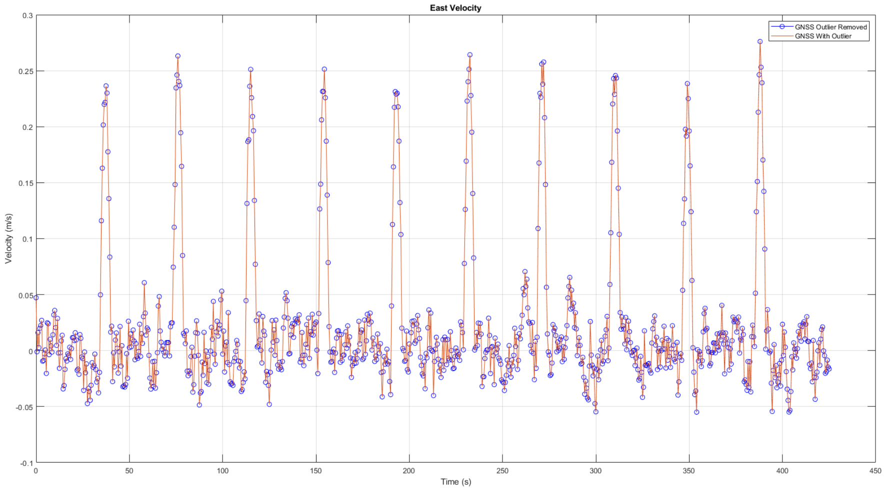


Figure 3: GNSS east velocity outlier

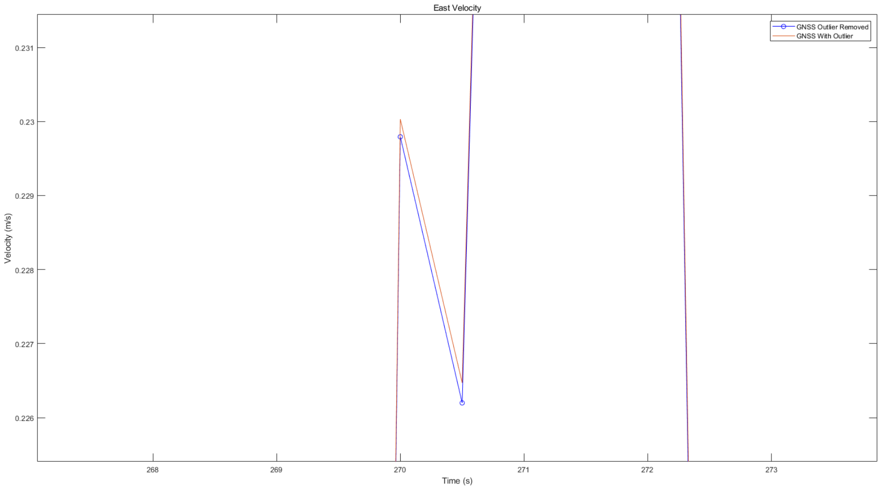


Figure 4: GNSS east velocity outlier zoomed in

Figure 3 shows the east velocity before and after the outliers were removed. This resulted in minor adjustments in the velocity, although they are difficult to see visually here and a zoomed in figure has been provided (See Figure 4).

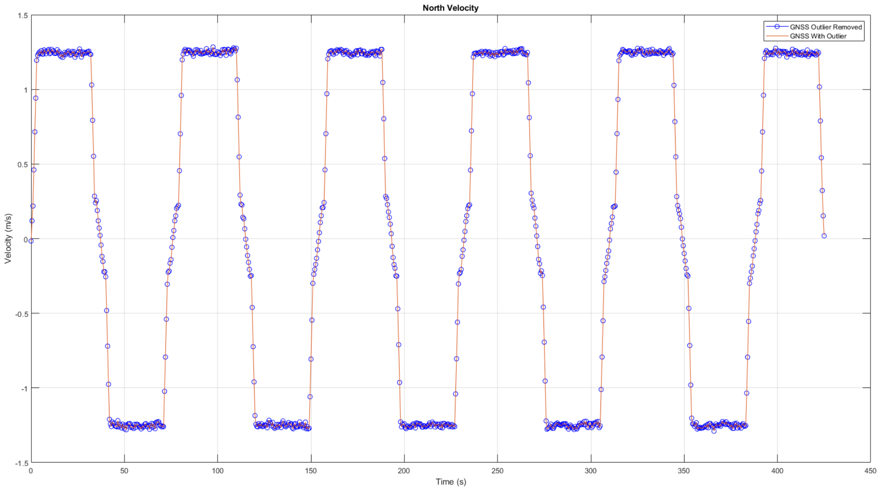


Figure 5: GNSS north velocity outlier

Figure 5 shows the north velocity before and after the outliers were removed. This had similar behaviour to that discussed regarding Figure 3 though even less noticeable.

Results: Our lawnmower looks to be on a lawn in Hyde Park, which is mostly flat and open grasslands. This means that there should be minimal blockage and reflection of signals, which implies that there wouldn't be many multi-path and non-line-of-sight errors.

CONTINUED

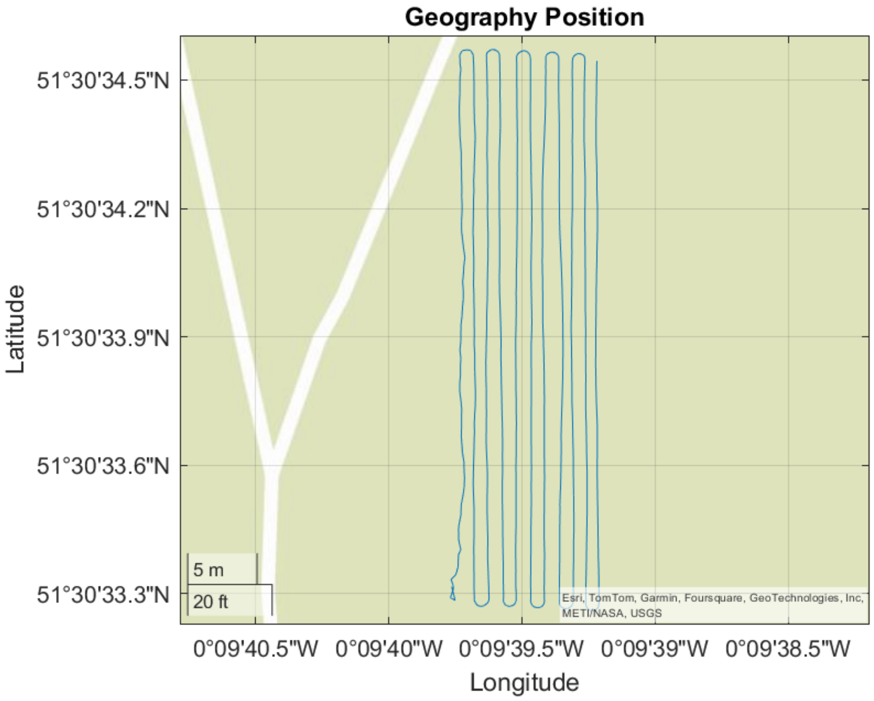


Figure 6: GNSS-only position solution GeoPlot

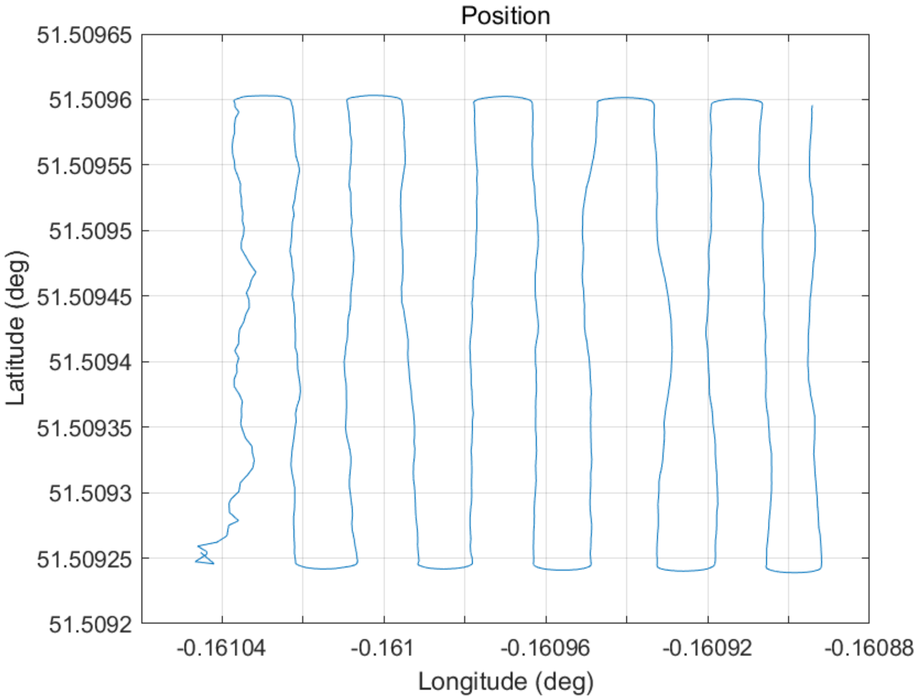


Figure 7: GNSS-only position solution

The position solution looks broadly reasonable, although the initial epochs seem to have greater errors which leads to rather a jagged trajectory. This may be because there are errors that are not being captured by our outlier threshold. Another possible explanation for this is may be due to the characteristics of a Kalman filter. Kalman filter is an iterative algorithm that continuously updates its state estimate based on new measurements and the dynamics of the system. As more measurements become available, the filter has more information to refine its estimates, which can potentially lead to improved performance later in the epochs.

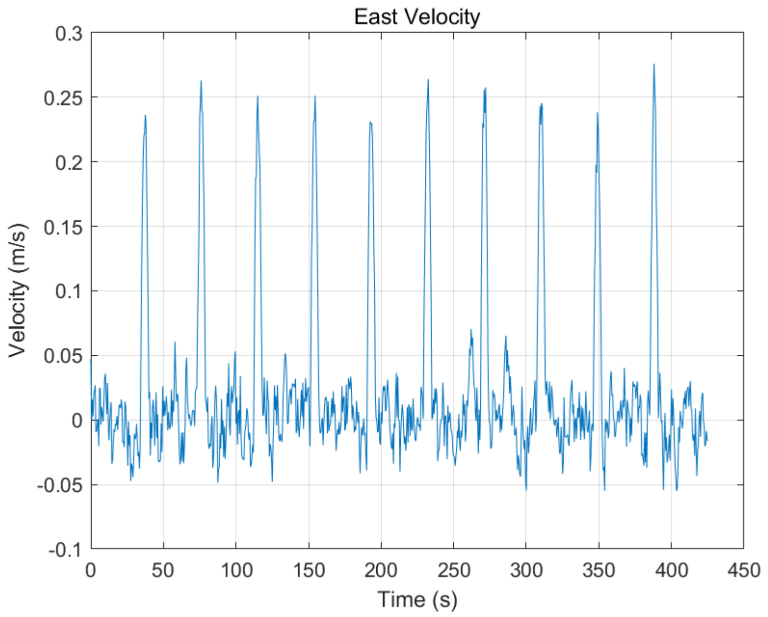


Figure 8: GNSS-only east velocity solution

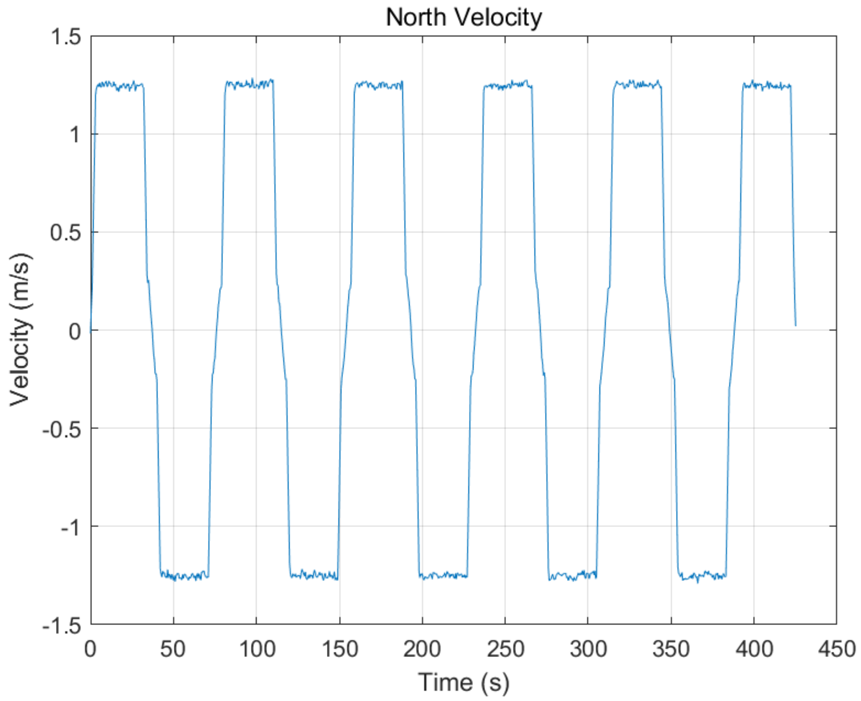


Figure 9: GNSS-only north velocity solution

Our velocity solutions look reasonable in comparison with our position estimates, reflecting that the lawnmower is covering land mostly in a north/south loop with tight turns towards the east. Figure 8 depicts the GNSS-only east velocity solution. The east velocity remains constant and increases when north velocity gets to 0 m/s. We do notice fluctuations in both velocities which may most likely be due to signal-in-space and atmospheric errors.

2.1.2 DR solution

We calculated a horizontal DR solution based on the wheel speed sensors, magnetic compass and gyroscope angular rate measurements, using the function Dead_Reckoning.m. In general, this function uses the same approach to DR calculation as in [4] Task 1. However, we used two modifications to the inputs.

First, as input we used a corrected gyroscope-derived heading solution using the basic open-loop integration architecture from [5, p. 38-41], as implemented in Gyro_Magnetometer_KF.m. As explained in [6], the accuracy of magnetic heading does not degrade over time but it may exhibit errors due to environmental magnetic anomalies. On the other hand, gyroscopes have high short term stability but their accuracy degrades over time, and also requires initialization. Combining both sensors allows us to amplify the advantages of both types of sensor.

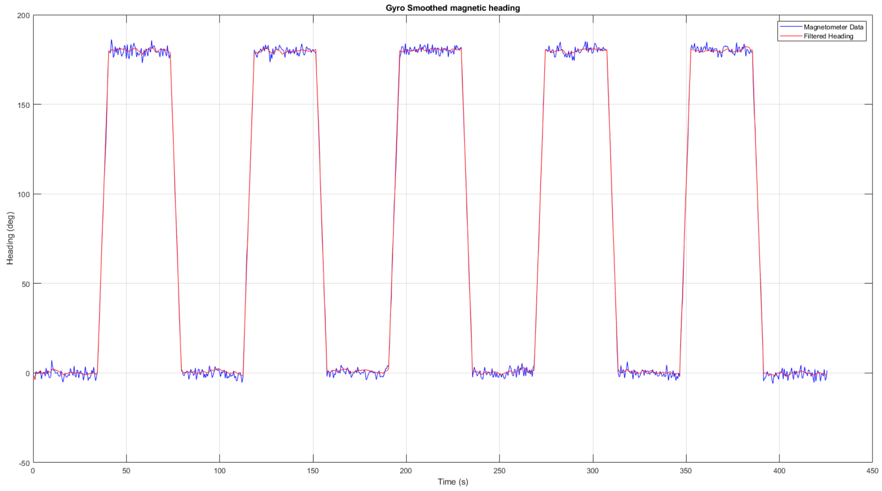


Figure 10: Gyro-smoothed magnetic heading

The impact this approach provides is visualised in Figure 10, which shows a comparison between sole magnetometer heading measurements provided in the csv file and the gyro-smoothed magnetic heading solution using the Kalman filter integration. We could clearly observe that the smoothed solution exhibits much less fluctuation compared to just using the magnetometer. This behaviour is expected as the lawnmower should have a rather smooth turn which should result in a similarly smooth change in heading.

We also tested two approaches for using the input from the four wheel speed sensors. In the function Calc_Change_Position.m, we implemented the approach described in [7, p. 234-237]. Specifically, we implemented equation 6.29 to calculate the change in position per epoch,

$\delta r_{eb}^{n}(t, t + \tau)$. We used the rear-wheel frame given the rear wheels are the driving wheels as specified in the coursework instructions. As the provided y-axis lever arm values were symmetrical, we assumed the offset of the frame from the GNSS receiver was solely the forward distance provided.[7] Following the approach in the text, we used these values to directly update the longitude, λ_b, and latitude, L_b, (equation 6.31) in an appropriately modified version of our DR function, Dead_Reckoning_modified.m.[7]

We compared this approach with simply averaging the wheel speeds from all four sensors in Calc_Avg_Speed.m. We used this value in the velocity calculation and position update approach, including damping, used in Task 1 in [4] as implemented in Dead_Reckoning.m.

The difference in the position solution between the two approaches can be seen by comparing the plot below with Figure 13 in the results in this section. Upon visual inspection the average method provided us with a smoother solution. We therefore decided to keep this version of the wheel speed inputs and DR calculation.

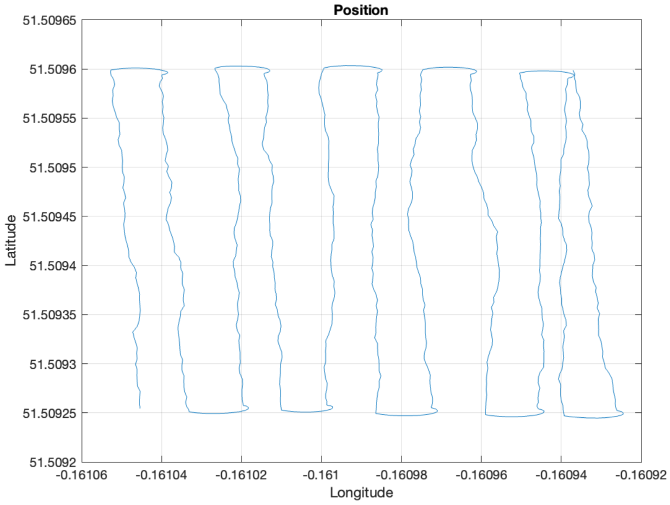


Figure 11: Rear-wheel linear odometry DR position solution

Figure 11 shows the DR position solution using the rear-wheel linear odometry input, which was discarded in favour of the simpler approach to averaging the wheel speed sensors.

Results: We know that DR systems tend to accumulate errors over time thus making this technique prone to drift [6]. We will address this by correcting the DR solution using our GNSS solution, as described in the next section. The velocity solutions for our DR approach look similar to the GNSS-only solutions though exhibiting much higher random noise.

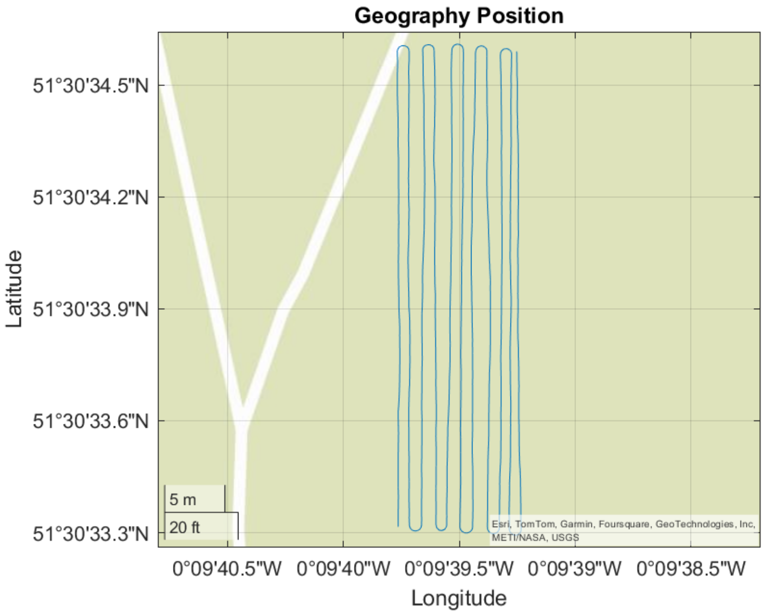


Figure 12: DR-only position solution GeoPlot

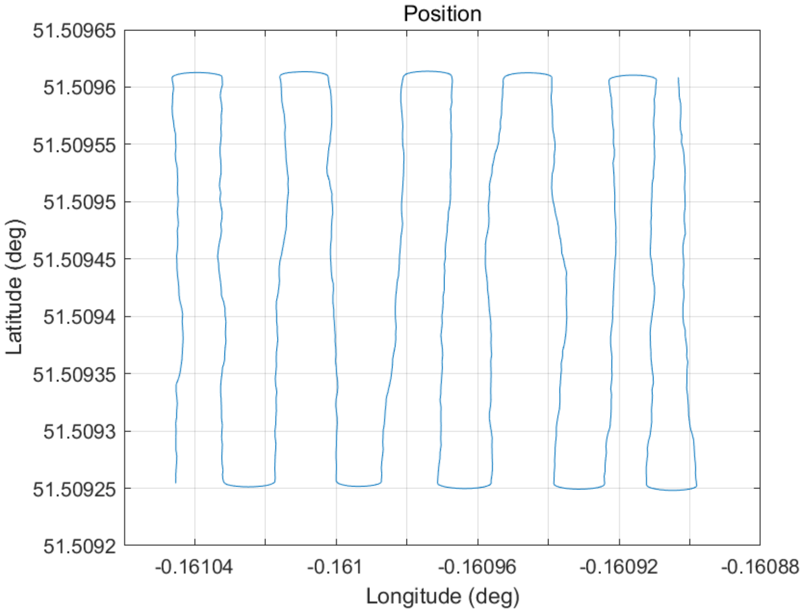


Figure 13: DR-only position solution

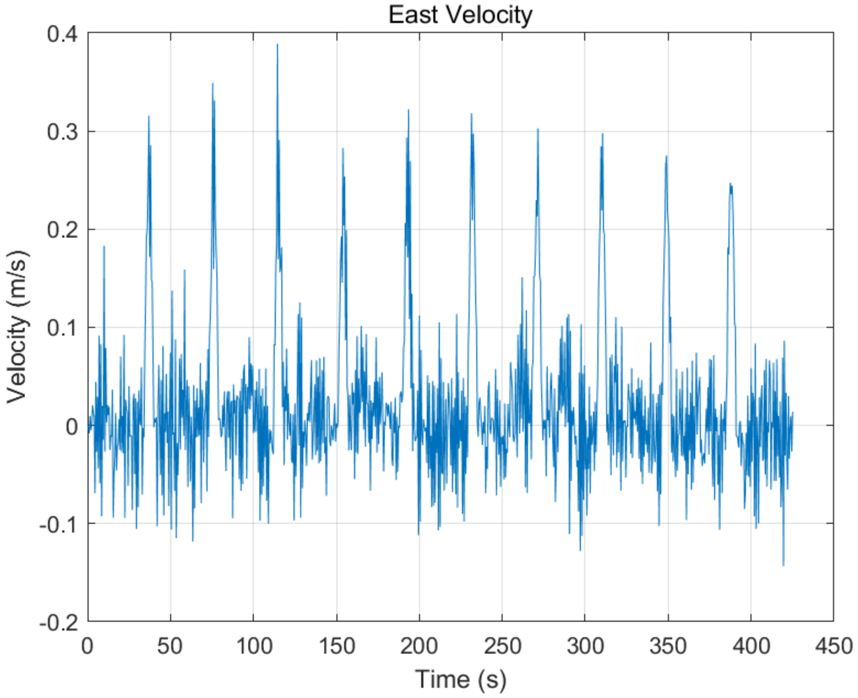


Figure 14: DR-only east velocity solution

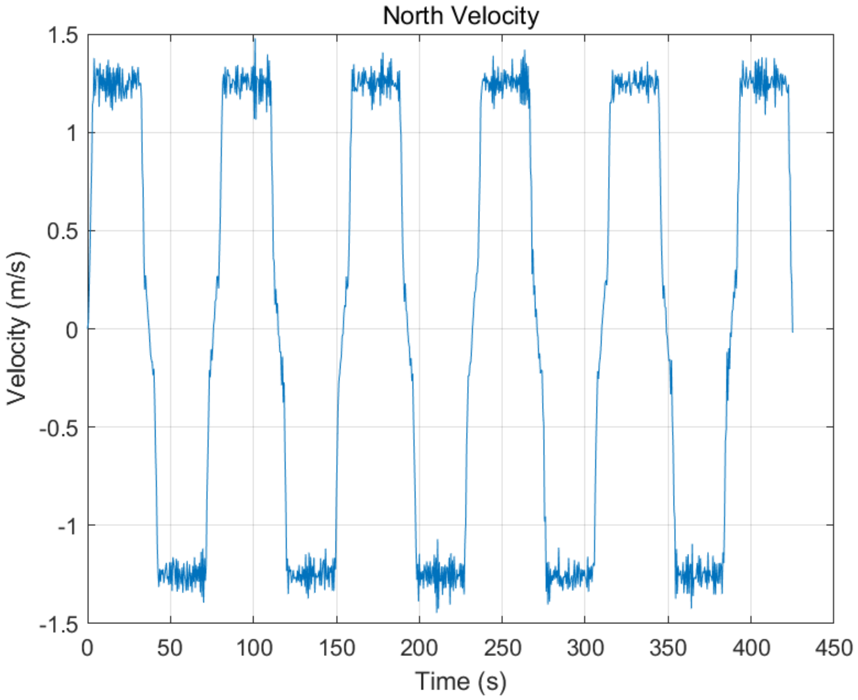


Figure 15: DR-only north velocity solution

CONTINUED

2.2 Final integrated solution

Finally, we utilised all sensors to calculate an integrated horizontal-only solution, relying on DR_GNSS.m. We used a basic loosely coupled open-loop integration approach as described in [5, p. 20]. We used our previously computed GNSS and DR navigation solutions, and corrected the DR solution using Kalman filter integration. Our state vector represented the 2D DR velocity $\delta v_N, \delta v_E$ and position error $\delta L, \delta\lambda$. We followed the basic KF Steps 1-10. The motivation for combining DR and GNSS solution is that GNSS measurements give rise to calculation of absolute position and velocities of the signal receiver, which tends to be rather noisy, whereas DR gives relative solutions but is prone to drift.[5] By combining both techniques, we could find the optimal solution based on these sensors.

Results: The solution can be found in final_solution.csv. The solution is broadly aligned with what we would expect from a robotic lawnmower.

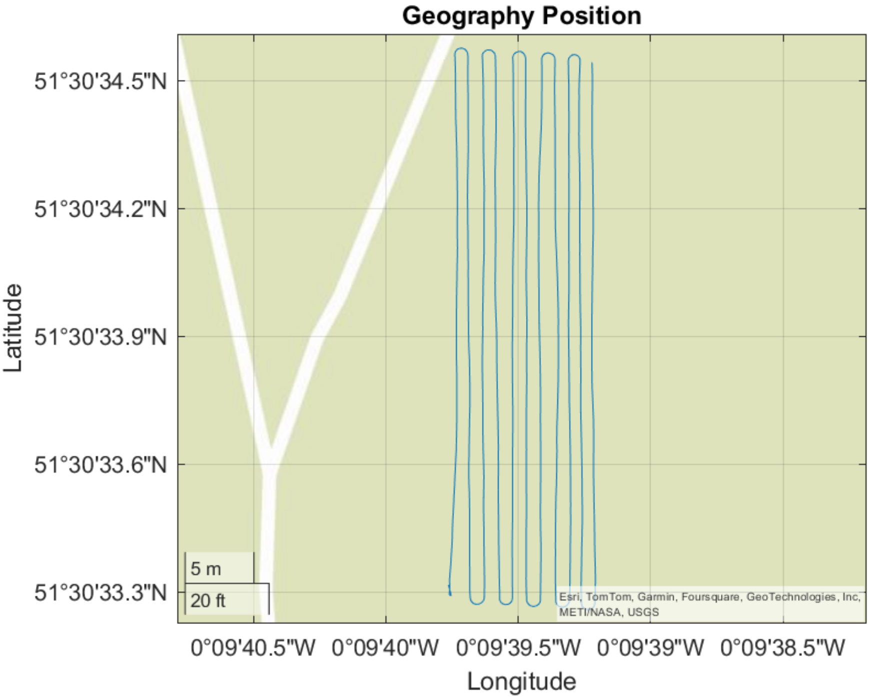


Figure 16: Integrated position solution GeoPlot

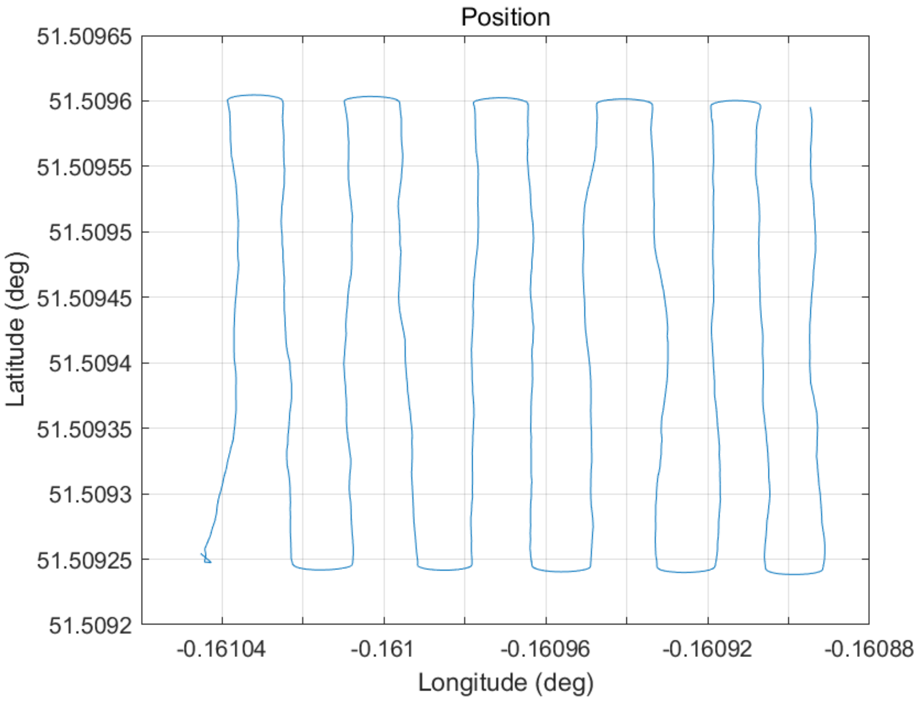


Figure 17: Integrated position solution

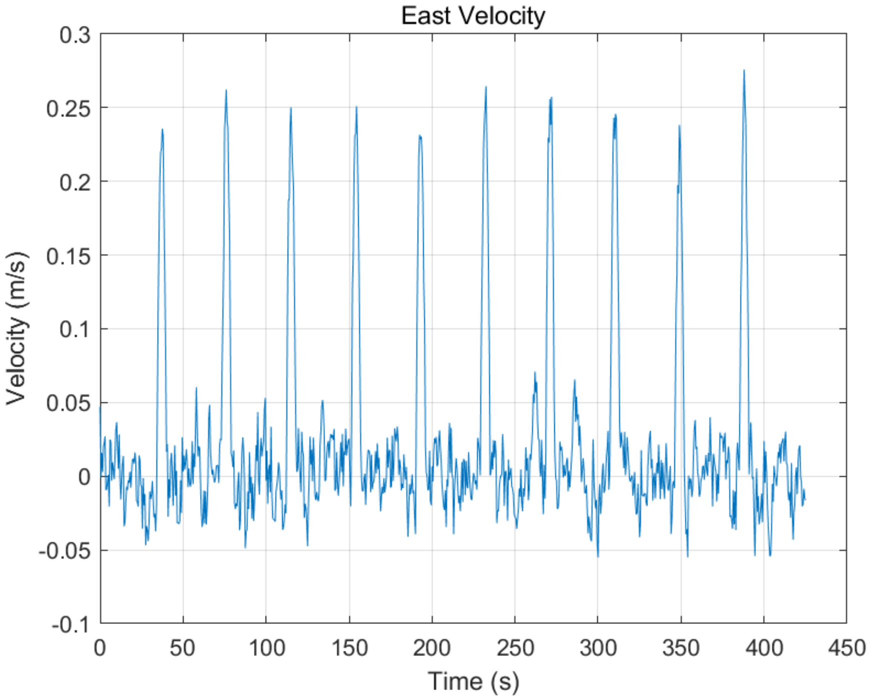


Figure 18: Integrated east velocity solution

CONTINUED

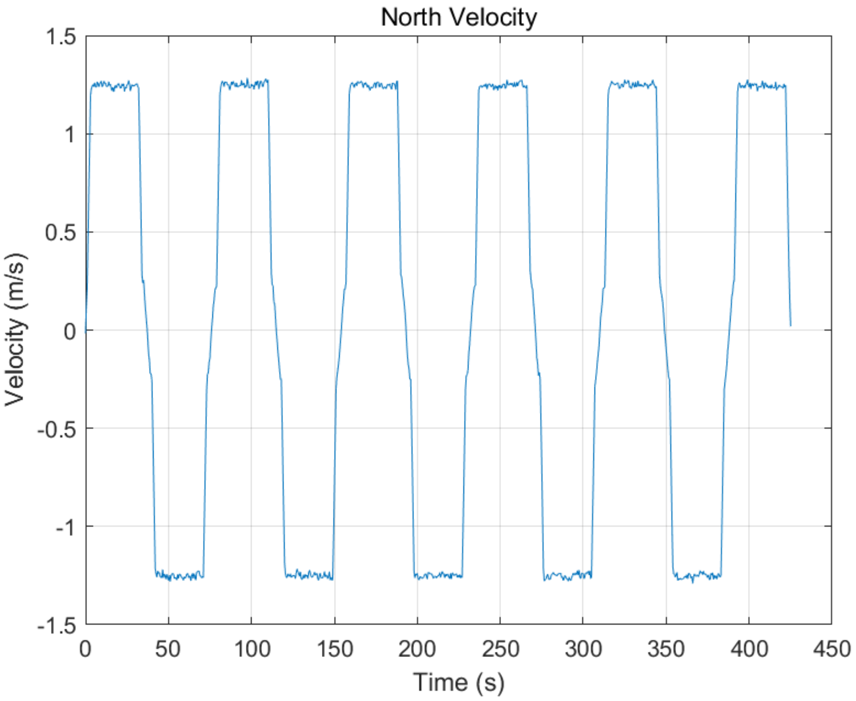


Figure 19: Integrated north velocity solution

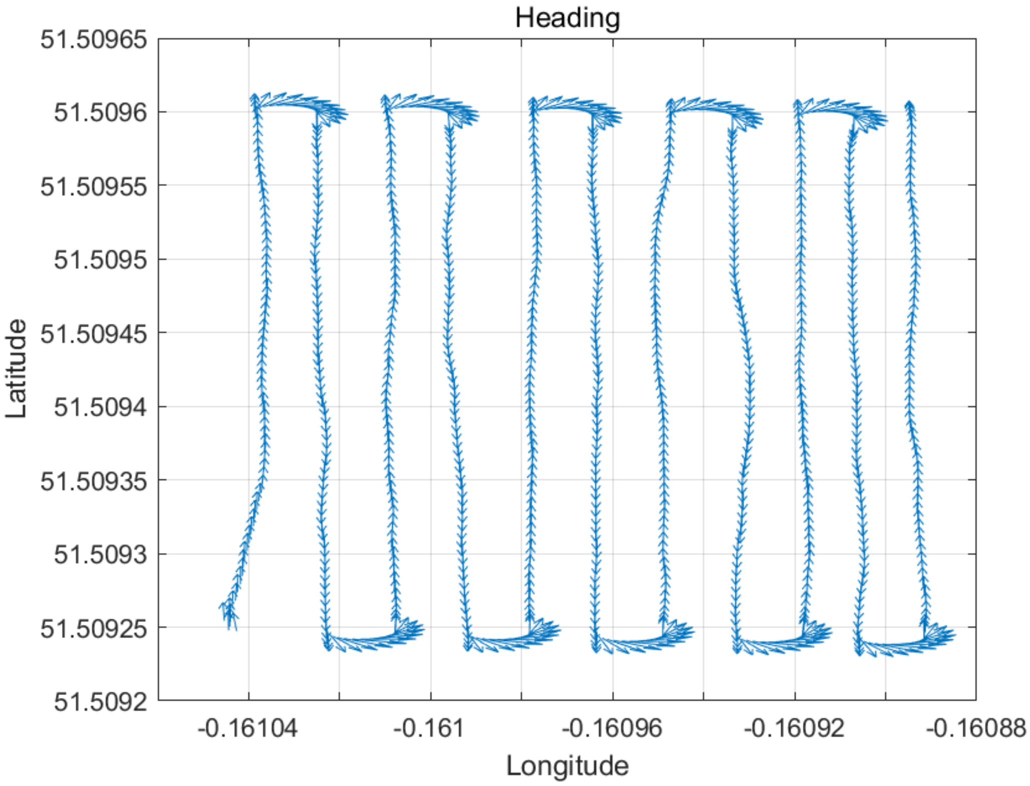


Figure 20: Heading solutions

We do not know the width of the area mowed - ideally the lawnmower's path would cover all of the grass in between its movements. We can see that the lawnmower seems to be moving in relatively straight lines and then turning. This indicates that the likely position inaccuracy in early epochs of the GNSS-only solution, and in the end epochs of the DR-only solution, has been addressed.

Figure 20 shows the quiver plot of the gyro-smoothed magnetic heading plotted along the final integrated position solution, which also indicates that our corrected heading is reasonable.

3 Final Discussion

The figures below compare the impact of the adjustments described in the methodology: the GNSS only solution, DR only solution and the integrated DR-GNSS solution.

Shown in Figure 21 are the positional solutions, here we would observe the drifting effect that we have mentioned in the DR section. In earlier epochs, it is evident that the DR solution exhibits less noise in the longitude data compared to GNSS. This is particularly noticeable in the integrated trajectory, as depicted by the DR-GNSS, where the trajectory appears to be more smooth. It is also expected that GNSS solution and the integrated solution are very close to each other as GNSS measurements allow us to calculate absolute position of the signal receiver.

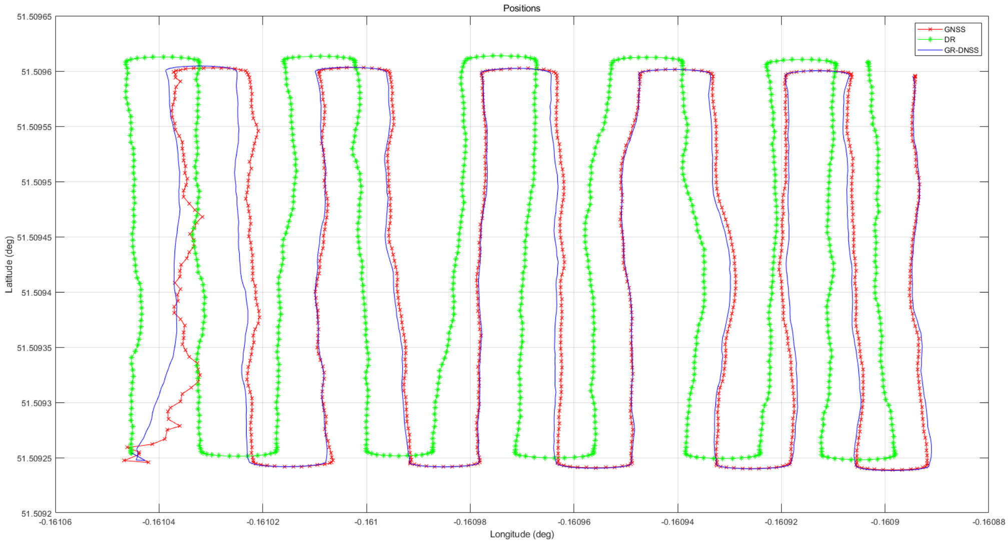


Figure 21: Comparison of position solutions

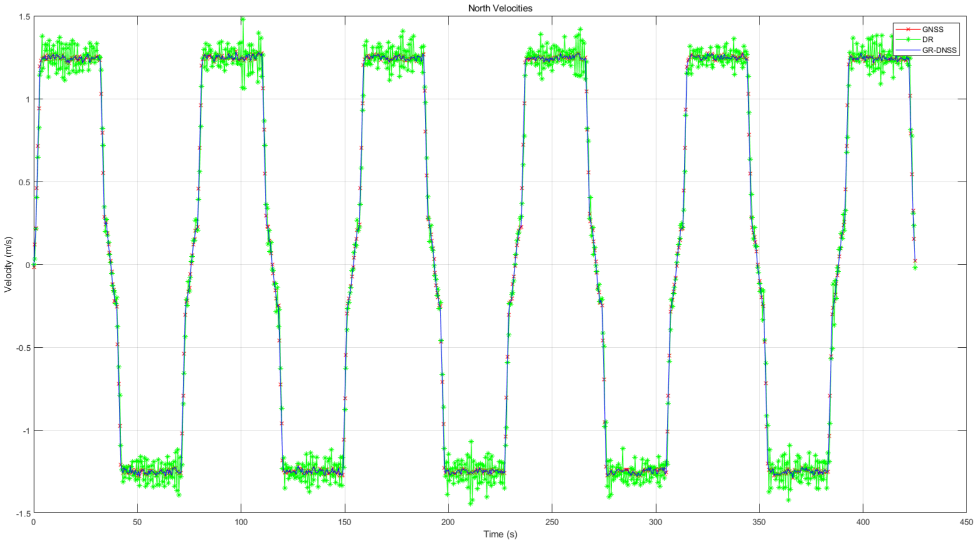


Figure 22: Comparison of north velocity solutions

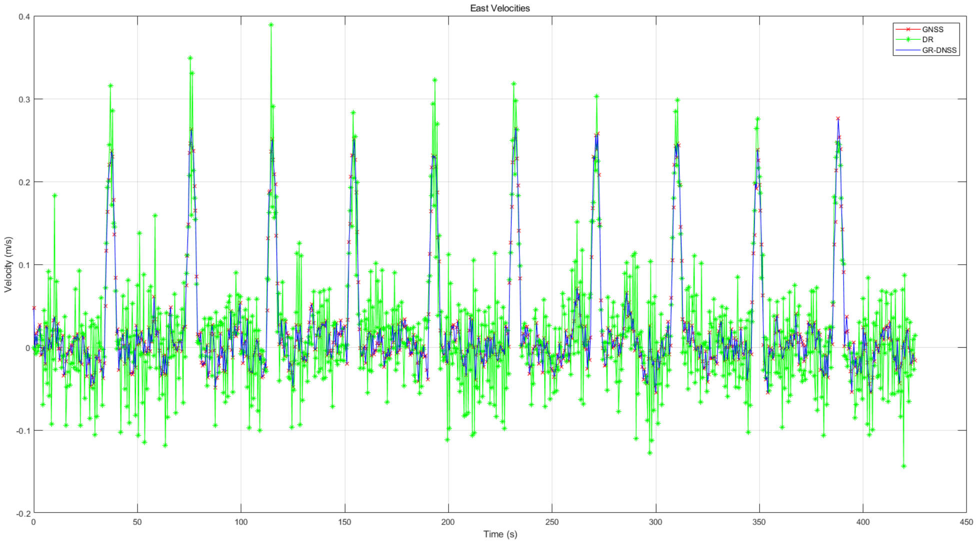
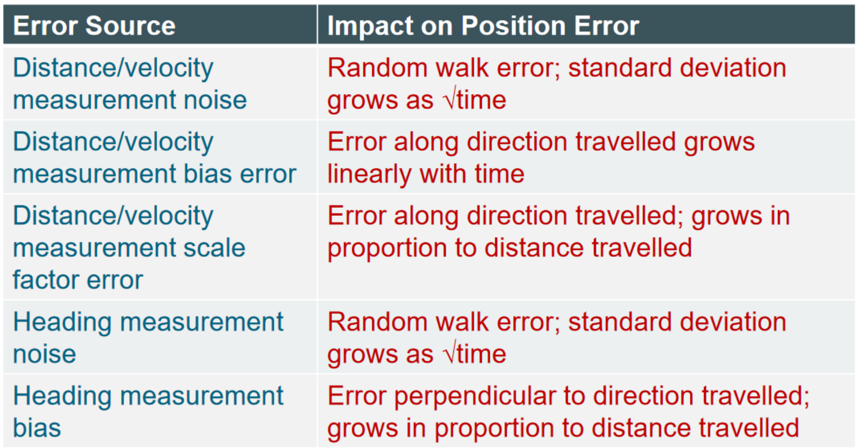


Figure 23: Comparison of east velocity solutions

Regarding the velocity solutions illustrated in Figure 22 and Figure 23, both GNSS and DR-GNSS methodologies appear to provide comparable velocity outcomes. However, it is noteworthy that the DR method exhibits significantly greater random errors when compared to the other approaches. This may be due to factors such as variation in terrain and environmental conditions. In addition, DR heavily relies on sensor measurements and does not have access to any absolute information, thus making it more prone to errors sources listed in Figure 24 as described in [6, p. 60].



Error Source	Impact on Position Error
Distance/velocity measurement noise	Random walk error; standard deviation grows as √time
Distance/velocity measurement bias error	Error along direction travelled grows linearly with time
Distance/velocity measurement scale factor error	Error along direction travelled; grows in proportion to distance travelled
Heading measurement noise	Random walk error; standard deviation grows as √time
Heading measurement bias	Error perpendicular to direction travelled; grows in proportion to distance travelled

Figure 24: Dead reckoning error sources

4 Bibliography

References

[1] Paul D. Groves. "GNSS and the Kalman FIlter Lecture 2B". In: *COMP0130: Robot Vision and Navigation* (2024), p. 40.

[2] Paul D. Groves. "GNSS and Least-squares estimation Lecture 1D". In: *COMP0130: Robot Vision and Navigation* (2024).

[3] Paul D. Groves. "Quality Control Lecture 1E". In: *COMP0130: Robot Vision and Navigation* (2024), p. 14.

[4] Paul D. Groves. "Workshop 3: Multisensor Navigation". In: *COMP0130: Robot Vision and Navigation* (2024).

[5] Paul D. Groves. "Integrated Navigation Lecture 3B". In: *COMP0130: Robot Vision and Navigation* (2024).

[6] Paul D. Groves. "Integrated Navigation Lecture 3A". In: *COMP0130: Robot Vision and Navigation* (2024), p. 54.

[7] Paul D. Groves. *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems (2nd Edition)*. Artech House, 2013, pp. 234–237. ISBN: 978-1-60807-005-3. URL: https://app.knovel.com/hotlink/toc/id:kpPGNSSIM9/principles-gnss-inertial/principles-gnss-inertial.

5 Code

5.1 Define_Constants.m

```
%Define_Constants
%SCRIPT This defines a number of constants for your use
% Created 16/11/16 by Paul Groves

% Copyright 2016, Paul Groves
```

CONTINUED

```
% License: BSD; see license.txt for details

% Modified by Group 5 Feb 2024

% Constants
deg_to_rad = 0.01745329252; % Degrees to radians conversion
   factor
rad_to_deg = 1/deg_to_rad; % Radians to degrees conversion
   factor
c = 299792458; % Speed of light in m/s
omega_ie = 7.292115E-5;  % Earth rotation rate in rad/s
Omega_ie = Skew_symmetric([0,0,omega_ie]);
R_0 = 6378137; %WGS84 Equatorial radius in meters
e = 0.0818191908425; %WGS84 eccentricity

sigma_r = 10; % Initial position standard deviation, assuming
   same for all directions
sigma_v = 0.1; % Initial velocity standard deviation, assuming
    same for all directions
sigma_co = 100000; % Initial clock offset standard deviation
sigma_cd = 200; % Reciever clock drift standard deviation

sigma_space = 1; % Signal in space error std
sigma_res_ion = 2; % Residual ionosphere error standard
   deviation at zenith
sigma_res_trop = 0.2; % Residual troposphere error standard
   deviation at zenith
sigma_code_multipath = 2; % Code tracking and multipath error
   std
sigma_rho_dot_multipath = 0.02; % Range rate tracking and
   multipath error std

sigma_rho_assumed = 10; % Assumed Psuedo-range measurement
   error std
sigma_rho_dot_assumed = 0.05; % Assumed Pseudo-range rate
   measurement error std

sigma_rho = sigma_rho_assumed;
sigma_rho_dot = sigma_rho_dot_assumed;

S_c_phi = 0.01; % Clock phase PSD
S_cf = 0.04; % Clock frequency PSD

T = 5; % Outlier detection threshold

tau = 0.5; % Propagation interval
```

```
S_a = 0.01; % Acceleration power spectral density

sigma_Gr = 5; % GNSS position measurements error standard
   deviation
sigma_Gv = 0.02; % GNSS velocity measurements have an error
   standard deviation
S_DR = 0.2; % DR velocity error power spectral density

micro_g_to_meters_per_second_squared = 9.80665E-6;

% Loosely coupled INS/GNSS Kalman filter parameters
S_rg = 3e-06; % Gyro noise PSD (deg^2 per hour, converted to
   rad^2/s)
% Accelerometer noise PSD (micro-g^2 per Hz, converted to m^2
   s^-3)
LC_KF_config.accel_noise_PSD = (200 *...
    micro_g_to_meters_per_second_squared)^2;
% Accelerometer bias random walk PSD (m^2 s^-5)
LC_KF_config.accel_bias_PSD = 1.0E-7;

S_bgd = 2.0E-12; % Gyro bias random walk PSD (rad^2 s^-3)
% Position measurement noise SD per axis (m)
LC_KF_config.pos_meas_SD = 2.5;
% Velocity measurement noise SD per axis (m/s)
LC_KF_config.vel_meas_SD = 0.05;

sigma_mh = 4*deg_to_rad; % Magnetic heading error standard
   deviation
sigma_gyro = 0.0001; % Gyroscope angular rate error standard
   deviation

% Location of lever arms of rear-wheel frame from GNSS sensor
l_bry = 0; % Lever arms are equidistant from centre of frame
   on y-axis
l_brx = -0.2;

% Ends
```

5.2 CalcSagnacMatrix.m

```
function C_I_e = CalcSagnacMatrix(range)
% Calculates Sagnac matrix given the range prediction
% Inputs:
%   range                   Single range prediction value
%
% Outputs:
```

CONTINUED

```
%   C_I_e                   Calculated Sagnac matrix
    Define_Constants;
    rep = omega_ie * range/c;
    C_I_e = [1,rep,0;-rep,1,0;0,0,1];
end
```

5.3 Calc_Avg_Speed.m

```
function avg_speed = Calc_Avg_Speed(wheel_speed_measurements,
   gyro_measurement)
% Calculates average speed of vehicle based on speed
   measurements for each wheel
% Inputs:
%   wheel_speed_measurements        Array of speed
   measurements for each wheel (m/s)
%   gyro_measurment                 gyroscope angular rate
   measurements (rad/s)
% Output:
%   avg_speed                       Calculated average speed

%% TODO Apply compensation based on turn rate from gyro
   measurements, or averaging speeds of left and right wheels
avg_speed = mean(wheel_speed_measurements);
end
```

5.4 Calc_Change_Position.m

```
function [delta_r_rear] = Calc_Change_Position(
   wheel_speed_measurements, gyro_measurement,
   compass_measurement)
% Calculates velocity of vehicle based on speed measurements
   for each wheel
% Inputs:
%   wheel_speed_measurements        Array of speed
   measurements for each wheel (m/s)
%   gyro_measurement                 gyroscope angular rate
   measurements (rad/s)
%   compass_measurement              magnetic compass
   measurements (degrees)
% Output:
%   velocity                        Calculated velocity

Define_Constants;

% Define inputs
psi_nb = compass_measurement * deg_to_rad;
```

```
psi_dot_nb = gyro_measurement;

% Calculate driving (rear) wheel-frame average speed
v_erL = wheel_speed_measurements(1);
v_erR = wheel_speed_measurements(2);
v_er = 0.5 * (v_erL + v_erR);

% Calculate change in position of rear wheels
delta_r_rear_temp = [cos(psi_nb) - 0.5 * psi_dot_nb * tau *
   sin(psi_nb); ...
    sin(psi_nb) + 0.5 * psi_dot_nb * tau * cos(psi_nb)] * v_er
        * tau;

delta_r_rear = delta_r_rear_temp + [cos(psi_nb), -sin(psi_nb);
    sin(psi_nb), cos(psi_nb)] * [l_bry; -l_brx] * psi_dot_nb *
   tau;

end
```

5.5 DR_GNSS.m

```
function dr_gnss_solution = DR_GNSS(gnss_solution,dr_solution,
   times)
% Calculates an integrated horizontal-only DR/GNSS solution
   using Kalman
% filtering
% Inputs:
%   gnss_solution                            Calculated GNSS
   Kalman Filter solution
%   dr_solution                              Calculated Dead
   Reckoning solution
%   times                                    Array of time
   value for each epoch
%
% Outputs:
%   dr_gnss_solution                         Calculated DR/
   GNSS solution
%                                               Col 1: Time (s
   )
%                                               Col 2:
   Latitude (deg)
%                                               Col 3:
   Longitude (deg)
%                                               Col 4: North
   Velocity (m/s)
%                                               Col 5: East
```

```
   Velocity (m/s)

Define_Constants;

epoch_num = height(times);

% Initialization using GNSS solutions
gnss_h = gnss_solution(1,4);
gnss_lat = gnss_solution(1,2) * deg_to_rad;

[R_N,R_E]= Radii_of_curvature(gnss_lat);

% Initialize state estimatino error covariance matrix,
   assuming same
% velocity uncertainty in each direction and same position
   uncertainty per
% direction
P_k_m_1_plus = [sigma_v^2,0,0,0
                0,sigma_v^2,0,0
                0,0,sigma_r^2/(R_N+gnss_h)^2,0
                0,0,0,sigma_r^2/((R_E+gnss_h)^2*(cos(gnss_lat)
                   )^2)];

% State vector containing erros of states: [delta_v_N,
   delta_v_E,delta_latitude,delta_longitude]'
x_caret_k_m_1_plus = zeros(4,1);

% Compute transition matrix
Phi_k_m_1 = [1,0,0,0
             0,1,0,0
             tau/(R_N+gnss_h),0,1,0
             0,tau/((R_E+gnss_h)*(cos(gnss_lat))),0,1];

% Compute system noise covariance matrix
Q_k_m_1 = [S_DR*tau,0,0.5*S_DR*tau^2/(R_N+gnss_h),0
           0,S_DR*tau,0,0.5*S_DR*tau^2/((R_E+gnss_h)*cos(
              gnss_lat))
           0.5*S_DR*tau^2/(R_N+gnss_h),0,S_DR*tau^3/(3*(R_N+
              gnss_h)^2),0
           0,0.5*S_DR*tau^2/((R_E+gnss_h)*cos(gnss_lat)),0,
              S_DR*tau^3/(3*(R_E+gnss_h)^2*(cos(gnss_lat)^2))];

dr_gnss_solution = zeros(epoch_num,5);
dr_gnss_solution(1,:) = [gnss_solution(1,1),gnss_solution(1,2)
   ,gnss_solution(1,3),gnss_solution(1,5),gnss_solution(1,6)];
```

```
for i = 2:epoch_num

    time = times(i);

    % Load GNSS solutions for current epoch
    gnss_lat = gnss_solution(i,2) * deg_to_rad;
    gnss_long = gnss_solution(i,3) * deg_to_rad;
    gnss_h = gnss_solution(i,4);
    gnss_v_N = gnss_solution(i,5);
    gnss_v_E = gnss_solution(i,6);

    % Load DR solutions for current epoch
    dr_lat = dr_solution(i,2)*deg_to_rad;
    dr_long = dr_solution(i,3)*deg_to_rad;
    dr_v_N = dr_solution(i,4);
    dr_v_E = dr_solution(i,5);

    [R_N,R_E]= Radii_of_curvature(gnss_lat);

    % Propagate state estimates and error covariance matrix
    x_caret_k_minus = Phi_k_m_1*x_caret_k_m_1_plus;
    P_k_minus = Phi_k_m_1*P_k_m_1_plus*Phi_k_m_1'+Q_k_m_1;

    % Compute measurement matrix
    H_k = [0,0,-1,0
           0,0,0,-1
           -1,0,0,0
           0,-1,0,0];

    % Compute measurement noise covariance matrix assuming
       same error std
    % for position measurements (sigma_Gr) and same error std
       for velocity
    % measurements (sigma_Gv) per axis
    R_k = [sigma_Gr^2/(R_N+gnss_h)^2,0,0,0
           0,sigma_Gr^2/((R_E+gnss_h)^2*cos(gnss_lat)^2),0,0
           0,0,sigma_Gv^2,0
           0,0,0,sigma_Gv^2];

    % Compute Kalman gain matrix
    K_k = P_k_minus*H_k'*inv(H_k*P_k_minus*H_k'+R_k);

    % Formulate measurement innovation vector
    delta_z_m_k = [gnss_lat-dr_lat;
                   gnss_long - dr_long;
                   gnss_v_N-dr_v_N;
```

CONTINUED

```
                   gnss_v_E-dr_v_E
                   ]-H_k*x_caret_k_minus;

    % Update state estimates and error covaraince matrix
    x_caret_k_m_1_plus = x_caret_k_minus + K_k*delta_z_m_k;
    P_k_m_1_plus = (eye(4) - K_k*H_k)*P_k_minus;

    % Apply correction to DR solution of current epoch
    correction = [dr_v_N;dr_v_E;dr_lat;dr_long]-
       x_caret_k_m_1_plus;
    solution = [time,correction(3:4)'*rad_to_deg,correction
       (1:2)'];
    dr_gnss_solution(i,:) = solution;

    % Compute transition matrix k minus 1
    Phi_k_m_1 = [1,0,0,0
                 0,1,0,0
                 tau/(R_N+gnss_h),0,1,0
                 0,tau/((R_E+gnss_h)*(cos(gnss_lat))),0,1];

    % Compute system noise covariance matrix k minus 1
    Q_k_m_1 = [S_DR*tau,0,0.5*S_DR*tau^2/(R_N+gnss_h),0
               0,S_DR*tau,0,0.5*S_DR*tau^2/((R_E+gnss_h)*cos(
                  gnss_lat))
               0.5*S_DR*tau^2/(R_N+gnss_h),0,S_DR*tau^3/(3*(
                  R_N+gnss_h)^2),0
               0,0.5*S_DR*tau^2/((R_E+gnss_h)*cos(gnss_lat))
                  ,0,S_DR*tau^3/(3*(R_E+gnss_h)^2*(cos(gnss_lat
                  )^2))];

end

% Writing solution to csv file
outputTable = array2table(dr_gnss_solution);
writetable(outputTable,"Solutions/DR_GNSS_Solution.csv",'
   WriteVariableNames',0)

end
```

5.6 Dead_Reckoning.m

```
function dr_solution = Dead_Reckoning(dr_measurement_data,
   gnss_solution,heading_solution,times)
% Calculates Dead Reckoning solution
% Inputs:
%   dr_measurement_data                   Array of raw dead
```

```
   reckoning measurement data
%   gnss_solution                         Array gnss kalman
   filter solutions
%   heading_solution                      Array of gyro-smooth
   magnetic heading solutions
%   times                                 Array of time value
   for each epoch
%
% Outputs:
%   dr_solution                           Calculated DR solution
   , for each row
%                                             Col 1: Time (s)
%                                             Col 2: Latitude (
   deg)
%                                             Col 3: Longitude (
   deg)
%                                             Col 4: Damped
   North Velocity (m/s)
%                                             Col 5: Damped East
    Velocity (m/s)

Define_Constants;

dr_measurement = dr_measurement_data(:,2:end);

% Total count of epochs
epoch_num = height(times);

lat = gnss_solution(1,2) * deg_to_rad;
long = gnss_solution(1,3)*deg_to_rad;
h = gnss_solution(1,4);

% Array for storing calculated dead reckoning solutions
dr_solution = zeros(epoch_num,5);

for i=1:epoch_num
    time = times(i);

    % Caulates average speed of vehicle given wheel
       measurements
    wheel_speed_measurements = dr_measurement(i,1:4);
    gyro_measurement = dr_measurement(i,5);
    v_k = Calc_Avg_Speed(wheel_speed_measurements,
       gyro_measurement);

    heading_k = heading_solution(i) * deg_to_rad;
    if i==1
        % Assume the instantaneous velocity at first epoch is
           given by the
        % speed and heading measurements at that time
        v_N_k = v_k*cos(heading_k);
        v_E_k = v_k*sin(heading_k);
    else

        [R_N,R_E]= Radii_of_curvature(lat);

        % Heading from previous epoch
        heading_k_m_1 = dr_measurement(i-1,6) * deg_to_rad;

        v_vec = 0.5*v_k*[cos(heading_k)+cos(heading_k_m_1);sin
           (heading_k)+sin(heading_k_m_1)];
        v_cap_N_k = v_vec(1);
        v_cap_E_k = v_vec(2);
        v_N_k=1.7*v_cap_N_k-0.7*v_N_k;
        v_E_k=1.7*v_cap_E_k-0.7*v_E_k;

        % Longitude and latitude calculation
        prevTime = times(i-1);
        lat = lat + v_cap_N_k*(time - prevTime)/(R_N+h);
        long = long + v_cap_E_k*(time - prevTime)/((R_E+h)*cos
           (lat));
    end

    dr_solution(i,:)=[time,lat*rad_to_deg,long*rad_to_deg,
       v_N_k,v_E_k];
end

solutionTable = array2table(dr_solution);
writetable(solutionTable,"Solutions/DR_Solution.csv",'
   WriteVariableNames',0)

end
```

5.7 Dead_Reckoning_modified.m

```
function dr_solution = Dead_Reckoning_modified(
   dr_measurement_data,gnss_solution,heading_solution,times)
% Calculates Dead Reckoning solution
% Inputs:
%   dr_measurement_data                     Array of raw dead
   reckoning measurement data
```

```
%   gnss_solution                       Array gnss kalman
   filter solutions
%   heading_solution                    Array of gyro-smooth
   magnetic heading solutions
%   times                               Array of time value
   for each epoch
%
% Outputs:
%   dr_solution                         Calculated DR solution
   , for each row
%                                           Col 1: Time (s)
%                                           Col 2: Latitude (
   deg)
%                                           Col 3: Longitude (
   deg)
%                                           Col 4: Damped
   North Velocity (m/s)
%                                           Col 5: Damped East
    Velocity (m/s)

Define_Constants;

dr_measurement = dr_measurement_data(:,2:end);

% Total count of epochs
epoch_num = height(times);

lat = gnss_solution(1,2) * deg_to_rad;
long = gnss_solution(1,3)*deg_to_rad;
h = gnss_solution(1,4);

% Array for storing calculated dead reckoning solutions
dr_solution = zeros(epoch_num,5);
delta_r_N_prev = zeros(epoch_num, 1);
delta_r_E_prev = zeros(epoch_num, 1);


for i=1:epoch_num
    time = times(i);

    % Caulates average speed of vehicle given wheel
       measurements
    wheel_speed_measurements = dr_measurement(i,1:4);
    gyro_measurement = dr_measurement(i,5);
    compass_measurement = dr_measurement(i, 6);
    heading_k = heading_solution(i) * deg_to_rad;

    if i==1
        % Assume the instantaneous velocity at first epoch is
           given by the
        % speed and heading measurements at that time
        v_k = Calc_Avg_Speed(wheel_speed_measurements);
        v_N_k = v_k*cos(heading_k);
        v_E_k = v_k*sin(heading_k);
    else
        [R_N,R_E]= Radii_of_curvature(lat);

        % Heading from previous epoch
        heading_k_m_1 = dr_measurement(i-1,6) * deg_to_rad;

        delta_r = Calc_Change_Position(
           wheel_speed_measurements,gyro_measurement,
           compass_measurement);
        delta_r_N = delta_r(1);
        delta_r_E = delta_r(2);

        % Store the delta_r values
        delta_r_N_prev(i) = delta_r_N;
        delta_r_E_prev(i) = delta_r_E;

        % Longitude and latitude calculation
        lat = lat + delta_r_N/(R_N+h);
        long = long + delta_r_E/((R_E+h)*cos(lat));

        % Calculate to save velocities
        v_N_k = delta_r_N / tau;
        v_E_k = delta_r_E / tau;
    end

    dr_solution(i,:)=[time,lat*rad_to_deg,long*rad_to_deg,
       v_N_k,v_E_k];
end

solutionTable = array2table(dr_solution);
writetable(solutionTable,"Solutions/DR_Solution.csv",'
   WriteVariableNames',0)

end
```

5.8 GNSS_KF.m

```
function gnss_solution = GNSS_KF(pseudo_range_data,
   pseudo_range_rate_data,times,fix_outlier)
% Calculates GNSS Kalman Filter estimation, saving solutions
   in GNSS_Solution.csv
% Inputs:
%   pseudo_range_data                 Array of raw pseudo
   range measurement data
%   pseudo_range_rate_data            Array of raw pseudo
   range rate measurement data
%   times                             Array of time value for
   each epoch
%   fix_outlier                       Flag indicating for
   fixing outliers
%                                     or not
%
% Outputs:
%   gnss_solution                     Calculated GNSS KF
   solution, for each row
%                                       Col 1: Time (s)
%                                       Col 2: Latitude (deg)
%                                       Col 3: Longitude (deg)
%                                       Col 4: Height (m)
%                                       Col 5: North Velocity
   (m/s)
%                                       Col 6: East Velocity (
   m/s)
%                                       Col 7: Down Velocity (
   m/s)

% Initializing constants
Define_Constants;

% Initialize GNSS KF Calculation constants
satellite_numbers = pseudo_range_data(1,2:end); % Array of
   satellite numbers
satellite_count = length(satellite_numbers); % Total count of
   satellites

% Total count of epochs
epoch_num = height(times);

% Extracting psuedo-range and pseudo-range rate only data
pseudo_ranges = pseudo_range_data(2:end,2:end);
pseudo_range_rates = pseudo_range_rate_data(2:end,2:end);
```

CONTINUED

```
% Array for saving results
gnss_solution = zeros(epoch_num,7);

% Initialize initial state for Kalman Filter, giving initial
   x_est and P_matrix
[x_caret_plus_k_m_1,P_plus_k_m_1,valid_pseudo_range_indicies,
   valid_pseudo_range_rate_indicies] = Initialise_GNSS_KF(times
   ,pseudo_ranges,pseudo_range_rates,satellite_numbers,
   fix_outlier);

% disp(valid_satellite_indicies)
% Compute transition matrix
phi_k_m_1 = [eye(3),eye(3)*tau,zeros(3,1),zeros(3,1)
             zeros(3),eye(3),zeros(3,1),zeros(3,1)
             zeros(1,3),zeros(1,3),1,tau
             zeros(1,3),zeros(1,3),0,1];

% Compute system noise covariance matrix
Q_k_m_1 = [S_a*tau^3*eye(3)/3,S_a*tau^2*eye(3)/2,zeros(3,1),
   zeros(3,1)
           S_a*tau^2*eye(3)/2,S_a*tau*eye(3),zeros(3,1),zeros
              (3,1)
           zeros(1,3),zeros(1,3),S_c_phi*tau+S_cf*tau^3/3,S_cf
              *tau^2/2
           zeros(1,3),zeros(1,3),S_cf*tau^2/2,S_cf*tau
          ];

% Initialization of predicted psuedo ranges
r_caret_as_minus = zeros(satellite_count,1);

for j = 1:satellite_count
    [sat_r_es_e,~]= Satellite_position_and_velocity(times(1),
       satellite_numbers(j));
    diff = sat_r_es_e.'-x_caret_plus_k_m_1(1:3);

    % Range Prediction initialization assuming identity sagnac
        effect matrix
    r_caret_as_minus(j) = sqrt(diff.'*diff);
end

% Line of sight and measurement matrix initialization
u_e_a = zeros(3,satellite_count);
H_k = zeros(satellite_count*2,8);
H_k(1:satellite_count,7) = 1;
H_k(satellite_count+1:end,8) = 1;
```

```
% Range rate predictions initialization
r_caret_dot_as_minus = zeros(satellite_count,1);

% Measurement innovation initialization
delta_z_min = zeros(satellite_count*2,1);

for epoch = 1:epoch_num

    % Load data for current epoch
    time = times(epoch);
    current_epoch_pseudo_ranges = pseudo_ranges(epoch,:);
    current_epoch_pseudo_range_rates = pseudo_range_rates(
       epoch,:);
    current_epoch_valid_pseudo_ranges_indicies =
       valid_pseudo_range_indicies(epoch,:);
    current_epoch_valid_pseudo_range_rate_indicies =
       valid_pseudo_range_rate_indicies(epoch,:);

    % Propagate state estimate and error covariance
    x_caret_minus_k = phi_k_m_1 * x_caret_plus_k_m_1;
    P_minus_k = phi_k_m_1*P_plus_k_m_1*phi_k_m_1.' + Q_k_m_1;

    % Extract each component of the state estimate vector
    r_caret_e_minus_ea_k = x_caret_minus_k(1:3);
    v_caret_e_minus_ea_k = x_caret_minus_k(4:6);
    roh_caret_a_minuc_c = x_caret_minus_k(7);
    roh_dot_caret_a_minuc_c = x_caret_minus_k(8);

    for j = 1:satellite_count
        % Calculate Sagnac matrix given range prediction of
           current
        % satellite
        C_I_e = CalcSagnacMatrix(r_caret_as_minus(j));

        % Range prediction from approximate user position to
           satellite
        [sat_r_es_e,sat_v_es_e]=
           Satellite_position_and_velocity(time,
           satellite_numbers(j));
        diff = C_I_e*sat_r_es_e.'-r_caret_e_minus_ea_k;
        r_caret_as_minus(j) = sqrt(diff.'*diff);

        % Line of sight vector calculation
        u_e_a(:,j) = diff / r_caret_as_minus(j);

        % Range rate prediction
```

CONTINUED

```
        r_caret_dot_as_minus(j) = u_e_a(:,j).' * (C_I_e*(
           sat_v_es_e.'+Omega_ie*sat_r_es_e.')-(
           v_caret_e_minus_ea_k+Omega_ie*r_caret_e_minus_ea_k))
           ;

        % Measurement matrix calculation
        H_k(j,1:3) = -u_e_a(:,j);
        H_k(j+satellite_count,4:6) = -u_e_a(:,j);
    end

    % Measurement noise covariance matrix calculation
    R_k = [eye(8,8) * sigma_rho^2,zeros(8,8)
           zeros(8,8),eye(8,8)*sigma_rho_dot^2];

    % Kalman gain matrix calculation
    K_k = P_minus_k * H_k.'*inv(H_k*P_minus_k*H_k.'+R_k);

    % Measurement innovation calculation

    for j = 1:satellite_count
        if current_epoch_valid_pseudo_ranges_indicies(j)
            delta_z_min(j) = current_epoch_pseudo_ranges(j) -
               r_caret_as_minus(j) - roh_caret_a_minuc_c;
        end
        if current_epoch_valid_pseudo_range_rate_indicies(j)
            delta_z_min(j+satellite_count) =
               current_epoch_pseudo_range_rates(j) -
               r_caret_dot_as_minus(j) -
               roh_dot_caret_a_minuc_c;
        end
    end

    % Update state estimates and error covariance
    x_caret_plus_k_m_1 = x_caret_minus_k + K_k * delta_z_min;
    P_plus_k_m_1 = (eye(8)-K_k*H_k)*P_minus_k;

    r_eb_e = x_caret_plus_k_m_1(1:3);
    v_eb_e = x_caret_plus_k_m_1(4:6);

    % Convert Cartesian ECEF solution to NED
    [L_b,lambda_b,h_b,v_eb_n] = pv_ECEF_to_NED(r_eb_e,v_eb_e);
    L_b = L_b * rad_to_deg;
    lambda_b = lambda_b * rad_to_deg;
    gnss_solution(epoch,:) = [time,L_b,lambda_b,h_b,v_eb_n.'];
end
```

```
if fix_outlier
    outputTable = table(gnss_solution);
    writetable(outputTable,"Solutions/GNSS_Solution"+".csv",'
       WriteVariableNames',0)
end
end
```

5.9 GNSS_Least_Squares.m

```
function [x_est,H_G_e,delta_z_min,delta_z_dot_min,
   r_caret_ea_e_minus,v_caret_ea_e_minus] = GNSS_Least_Squares(
   time,epoch_pseudo_range_measurements,
   epoch_pseudo_range_rate_measurements,satellite_numbers,
   r_caret_ea_e_minus,v_caret_ea_e_minus)
% Calculates least square solution for current epoch GNSS data
%
% Inputs:
%   epoch                                           Index of
   current epoch
%   times                                           Time of current
    epoch
%   epoch_pseudo_range_measurements                 Array of pseudo
   -range measurements for all epochs from all satellites
%   epoch_pseudo_range_rate_measurements            Array of pseudo
   -range rate measurements for all epochs from all satellites
%   satellite_numbers                               Array of the
   number given to each of the satellites
%   r_caret_ea_e_minus                              User position
   prediction
%   v_caret_ea_e_minus                              User velocity
   prediction
%
% Outputs:
%   x_est                                           Vector of
   initial state estimates:
%                                                        Rows 1-3
              estimated ECEF user position (m)
%                                                        Rows 4-6
              estimated ECEF user velocity (m/s)
%                                                        Row 7
                 estimated receiver clock offset (m)
%                                                        Row 8
                 estimated receiver clock drift (m/s)
%  H_G_e                                            Calculated
   Measurement Matrix
%  delta_z_min                                      Calculate
```

CONTINUED

```
   innovation vector for positions
%  delta_z_dot_min                              Calculate
   innovation vector for velocities
%  r_caret_ea_e_minus                           Position
   prediction for next epoch
%  v_caret_ea_e_minus                           Velocity
   prediction for next epoch

Define_Constants;

% Load in given data at current epoch
number_of_satellites = length(satellite_numbers);

% Initialization of range prediction array
r_caret_as_minus = zeros(1,number_of_satellites);

% Initialization of range rate prediction array
r_caret_dot_as_minus = zeros(1,number_of_satellites);

% Initialization of measurement Innovation vector for position
delta_z_min = zeros(number_of_satellites,1);

% Initialization of measurement Innovation vector for velocity
delta_z_dot_min = zeros(number_of_satellites,1);

r_caret_es_e = zeros(3,number_of_satellites);
v_caret_es_e = zeros(3,number_of_satellites);
H_G_e = zeros(number_of_satellites,4);

% Predicted receiver clock offset
delta_rho_caret_c_a_minus = 0;

%Predicted receiver clock drift
delta_rho_caret_dot_c_a_minus = 0;

for j = 1:number_of_satellites
    % Compute Cartesian ECEF positions and velocities of
       satellites at epoch 0
    [sat_r_caret_es_e,sat_v_es_e]=
       Satellite_position_and_velocity(time,satellite_numbers(j
       ));
    r_caret_es_e(:,j) = sat_r_caret_es_e;
    v_caret_es_e(:,j) = sat_v_es_e;

    % Range Prediction initialization assuming identity sagnac
        effect matrix
    diff = r_caret_es_e(:,j)-r_caret_ea_e_minus;
    r_caret_as_minus(j) = sqrt(diff.'*diff);
end

diff = 1;

% Calculate least square solution for state estimate
while diff > 0.0001

    for j = 1:number_of_satellites
        % Caculate Sagnac matrix given range prediction of
           current satellite
        C_I_e = CalcSagnacMatrix(r_caret_as_minus(j));
        diff = C_I_e*r_caret_es_e(:,j)-r_caret_ea_e_minus;

        % Calculation for the ranges prediction of current
           satellite
        r_caret_as_minus(j) = sqrt(diff.'*diff);

        % Compute line-of-sight unit vector
        u_caret_aj_e_minus = (C_I_e*r_caret_es_e(:,j)-
           r_caret_ea_e_minus)/r_caret_as_minus(j);

        % Calculation for the range rate prediction of current
            satellite
        a = C_I_e*(v_caret_es_e(:,j)+Omega_ie*r_caret_es_e(:,j
           ));
        b = v_caret_ea_e_minus + Omega_ie*r_caret_ea_e_minus;
        r_caret_dot_as_minus(j) = u_caret_aj_e_minus' * (a-b);

        % Calculating Measurement Matrix
        H_G_e(j,1:3) = -u_caret_aj_e_minus.';
        H_G_e(j,4) = 1;

        % Calculate innovation vectors for position and
           velocity
        delta_z_min(j) = epoch_pseudo_range_measurements(j)-
           r_caret_as_minus(j)-delta_rho_caret_c_a_minus;
        delta_z_dot_min(j) =
           epoch_pseudo_range_rate_measurements(j)-
           r_caret_dot_as_minus(j)-
           delta_rho_caret_dot_c_a_minus;
    end

    % Formulate predicted state vector for position
    x_caret_min = [r_caret_ea_e_minus;
```

CONTINUED

```
       delta_rho_caret_c_a_minus];

    % Unweighted least-squares calculation, computing position
        and receiver
    % clock offset
    x_caret_plus = x_caret_min + inv(H_G_e.'*H_G_e)*H_G_e.'*
       delta_z_min;
    r_caret_ea_e_plus = x_caret_plus(1:3);
    delta_rho_caret_c_a_plus = x_caret_plus(end);

    % Formulate predicted state vector for velocity
    x_caret_min = [v_caret_ea_e_minus;
       delta_rho_caret_dot_c_a_minus];

    % Unweighted least-squares calculation, computing velocity
        and receiver
    % clock drift
    x_caret_plus = x_caret_min + inv(H_G_e.'*H_G_e)*H_G_e.'*
       delta_z_dot_min;
    v_caret_ea_e_plus = x_caret_plus(1:3);
    delta_rho_caret_dot_c_a_plus = x_caret_plus(end);

    % Calculate difference compared to previous computation
    diff = norm([r_caret_ea_e_plus,v_caret_ea_e_plus]-[
       r_caret_ea_e_minus,v_caret_ea_e_minus]);

    % For next iteration
    r_caret_ea_e_minus = r_caret_ea_e_plus;
    v_caret_ea_e_minus = v_caret_ea_e_plus;
end

x_est = zeros(8,1);
x_est(1:3) = r_caret_ea_e_plus;
x_est(4:6) = v_caret_ea_e_plus;
x_est(7) = delta_rho_caret_c_a_plus;
x_est(8) = delta_rho_caret_dot_c_a_plus;

end
```

5.10 Gyro_Magnetometer_KF.m

```
function gyro_mag_heading_solution = Gyro_Magnetometer_KF(
   dr_measurement_data,times)
% Caculates gyro smoothed magnetic heading solution using
   Kalman filter
% Inputs:
```

```
%   dr_measurement_data                  Array of raw dead
   reckoning measurement data
%   times                                Array of time value
   for each epoch
%
% Outputs:
%   gyro_mag_heading_solution            Calculated heading
   solution for all epochs (deg)

Define_Constants;

dr_measurement = dr_measurement_data(:,2:end);

epoch_num = height(times);

% Setting initial covariance matrix
P_k_m_1_plus = [sigma_gyro^2,0
                0,1];

% First state being delta gyro heading, second state being
   gyro bias
x_caret_k_m_1_plus = zeros(2,1);

% Compute transition matrix
Phi_k_m_1 = [1,tau
             0,1];

% Compute system noise covariance matrix
Q_k_m_1 = [S_rg*tau+(S_bgd*tau^3)/3,0.5*S_bgd*tau^2
           0.5*S_bgd*tau^2,S_bgd*tau
           ];

% Compute measurement matrix
H_k = [-1,0];

% Compute measurement noise covariance
R_k = sigma_mh^2;

gyro_data = dr_measurement(:,5);
mag_data = dr_measurement(:,6)*deg_to_rad;

gyro_mag_heading_solution = zeros(epoch_num,1);
gyro_mag_heading_solution(1) = mag_data(1)*rad_to_deg;

% Set initial gyro heading as initial magnetic heading
psi_k_g = mag_data(1);
```

CONTINUED

```
for i = 2:epoch_num
    % Obtain magnetometer measurement
    psi_k_M = mag_data(i);

    % Propagate gyro heading using gyro measured angular rate
    psi_k_g = psi_k_g+gyro_data(i)*tau;

    % Propagate state estimates and error covariance matrix
    x_caret_k_minus = Phi_k_m_1*x_caret_k_m_1_plus;
    P_k_minus = Phi_k_m_1*P_k_m_1_plus*Phi_k_m_1'+Q_k_m_1;

    % Compute Kalman gain matrix
    K_k = P_k_minus*H_k'*inv(H_k*P_k_minus*H_k'+R_k);

    % Formulate measurement innovation vector
    delta_z_m_k = psi_k_M-psi_k_g+x_caret_k_minus(1);

    % Update state estimates and error covaraince matrix
    x_caret_k_m_1_plus = x_caret_k_minus + K_k*delta_z_m_k;
    P_k_m_1_plus = (eye(2) - K_k*H_k)*P_k_minus;

    % Apply correction to solution of current epoch
    correction = psi_k_g-x_caret_k_m_1_plus(1);
    gyro_mag_heading_solution(i) = correction*rad_to_deg;

end

% Plotting comparison between unsmoothed and smoothed heading
   solution
t = (1:epoch_num) * tau;
f = figure("Visible","off");
plot(t, mag_data*rad_to_deg, 'b', t, gyro_mag_heading_solution
   , 'r');
legend('Magnetometer Data', 'Filtered Heading');
xlabel('Time (s)');
ylabel('Heading (deg)');
title('Gyro Smoothed magnetic heading');
saveas(f,"Figures/Heading/smoothedComparison.png")

% Writing solution to csv file
outputTable = array2table(gyro_mag_heading_solution);
writetable(outputTable,"Solutions/
   gyro_smoothed_heading_solution.csv",'WriteVariableNames',0)

end
```

5.11 Initialise_GNSS_KF.m

```
function [x_est,P_matrix,valid_pseudo_range_indicies,
   valid_pseudo_range_rate_indicies] = Initialise_GNSS_KF(times
   ,pseudo_ranges,pseudo_range_rates,satellite_numbers,
   fix_outlier)
%Initialise_GNSS_KF - Initializes the GNSS EKF state estimates
    and error
%covariance matrix using one epoch least squares
%
% Inputs:
%   times                                 Array of time value
   for each epoch
%   pseudo_ranges                         Array of pseudo-range
   measurements for all epochs from all satellites
%   pseudo_range_rates                    Array of pseudo-range
   rate measurements for all epochs from all satellites
%   satellite_numbers                     Array of the number
   given to each of the satellites
%   number_of_satellites                  Total count of number
   of satellites giving the measurements
%   fix_outlier                           Flag indicating for
   fixing outliers
%
% Outputs:
%   x_est                                 Vector of initial
   state estimates:
%                                             Rows 1-3
              estimated ECEF user position (m)
%                                             Rows 4-6
              estimated ECEF user velocity (m/s)
%                                             Row 7
                 estimated receiver clock offset (m)
%                                             Row 8
                 estimated receiver clock drift (m/s)
%   P_matrix                              State estimation error
    covariance matrix
%   valid_pseudo_range_indicies           Matrix indicating
   whether the satellite pseudo range
%                                         is an outlier or not (
   number of epochs X number of satellites)
%   valid_pseudo_range_rate_indicies      Matrix indicating
   whether the satellite pseudo range rate
%                                         is an outlier or not (
   number of epochs X number of satellites)

Define_Constants;
```

CONTINUED

```
% Initial user position prediction
r_caret_ea_e_minus = [0;0;0];

% Initial user velocity prediction
v_caret_ea_e_minus = [0;0;0];

epoch_num = height(times);

valid_pseudo_range_indicies = ones(epoch_num,length(
   satellite_numbers));
valid_pseudo_range_rate_indicies = ones(epoch_num,length(
   satellite_numbers));

% First epoch outlier correction

original_r_caret_ea_e_minus = r_caret_ea_e_minus;
original_v_caret_ea_e_minus = v_caret_ea_e_minus;
valid_satellite_numbers = satellite_numbers;
epoch_pseudo_range_measurements = pseudo_ranges(1,:);
epoch_pseudo_range_rate_measurements = pseudo_range_rates(1,:)
   ;
[x_est_m,H_G_e,delta_z_min,delta_z_dot_min,r_caret_ea_e_minus,
   v_caret_ea_e_minus] = GNSS_Least_Squares(times(1),
   epoch_pseudo_range_measurements,
   epoch_pseudo_range_rate_measurements,satellite_numbers,
   r_caret_ea_e_minus,v_caret_ea_e_minus);


if fix_outlier
    [˜,maxJ] = Outlier_Detection(length(satellite_numbers),
       H_G_e,delta_z_min,sigma_rho);
    [˜,maxJDot] = Outlier_Detection(length(satellite_numbers),
       H_G_e,delta_z_dot_min,sigma_rho_dot);

    while maxJ > 0
        % Do the calculation again with outlier removed
        epoch_pseudo_range_measurements(maxJ) = [];
        epoch_pseudo_range_rate_measurements(maxJ) = [];
        valid_satellite_numbers(maxJ) = [];
        [x_est_m,H_G_e,delta_z_min,delta_z_dot_min,
           r_caret_ea_e_minus,v_caret_ea_e_minus] =
           GNSS_Least_Squares(times(1),
           epoch_pseudo_range_measurements,
           epoch_pseudo_range_rate_measurements,
           valid_satellite_numbers,original_r_caret_ea_e_minus,
           original_v_caret_ea_e_minus);
        [~,maxJ] = Outlier_Detection(length(
           valid_satellite_numbers),H_G_e,delta_z_min,sigma_rho
           );
        [~,maxJDot] = Outlier_Detection(length(
           valid_satellite_numbers),H_G_e,delta_z_dot_min,
           sigma_rho_dot);
    end

    % Set initial estimation with all outliers removed
    x_est = x_est_m;
    % Detect satellites that has outliers for following epochs
    for epoch=2:epoch_num
        epoch_pseudo_range_measurements = pseudo_ranges(epoch
           ,:);
        epoch_pseudo_range_rate_measurements =
           pseudo_range_rates(epoch,:);
        [~,H_G_e,delta_z_min,delta_z_dot_min,
           r_caret_ea_e_minus,v_caret_ea_e_minus] =
           GNSS_Least_Squares(times(epoch),
           epoch_pseudo_range_measurements,
           epoch_pseudo_range_rate_measurements,
           satellite_numbers,r_caret_ea_e_minus,
           v_caret_ea_e_minus);
        [epoch_valid_psuedo_range_indicies,~] =
           Outlier_Detection(length(satellite_numbers),H_G_e,
           delta_z_min,sigma_rho);
        [epoch_valid_psuedo_range_rate_indicies,~] =
           Outlier_Detection(length(satellite_numbers),H_G_e,
           delta_z_dot_min,sigma_rho_dot);
        valid_pseudo_range_indicies(epoch,:) =
           epoch_valid_psuedo_range_indicies;
        valid_pseudo_range_rate_indicies(epoch,:) =
           epoch_valid_psuedo_range_rate_indicies;
    end
else
    x_est = x_est_m;
end

P_matrix =  zeros(8);
P_matrix(1,1) = sigma_r^2;
P_matrix(2,2) = sigma_r^2;
P_matrix(3,3) = sigma_r^2;
P_matrix(4,4) = sigma_v^2;
P_matrix(5,5) = sigma_v^2;
P_matrix(6,6) = sigma_v^2;
```

CONTINUED

```
P_matrix(7,7) = sigma_co^2;
P_matrix(8,8) = sigma_cd^2;

% Ends
end
```

5.12 Load_Data.m

```
% Load in data from given csv files
pseudo_range_file = readtable("Pseudo_ranges.csv");
pseudo_range_data = table2array(pseudo_range_file);

pseudo_range_rate_file = readtable("Pseudo_range_rates.csv");
pseudo_range_rate_data = table2array(pseudo_range_rate_file);

deadReckoningFile = readtable("Dead_Reckoning.csv");
dr_measurement_data = table2array(deadReckoningFile);

% Array of time value for each epoch, being the first column
   of the files
times = pseudo_range_data(2:end,1);
```

5.13 Outlier_Data_Correction.m

```
function [corrected_data] = Outlier_Data_Correction(
   valid_indicies,data)
% Corrects inputting data according to an array of valid
   indicies, where
% index with 1 means the data at that index is valid and vice
   versa
% Inputs:
%   valid_indicies          Valid indicies array
%   data                    Data to be corrected
%
% Outputs:
%   corrected_data         Data corrected using the valid
   indicies array
corrected_data = (nonzeros(valid_indicies.*data))';
end
```

5.14 Outlier_Detection.m

```
function [valid_indicies, maxJ] = Outlier_Detection(
   number_of_satellites,H_G_e,delta_z_min,sigma)
% Detects outliers within given data
```

```
%
% Inputs:
%   number_of_satellites         Total count of number of
   satellites giving
%                                the measurements
%   H_G_e                        Measurement Matrix
%   delta_z_min                  Innovation vector
%   sigma                        Measurement error standard
   deviation
%
% Outputs:
%   valid_indicies               Array of (1,
   number_of_satellites)
%                                indicating whether the
   measurement of the
%                                satellite is an outlier (0)
   or inliner (1)
%   maxJ                         Index of satellite giving the
    highest
%                                normalized residual value
   larger than
%                                threshold

Define_Constants;

% Compute residuals vector
v = (H_G_e * inv(H_G_e.' * H_G_e) * H_G_e.' - eye(
   number_of_satellites)) * delta_z_min;

% Compute residuals covariance matrix
C_v = (eye(number_of_satellites) - H_G_e * inv(H_G_e.' * H_G_e
   ) * H_G_e.') * sigma^2;

% Setup
valid_indicies = zeros(1,number_of_satellites);
maxResidual = -inf;
maxJ = 0;

% Check for outliers
for j=1:number_of_satellites
    normalized_residual = norm(v(j))/sqrt(C_v(j,j));
    if normalized_residual > T
        if normalized_residual > maxResidual
            maxJ = j;
            maxResidual = normalized_residual;
        end
    else
        valid_indicies(j) = 1;
    end
end
end
```

5.15 Plot_Graph.m

```
function Plot_Graph(longitude,latitude,v_N,v_E,times,plotType,
   show)
% Function for plotting figures of vehicle position and
   velocities and
% saving them into Figures folder
% Inputs:
%   longitude             Array of longitude solutions of the
   vehicle
%   latitude              Array of longitude solutions of the
   vehicle
%   v_N                   Array of north velocity solutions of
    the vehicle
%   v_E                   Array of east velocity solutions of
   the vehicle
%   times                 Array of time value for each epoch
%   plotType              String indicating the type of the
   solution (GNSS/DR)
%   show                  Boolean indicating to show the plots
    or not

% Setting visbility of figures
visible = "off";
if show
    visible = "on";
end

f = figure("Visible",visible);
plot(longitude, latitude);
title("Position");
xlabel("Longitude (deg)");
ylabel("Latitude (deg)");
grid on;
saveas(f,"Figures/"+plotType+"/"+plotType+"_position.png")

f = figure("Visible",visible);
plot(times, v_N);
title("North Velocity");
xlabel("Time (s)");
```

```
ylabel("Velocity (m/s)");
grid on;
saveas(f,"Figures/"+plotType+"/"+plotType+"_velocity_north.png
   ")

f = figure("Visible",visible);
plot(times, v_E);
title("East Velocity");
xlabel("Time (s)");
ylabel("Velocity (m/s)");
grid on;
saveas(f,"Figures/"+plotType+"/"+plotType+"_velocity_east.png
   ")

f = figure("Visible",visible);
geoplot(latitude,longitude);
title("Geography Position");
grid on;
geobasemap streets
saveas(f,"Figures/"+plotType+"/"+plotType+"_geoplot.png")

end
```

5.16 Plot_heading.m

```
function Plot_heading(dr_gnss_longitude,dr_gnss_latitude,
   heading_solution,visible)
% Function for plotting heading of the vehicle using quiver
   plots and
% saving them into Figures folder
% Inputs:
%   dr_gnss_longitude              Array of integrated
   longitude solutions of the vehicle
%   dr_gnss_latitude               Array of integrated
   longitude solutions of the vehicle
%   heading_solution               Array of heading solutions
%   show                           Boolean indicating to show
   the plots or not

Define_Constants;
f = figure("Visible",visible);
epoch_num = height(heading_solution);
x = zeros(epoch_num,1);
y = zeros(epoch_num,1);

% Magnitude of each quiver vector
```

```
mag = 0.0000000001;

[y(1),x(1)] = pol2cart(heading_solution(1)*deg_to_rad,mag);

for i = 2: epoch_num
    % mag = sqrt((dr_gnss_latitude(i)-dr_gnss_latitude(i-1))
       ^2+(dr_gnss_longitude(i)-dr_gnss_longitude(i-1))^2);

    % Compute vector from given heading
    [yVec,xVec] = pol2cart(heading_solution(i)*deg_to_rad,mag)
       ;
    x(i) = xVec;
    y(i) = yVec;
end

quiver(dr_gnss_longitude,dr_gnss_latitude,x,y);
title("Heading");
xlabel("Longitude");
ylabel("Latitude");
grid on;
saveas(f,"Figures/Heading/heading.png")
end
```

5.17 main.m

```
clear;

Load_Data;

%% GNSS With Outlier Solution
gnss_outlier_solution = GNSS_KF(pseudo_range_data,
   pseudo_range_rate_data,times,false);
gnss_outlier_latitude = gnss_outlier_solution(:,2);
gnss_outlier_longitude = gnss_outlier_solution(:,3);
gnss_outlier_velocity_n = gnss_outlier_solution(:,5);
gnss_outlier_velocity_e = gnss_outlier_solution(:,6);

%% GNSS Outlier Removed Solution
gnss_solution = GNSS_KF(pseudo_range_data,
   pseudo_range_rate_data,times,true);
gnss_latitude = gnss_solution(:,2);
gnss_longitude = gnss_solution(:,3);
gnss_velocity_n = gnss_solution(:,5);
gnss_velocity_e = gnss_solution(:,6);

%% Gyro-smoothed Magnetic Heading solution
```

```
gyro_mag_heading_solution = Gyro_Magnetometer_KF(
   dr_measurement_data,times);
% gyro_mag_heading_solution = dr_measurement_data(:,7);

%% DR Solution
dr_solution = Dead_Reckoning(dr_measurement_data,gnss_solution
   ,gyro_mag_heading_solution,times);
dr_latitude = dr_solution(:,2);
dr_longitude = dr_solution(:,3);
dr_velocity_n = dr_solution(:,4);
dr_velocity_e = dr_solution(:,5);

%% DR-GNSS Solution
dr_gnss_solution = DR_GNSS(gnss_solution,dr_solution,times);
dr_gnss_latitude = dr_gnss_solution(:,2);
dr_gnss_longitude = dr_gnss_solution(:,3);
dr_gnss_velocity_n = dr_gnss_solution(:,4);
dr_gnss_velocity_e = dr_gnss_solution(:,5);

%% Final Output
final_output = [dr_gnss_solution,gyro_mag_heading_solution];
outputTable = array2table(final_output);
writetable(outputTable,"Solutions/final_solution.csv",'
   WriteVariableNames',0)


%% Plotting
% figure;
% plot(times, gnss_velocity_e,"xr-");
% hold on
% plot(times, dr_velocity_e,"*g-");
% hold on
% plot(times,dr_gnss_velocity_e,"b");
% grid on
% title("East Velocities")
% xlabel("Time (s)");
% ylabel("Velocity (m/s)");
% legend(["GNSS","DR","GR-DNSS"]);

% figure;
% plot(gnss_longitude,gnss_latitude,"b-o");
% hold on
% plot(gnss_outlier_longitude,gnss_outlier_latitude);
% title("Position");
% xlabel("Longitude (deg)");
% ylabel("Latitude (deg)");
```

CONTINUED

```
% grid on
% legend(["GNSS Outlier Removed","GNSS With Outlier"]);

Plot_Graph(gnss_longitude,gnss_latitude,gnss_velocity_n,
   gnss_velocity_e,times,"GNSS",false)

Plot_Graph(gnss_outlier_longitude,gnss_outlier_latitude,
   gnss_outlier_velocity_n,gnss_outlier_velocity_e,times,"GNSS-
   Outlier",false)

Plot_Graph(dr_longitude,dr_latitude,dr_velocity_n,
   dr_velocity_e,times,"DR",false)

Plot_Graph(dr_gnss_longitude,dr_gnss_latitude,
   dr_gnss_velocity_n,dr_gnss_velocity_e,times,"DR-GNSS",false)

Plot_heading(dr_gnss_longitude,dr_gnss_latitude,
   gyro_mag_heading_solution,false);
```

END OF COURSEWORK