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

Yuansheng Zhang*, Joseph Rowell*& Vanessa Igodifo*

10/02/2022

Kalman Filter

To initialise the Kalman filter state vector estimate, the estimated position and estimated velocity were used to initialise the error covariance matrix and initial state uncertainties, being $10m$ and $0.05ms^{-1}$ for pseudo range measurements and pseudo rate range measurements respectively. [1][2]

The steps to build a Kalman filter [2] are as follows:

Step 1: Use deterministic system model to calculate transition matrix $\mathbf{\Phi_{k-1}}$.
The transition matrix defines how the measurement state vector $\hat{\mathbf{x}}$ changes with time as a function of the dynamics of the system.

Step 2: Use stochastic system model to calculate system noise covariance $\mathbf{Q_{k-1}}$.
We add this additional term on to represent that the uncertainty in our state estimate has increased from the previous estimate to the current one, since we have moved to an unknown location. In our case, we know that the position and velocity of the lawnmower has changed, but do not know by how much.

Step 3: Propagate state estimates: $\hat{\mathbf{x}}_{\mathbf{k}}^{-} = \mathbf{\Phi_{k-1}}\hat{\mathbf{x}}_{\mathbf{k-1}}^{+}$
Using the transition matrix defined previously, we can calculate the predicted position, velocity, clock offset, and clock drift from the estimated values of these parameters, which are initialised before the first epoch as matrices and vectors of zeros of the appropriate dimensions (one measurement per satellite per time step).

Step 4: Calculate error covariance: $\mathbf{P}_{\mathbf{k}}^{-} = \mathbf{\Phi_{k-1}}\mathbf{P}_{\mathbf{k-1}}^{+}\mathbf{\Phi}_{\mathbf{k-1}}^{\mathbf{T}} + \mathbf{Q_{k-1}}$
This is a matrix of both the independent errors that affect individual measurements and contribute to the off diagonals of the error covariance matrix, and the correlated errors that affect multiple measurements and contribute to all elements of the error covariance matrix. In our case, we assume that the GNSS measurements have already been corrected for the ionosphere, troposphere, and satellite clock errors, but initialise a series of GNSS error specifications, assumed under typical conditions.

Step 5: Calculate Measurement matrix $\mathbf{H_k}$.
This matrix comprises the partial derivatives of the measurement function with respect to the states (position velocity, clock drift, clock offset). It is calculated using the predicted states from the previous steps.

Step 6: Calculate measurement noise covariance $\mathbf{R}_k$.
For sequential least squares problems, we are dealing with data where measurements become available

[0]*These authors contributed equally to this work.

at different times, i.e. the data is collected from each satellite at each time step, so is processed accordingly. This allows us to obtain intermediate solutions for the state estimates and monitor our progress. The measurement noise covariance models errors that are uncorrelated across successive epochs. For a given epoch, k, the measurement noise covariance $\mathbf{R_k}$ allows us to model the errors common to the measurements made at each epoch.

Step 7: Calculate Kalman gain matrix: $\mathbf{K_k} = \mathbf{P_k^-}\mathbf{H_k^T}(\mathbf{R_k} + \mathbf{H_k}\mathbf{P_k^-}\mathbf{H_k^T})^{-1})$
For multi-epoch least squares, the gain matrix can be used to better formulate the problem in a recursive manner. A Kalman Gain matrix is generated at every epoch to both transform from the measurement domain to the state domain, and weight each measurement in the state vector estimate.

Step 8: Formulate measurements $\tilde{\mathbf{z}}_\mathbf{k}$: $\tilde{\mathbf{z}}_\mathbf{k} = \mathbf{H_k}\mathbf{x_k} + \mathbf{w_{m,k}}$, where $\tilde{\mathbf{z}}_\mathbf{k}$ are the measurements, $\mathbf{x_k}$ are true states, and $\mathbf{w_{m,k}}$ is measurement noise.

Step 9: Measurement update of state estimates, using the measurement innovation and Kalman gain matrix to update the state vector estimate : $\hat{\mathbf{x}}_\mathbf{k}^+ = \hat{\mathbf{x}}_\mathbf{k}^- + \mathbf{K_k}(\tilde{\mathbf{z}_k} - \mathbf{H_k}\hat{\mathbf{x}}_\mathbf{k}^-)$

Step 10: Measurement update of error covariance: $\mathbf{P_k^+} = (\mathbf{I} - \mathbf{K_k}\mathbf{H_k})\mathbf{P_k^-}$, where $\mathbf{P_k^+}$ is the states estimates error covariance, $\mathbf{K_k}$ is the Kalman Gain matrix, $\mathbf{H_k}$ is the measurement matrix, and $\mathbf{P_k^-}$ is the predicted states error covariance.

Figure 1 shows the geodesic position plot of the lawnmower calculated solely using GNSS data from the 8 satellites. Figure 2 shows the 2D velocity of the lawnmover at each point in time - again calculated from GNSS satellite data. As can be seen at the beginning of the plot, there are errors in the position, which likely arose due to the naive initialisation of all states as 0.

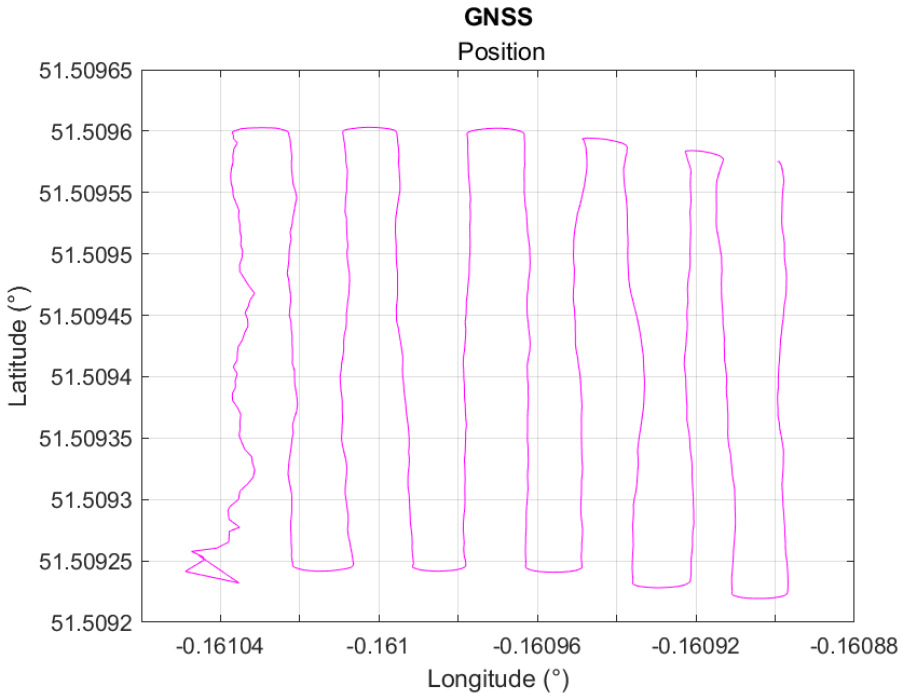

Figure 1: GNSS Position Plot

Figure 2 shows the 2D velocity of the lawnmower from GNSS data collected by the 8 satellites. Velocity is the first order derivative of position with respect to time, which can be seen by comparing this with the plot in Figure 1. The plot of velocity north is highly correlated to the change in latitude, as

latitude is the angular distance measured from north to south. A positive velocity shows travel in the southbound direction, which can be seen as an increase in latitude as the lawnmower traverses from the left hand side of the space to the right. The velocity east stays fairly constant, and corresponds to the longitude measurements (where longitude is the angular distance measured east to west). It also remains around $0m/s$, but increases to a slightly positive value when the velocity north gets to $0m/s$. This is typical of the scenario we are observing and indicates the lawnmower has reached the top/bottom of the lawn, and moves east slightly before changing direction and traversing the lawn in the opposite direction.

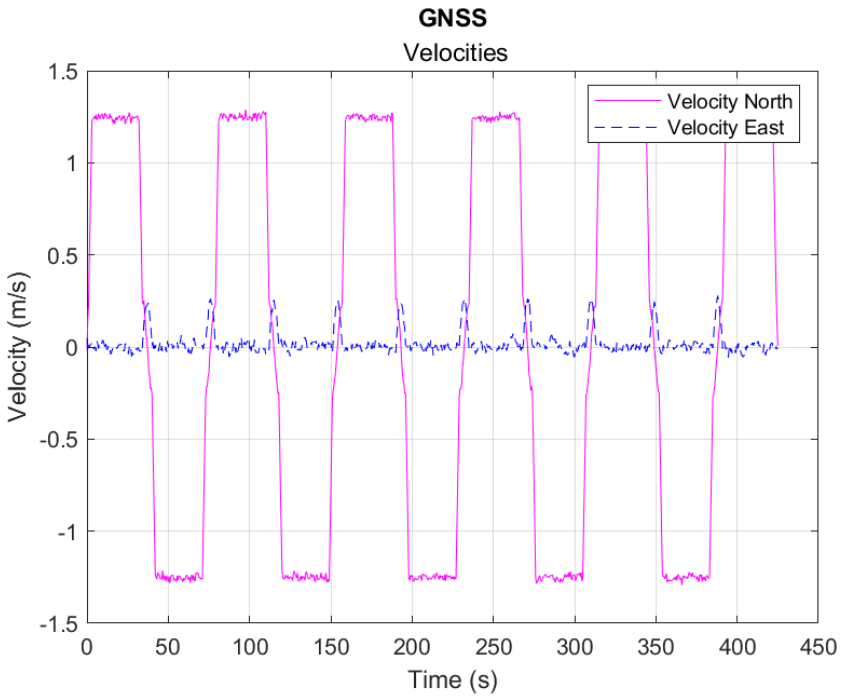

Figure 2: GNSS Velocity Plot

Figure 3 shows the quiver plot of the GNSS results, which indicated the velocity magnitude and direction superposed on a plot of the longitude and latitude of the lawnmower.

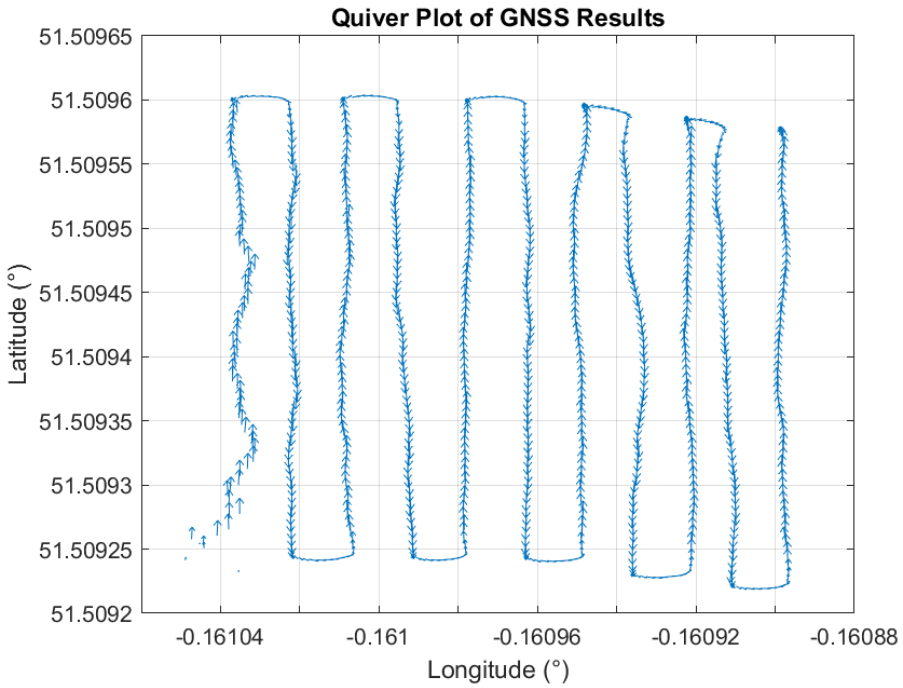

Figure 3: GNSS Quiver Plot

Figure 4 shows a GeoPlot of the lawnmower's 2D position, which overlays the position on a map of the lawnmower's actual location.

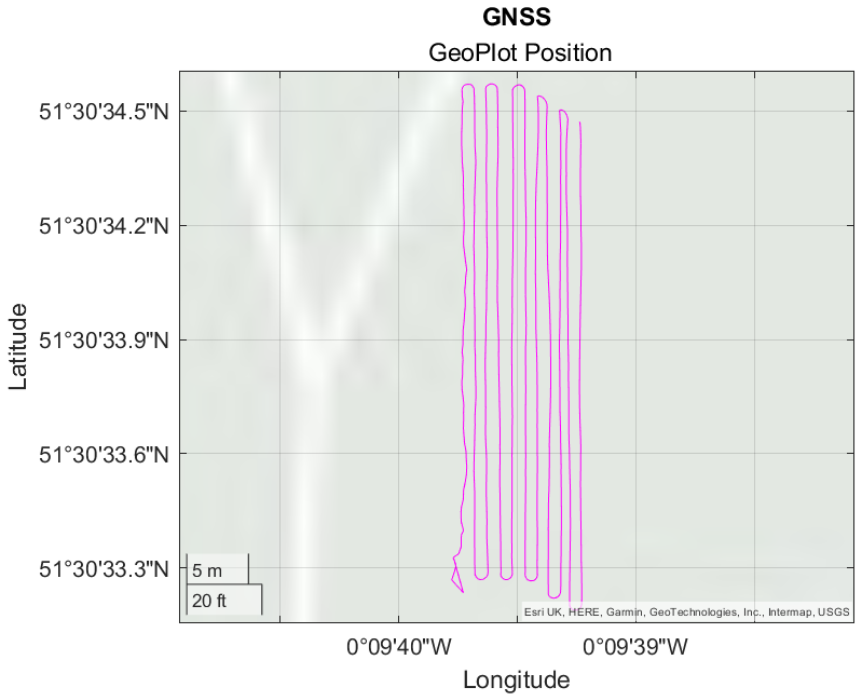

Figure 4: GNSS GeoPlot

Dead Reckoning

Dead Reckoning allows for motion measurement within a body axes by utilising initial orientation (heading for 2D) and wheel speed odometry of the lawnmower. To calculate the dead Reckoning solution, the method is as follows:

- Import data from .csv
- Input known initial longitude, latitude and alititude from GNSS solution.
- Initialise variables from dead reckoning data csv file
- Calculate average wheel velocity

$$averagewheelvelocity = \frac{1}{2} * wheelspeed3 + wheelspeed4$$

- Calculate longitudes and latitudes
- Calculate damped instant velocity
- Export struct results to .csv and plot

Dead reckoning position is given as the integral of the velocity.

$$\begin{pmatrix} x^p_{pb}(t) \\ y^p_{pb}(t) \end{pmatrix} = \begin{pmatrix} x^p_{pb}(t_0) \\ y^p_{pb}(t_0) \end{pmatrix} + \int_{t_0}^{t} \begin{pmatrix} v^p_{pb,x}(t') \\ v^p_{pb,y}(t') \end{pmatrix} dt'$$

Dead reckoning positioning in 2D was chosen as the lawnmower essentially moves in 2D, and can be calculated as follows, using the heading solution ψ_{pb} to transform the velocity measurements from the body frame to the environment frame axes.

$$\begin{pmatrix} v^p_{pb,x}(t') \\ v^p_{pb,y}(t') \end{pmatrix} = \begin{pmatrix} cos(\psi_{pb}(t')) & -sin(\psi_{pb}(t')) \\ sin(\psi_{pb}(t')) & cos(\psi_{pb}(t')) \end{pmatrix} \begin{pmatrix} v^b_{pb,x}(t') \\ v^b_{pb,y}(t') \end{pmatrix}$$

Although dead reckoning solution has sources of error as shown in table below [3]

Error Source	Impact on Position Error
Distance/velocity measurement noise	Random walk error; standard deviation grows as $\sqrt{time}$
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 $\sqrt{time}$
Heading measurement bias	Error perpendicular to direction travelled; grows in proportion to distance travelled

Figure 5 shows the dead reckoning solution for the 2D position of the lawnmower. As can be seen when compared to Figure 1, the estimated latitude and longitude of the lawnmower is the fairly similar, with slight differences in the latitude especially, which may be due to the fact that the position error grows with time in the dead reckoning solution. Also, GNSS vertical positioning errors are nearly twice the size of horizontal errors due to signals only coming from above, rather than in all directions. [1]

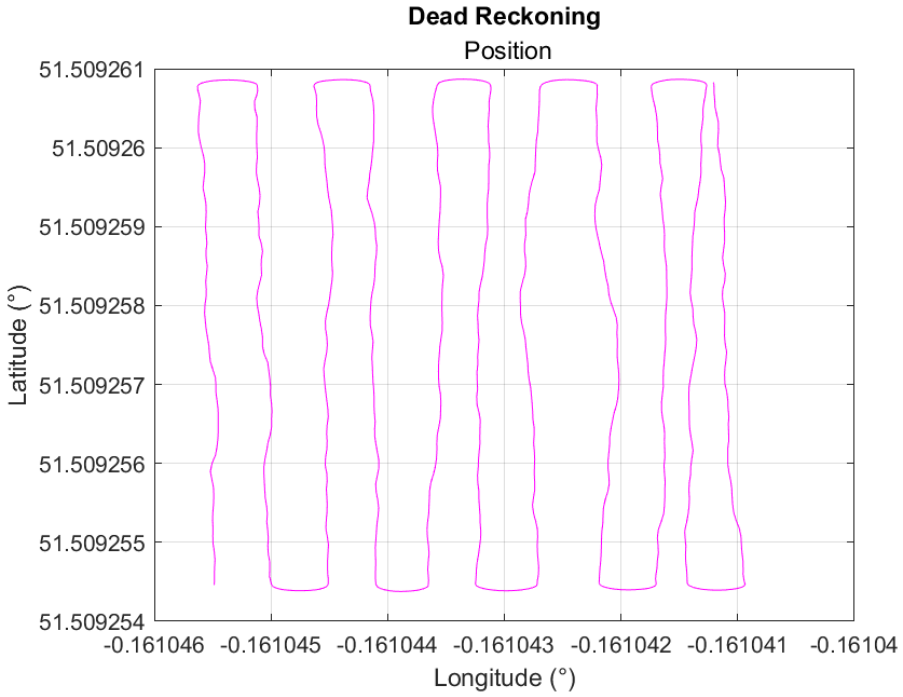

Figure 5: Dead Reckoning Position Plot

Figure 6 shows the dead reckoning solution for the 2D velocity of the lawnmower. Measurements in this method are made in the sensor body frame [3], i.e. in the frame of reference of the lawnmower. This explains why the north velocity of the lawnmower is always positive, whereas before with the GNSS solution, we saw both positive and negative velocities being recorded. GNSS measurements use ECEF (Cartesian Earth-centred Earth-Fixed) referenced velocity [2]. From the frame of reference of the lawnmower, the wheels only spin forwards, therefore, the velocity of the lawnmower is in the forwards (or positive) direction only.

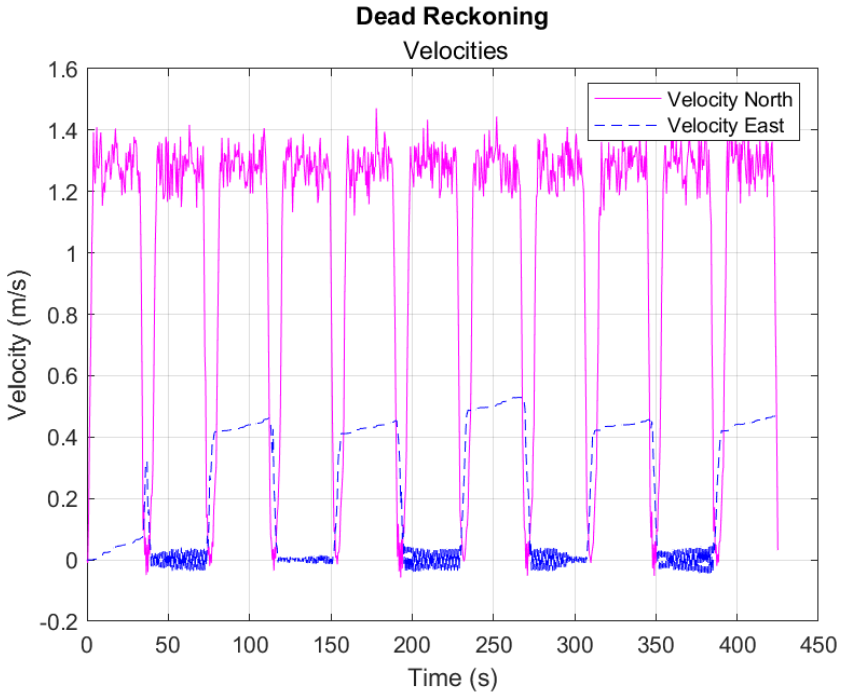

Figure 6: Dead Reckoning Velocity Plot

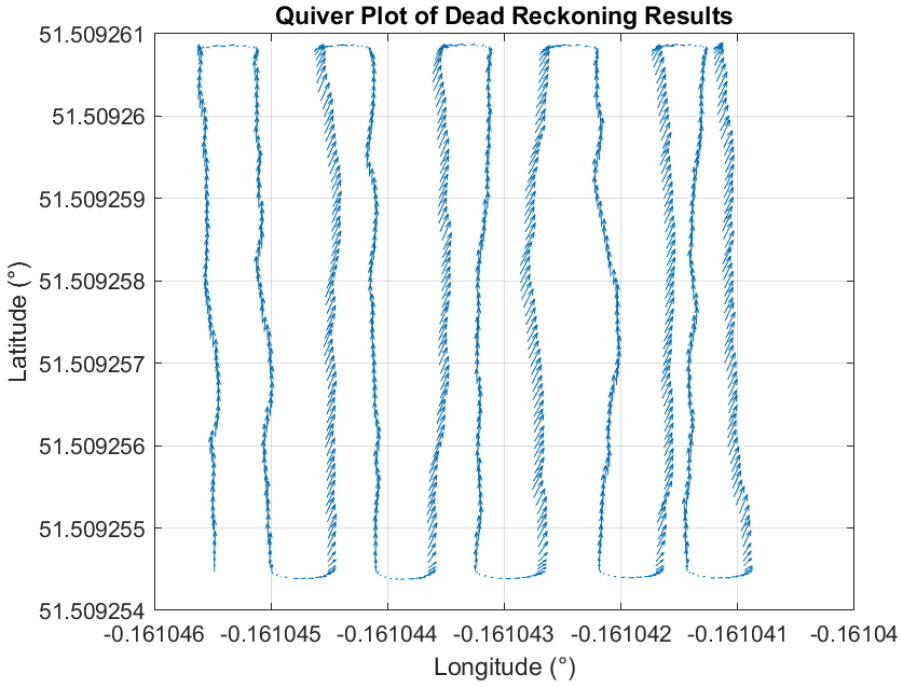

Figure 7: Dead Reckoning Quiver Plot

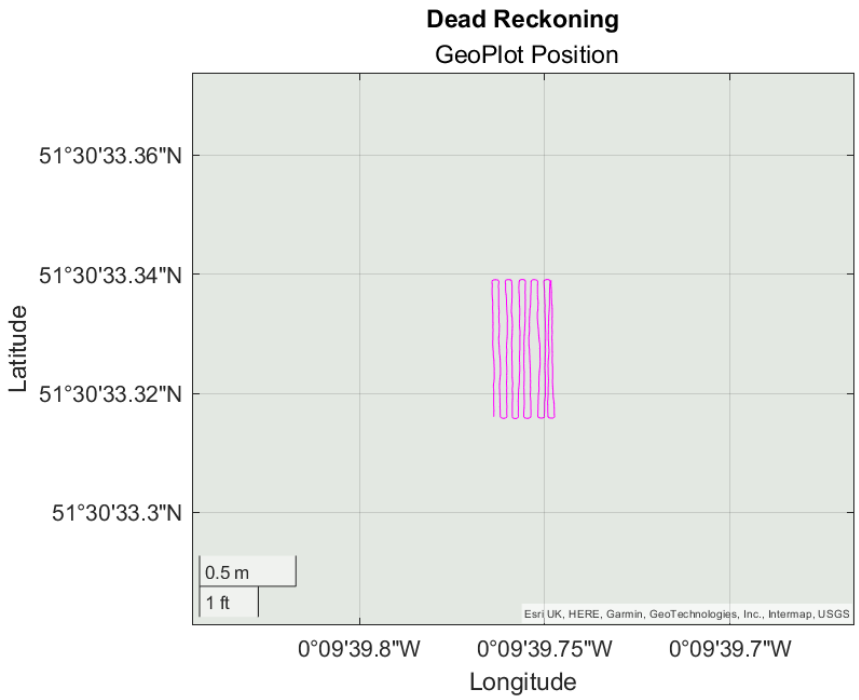

Figure 8: Dead Reckoning GeoPlot

The Geoplot in Figure 8 shows the autonomous lawnmower in Hyde Park. Whilst the Quiver Plot shows both position, velocity and heading at that given position.

Integrated Kalman Filter, Dead Reckoning Solution

Steps taken to integrate the Kalman Filter solutions and the Dead reckoning sensing solutions are as follows: **NED Step 1:** Calculate the Transition Matrix [4]

$$\phi_{\mathbf{k-1}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \frac{\tau_s}{R_N+h_b} & 0 & 0 & 1 & 0 & 0 \\ 0 & \frac{\tau_s}{(R_E+h_b)cos(L_b)} & 0 & 0 & 1 & 0 \\ 0 & 0 & -\tau_s & 0 & 0 & 1 \end{pmatrix}$$

Where τ_s = time interval.
R_N =meridian radius of curvature.
R_E =transverse radius of curvature.

$$R_N = \frac{R_0(1-e^2)}{(1-e^2 sin^2 L_b)^{\frac{3}{2}}}$$

$$R_E = \frac{R_0}{\sqrt{1-e^2 sin^2 L_b}}$$

Step 2: Calculate System Noise Covariance. [4] Where PSD = Power Spectral Density.

$$\mathbf{Q_{k-1}} = \begin{pmatrix} PSD\tau_s & 0 & 0 & \frac{1}{2}\frac{PSD\tau_s^2}{R_N+h_b} & 0 & 0 \\ 0 & PSD\tau_s & 0 & 0 & \frac{1}{2}\frac{PSD\tau_s^2}{(R_E+h_b)cos(L_b)} & 0 \\ 0 & 0 & PSD\tau_s & 0 & 0 & -\frac{1}{2}PSD\tau_s^2 \\ \frac{1}{2}\frac{PSD\tau_s^2}{R_N+h_b} & 0 & 0 & \frac{1}{3}\frac{PSD\tau_s^3}{(R_N+h_b)^2} & 0 & 0 \\ 0 & \frac{1}{2}\frac{PSD\tau_s^2}{(R_E+h_b)cos(L_b)} & 0 & 0 & \frac{1}{3}\frac{PSD\tau_s^3}{(R_E+h_b)^2 cos^2 L_b} & 0 \\ 0 & 0 & -\frac{1}{2}PSD\tau_s^2 & 0 & 0 & \frac{1}{3}PSD\tau_s^3 \end{pmatrix}$$

NED Step 5: Calculate Measurement Matrix [4]

$$\mathbf{H_k} = \begin{pmatrix} \mathbf{0_3} & \mathbf{-I_3} \\ \mathbf{-I_3} & \mathbf{0_3} \end{pmatrix}$$

NED Step 6: Measurement Noise Covariance [4]

$$\mathbf{R_k} = \begin{pmatrix} \frac{\sigma_{Gr}^2}{(R_N=h_b)^2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{\sigma_{Gr}^2}{(R_E+h_b)^2)cos^2 L_b} & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_{Gr}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_{Gv}^2 & 0 & 0 \\ 0 & 0 & 0 & & \sigma_{Gv}^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_{Gv}^2 \end{pmatrix}$$

Where σ_{Gr}^2 is the variance per axis of GNSS position noise (m^2).
NED Step 8: Measurement Innovation [4]

$$\delta \mathbf{z} = \tilde{\mathbf{z}}_\mathbf{k} - \mathbf{H_k}\hat{\mathbf{x}}_\mathbf{k}^-$$

Where

$$\delta \tilde{z}_k = \begin{pmatrix} L_k^G - L_k^D \\ \lambda_k^G - \lambda_k^D \\ v_{N,k}^G - v_{N,k}^D \\ v_{E,k}^G - v_{E,k}^D \end{pmatrix} - \mathbf{H}_k\hat{\mathbf{x}}_k = \begin{pmatrix} L_k^G - L_k^D + \delta L_k^- \\ \lambda_k^G - \lambda_k^G + \delta\lambda_k^- \\ v_{N,k}^G - v_{N,k}^D + \delta v_{N,k}^- \\ v_{E,k}^G - v_{E,k}^D + \delta v_{E,k}^- \end{pmatrix}$$

Where G denotes GNSS solution and D indicates the Dead Reckoning Solution.
Step 9 Update the state estimates using

$$\hat{\mathbf{x}}_\mathbf{k}^+ = \hat{\mathbf{x}}_\mathbf{k}^- + \mathbf{K}_\mathbf{k}^-\mathbf{}$$

Step 10 Update the error covariance matrix using:

$$P_k^+ = (I - K_k H_k)P_k^-$$

Finally, the kalman filter estimates were used to correct the dead reckoning the solution at each epoch, k, using

$$L_k^C = L_k^D - \delta L_k^+$$
$$\lambda_k^C = \lambda_k^D - \delta\lambda_k^+$$
$$v_{N,k}^C = v_{N,k}^D - \delta v_{N,k}^+$$
$$v_{E,k}^C = v_{E,k}^D - \delta v_{E,k}^+$$

[5]

Figure 9 shows the position plot for the integrated solution of both GNSS and dead reckoning. This method involved the combination of both positioning methods to obtain a more accurate position solution.

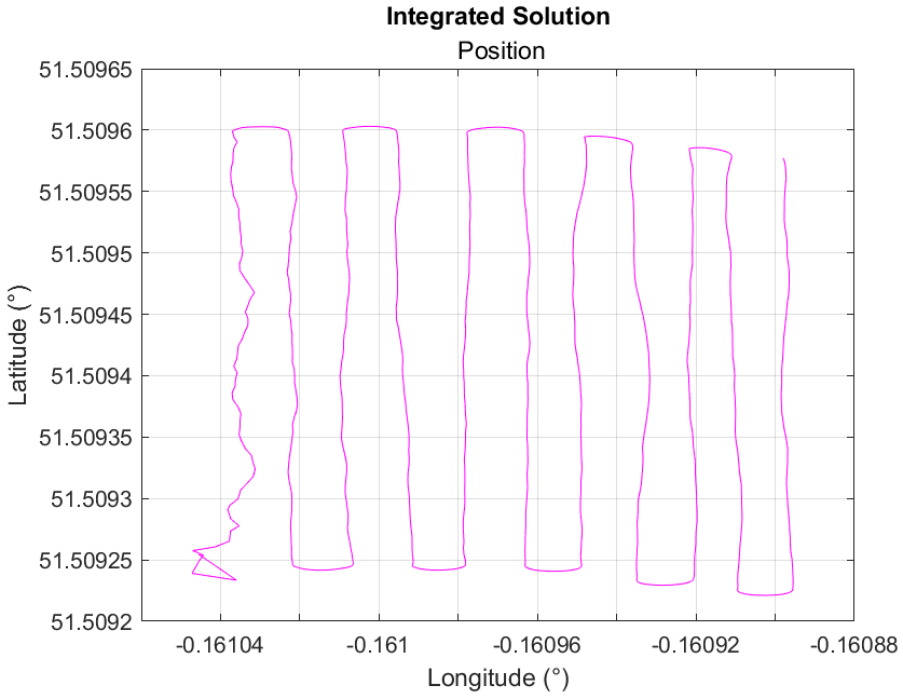

Figure 9: Integrated Solution Position Plot

Figure 10 shows the velocity solution for the combination of both GNSS and dead reckoning solutions. With more data comes the ability to detect errors and outliers easier. The velocity solution shows much less variation (after the primary initialisation), where the solutions of both technologies have their own advantages to ultimately identify a better solution.

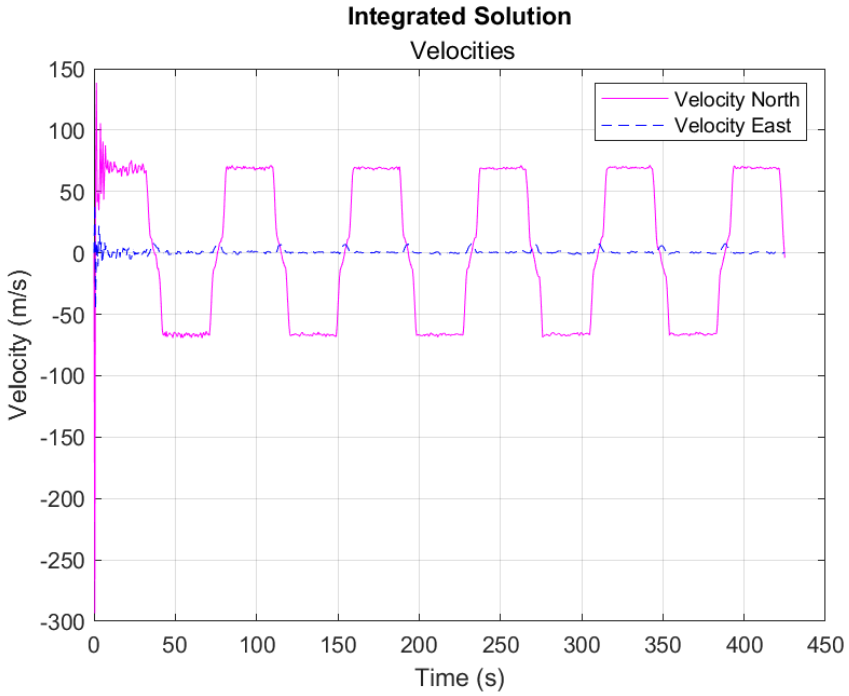

Figure 10: Integrated Solution Velocity Plot

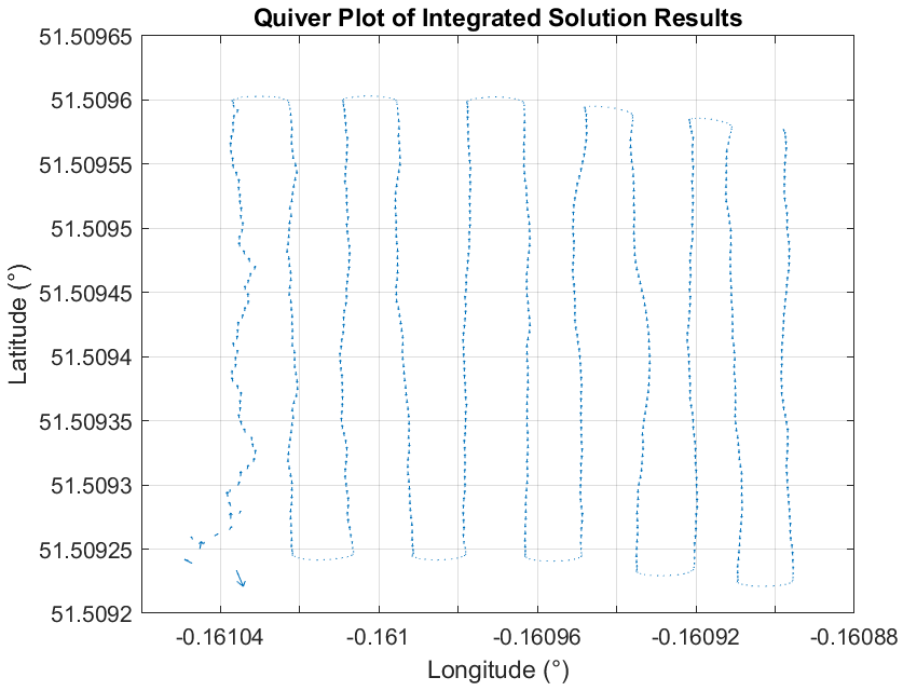

Figure 11: Integrated Solution Quiver Plot

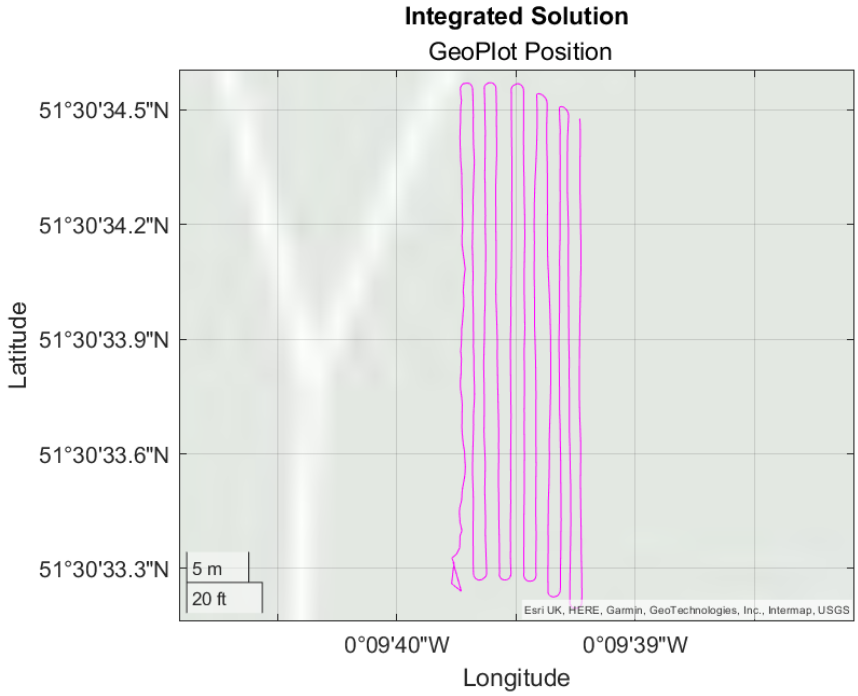

Figure 12: Integrated Solution GeoPlot

The GNSS solution allows us to more accurately calibrate the systematic errors of the sensors used to to calculate the dead reckoning solution. It also prevents the position solution error of dead reckoning from growing with time, as it has a tendency to do. Conversely, dead reckoning also improves the robustness of GNSS by boosting continuity and increasing the bandwidth of the position solution to smooth out noise. [2][4].

References

[1] D. P. D. Groves, "2A: GNSS Errors and Advanced Techniques," 2022.

[2] D. P. D. Groves, "2B: The Kalman Filter and its uses for GNSS," 2022.

[3] D. P. D. Groves, "3A: Motion Sensing, Dead Reckoning and Inertial Navigation," 2022.

[4] D. P. D. Groves, "3B: Multisensor Integrated Navigation," 2022.

[5] D. P. D. Groves, "Workshop 3: Multisensor navigation," 2022.

Appendix

Listings


```
function GNSSResults = GNSSSolution(Pseudo_ranges_data, Pseudo_range_rates_data)
% Compute user geodetic position and velocity with GNSS measurement
%
% Inputs:
%   Pseudo_ranges_data      measured psuedo range data matrix
%   Pseudo_range_Rates_data     measured psuedo range rate data matrix
%
% Outputs:
%   GNSSResults     GNSS results struct with fields:
%       timeStamp       Time in seconds     this must match the data files.
%       latitude        Geodetic latitude in degrees
%       longitude       Geodetic longitude in degrees
%       velNorth        North velocity in metres per second
%       velEast         East velocity in metres per second
%       heading         Heading in degrees

% Everything to 15 decimal places using format long
format long

%% Import data

%RawData.tableRanges = (readmatrix("data/Pseudo_ranges.csv"));
%RawData.tableRates = (readmatrix("data/Pseudo_range_rates.csv"));
RawData.tableRanges = Pseudo_ranges_data;
RawData.tableRates = Pseudo_range_rates_data;
RawData.timeStamp = RawData.tableRanges(2:end, 1);
RawData.satNos = RawData.tableRanges(1, 2:end);
RawData.pseudoRanges = RawData.tableRanges(2:end, 2:end);
RawData.pseudoRangeRates = RawData.tableRates(2:end, 2:end);


%% Initialisation

Define_Constants;

% Get satellite number
nSat = length(RawData.satNos);

% Compute inital error covariance matrix
P_k = [sdOfs^2 * ones(3,1);
    sdDft^2 * ones(3,1);
    sdOfs^2;
    sdDft^2];
P_k = diag(P_k);

% initial state
X = zeros(8,1); %[x y z v_x v_y v_z rClockOffset rClockDrift]


%% First epoch
xNorm = 0;
diff = 1;
threshold = 1e-5;

% Compute first epoch of least squares solution
t=0;
[X, H_k, dZ] = getGNSSLeastSquare(RawData, X, nSat, omega_ie, Omega_ie, c, diff, xNorm
    , threshold, t);

```

```
% Detect and remove outliers
ifOutlier = true;

while ifOutlier == true
    [RawData, nSat, ifOutlier] = detectOutliers (RawData, nSat, ifOutlier, H_k, dZ);
    % Compute least squares solution with outliers removed
    [X, H_k, dZ] = getGNSSLeastSquare(RawData, X, nSat, omega_ie, Omega_ie, c, diff,
    xNorm, threshold, t);
end

% Convert ECEF to NED
[L_b,lambda_b,h,v_eb_n] = pv_ECEF_to_NED(X(1:3), X(4:6));

% Record results
GNSSResults.timeStamp = 0 * ones(1,1);
GNSSResults.latitude = L_b * rad_to_deg * ones(1,1);
GNSSResults.longitude = lambda_b * rad_to_deg * ones(1,1);
GNSSResults.velNorth = v_eb_n(1) * ones(1,1);
GNSSResults.velEast = v_eb_n(2) * ones(1,1);
GNSSResults.heading = atan2 (v_eb_n(2), v_eb_n(1)) * rad_to_deg * ones(1,1);
GNSSResults.altitude = h;

%% Multi-epoch with Kalman Filter
for t = RawData.timeStamp(2):tau :RawData.timeStamp(end)
    % Step 1: Compute the transition matrix
    phi = getTransMat (tau);

    % Step 2: Compute the system noise covariance matrix
    Q = getSysNoiseCovMat (S_a, tau, S_cf, S_cphi);

    % Step 3: Use the transition matrix to propagate the state estimates
    X = phi * X;

    % Step 4: Propagate the error covariance matrix
    P_k = phi * P_k * phi' + Q;

    % Predict the ranges from the approximate user position to each satellite
    [r_aj, C_e, r_ej, v_ej] = predictRanges(RawData, X, nSat, omega_ie, c, t);

    % Compute the line-of-sight unit vector from the approximate user position to each
     satellite
    U_aj = getLineOfSightVec (nSat, C_e, r_ej, r_aj, X);

    % Step 5: Compute the measurement matrix
    H_k = getMeaMat (nSat, X, U_aj);

    % Step 6: Compute the measurement noise covariance matrix
    R_k = [eye(nSat) * sdPseuRanMea^2, zeros(nSat);
        zeros(nSat), eye(nSat) * sdPseuRanRateMea^2];

    % Step 7: Compute the Kalman gain matrix
    K_k = P_k * H_k' / (H_k * P_k * H_k' + R_k);

    % Step 8: Formulate the measurement innovation vector, dZ
    dZ = getInnovVec (RawData,r_aj, X, Omega_ie, U_aj, C_e, v_ej, r_ej, nSat, t);

    % Step 9: Update the state estimates
    X = X + K_k * dZ;

    % Step 10: Update the error covariance matrix
    P_k = (eye(8) - K_k * H_k) * P_k;

    % Convert ECEF to NED
    [L_b,lambda_b,h,v_eb_n] = pv_ECEF_to_NED(X(1:3), X(4:6));

    % Record results
    GNSSResults.timeStamp (end+1) = t;
    GNSSResults.latitude (end+1) = L_b * rad_to_deg;
    GNSSResults.longitude (end+1) = lambda_b * rad_to_deg;
    GNSSResults.velNorth (end+1) = v_eb_n(1);
    GNSSResults.velEast (end+1) = v_eb_n(2);
```

```
    GNSSResults.heading (end+1) = atan2 (v_eb_n(2), v_eb_n(1)) * rad_to_deg;
    GNSSResults.altitude (end+1) = h;
end

% Convert each field to column vector
fns = fieldnames(GNSSResults);
for iField = 1:numel (fns)
    GNSSResults.(fns{iField}) = GNSSResults.(fns{iField})';
end

writetable(struct2table(GNSSResults), 'Results/GNSSResults.xlsx')

end
```

Listing 1: GNSS Kalman Filter Solution

```
function integratedResults = IntegratedSolution(deadReckoningData, ...
    deadReckoningResults, Pseudo_range_rates, Pseudo_ranges, GNSSResults)
%INPUTS:
%   deadReckoningData:
%       Dead Reckoning data imported csv (matrix)
%   deadReckoningResults:
%       Results from deadReckoningSolution (struct)
%   GNSSResults:
%       Results from GNSS (struct)
%   tau:
%       time interval (int)

%OUTPUTS:
%   integratedResults: Integrated solution of dead reckoning and GNSS
%       struct with fields:
%          times        Time in seconds     this must match the data files.
%          latitudes    Geodetic latitude in degrees
%          longitudes   Geodetic longitude in degrees
%          vel_N        North velocity in metres per second
%          vel_E        East velocity in metres per second
%          heading      Heading in degrees

    %Import useful constants
    Define_Constants;

    times        = deadReckoningResults.times;
    drlatitudes  = deadReckoningResults.latitudes;
    drlongitudes = deadReckoningResults.longitudes;
    drvel_N      = deadReckoningResults.vel_N;
    drvel_E      = deadReckoningResults.vel_E;
    drheading    = deadReckoningResults.heading; %TODO convert to radians
    gnsstimes      = GNSSResults.timeStamp;
    gnsslatitudes  = GNSSResults.latitude;
    gnsslongitudes = GNSSResults.longitude;
    gnssvel_N      = GNSSResults.velNorth;
    gnssvel_E      = GNSSResults.velEast;
    gnssheading    = GNSSResults.heading;
    gnssaltitude   = GNSSResults.altitude;

    drgyro = deadReckoningData(:,7) * deg_to_rad;

    nSat = length(Pseudo_range_rates(1, 2:end));

%Calculate diffference between systems , integrated kalman filter solution
%% TODO initialise x, P, h???
    [RN,RE] = Radii_of_curvature(drlatitudes(1));
    epochs = size(times,1);
    x = zeros(6,1);
    h = gnssaltitude(1);
    % Compute inital error covariance matrix
%     P = [sdDft^2 * ones(3,1);
%          sdOfs^2 * ones(3,1)];
% Assume initial velocity uncertainty 0.02, posiiton undecterinay 1
    P =  [0.02^2 * ones(3,1); 1; 1/(RN + h(1))^2; 1/((RE + h(1))^2 * cos(drlatitudes
    (1)))];
```

```
    P = diag(P);

    for i = 1:epochs
        %NED step 1: Transition matrix psi
        psi = eye(6);
        [RN,RE] = Radii_of_curvature(drlatitudes(i));
        psi(4,1) = tau / (RN + h);
        psi(5,2) = tau / ((RE + h) * cos(drlatitudes(i)));
        psi(6,3) = tau;
        %NED step 2: System Noise covariance matrix Q
        %PSD IN Define Constants? tau, h too?
        Q = [PSD * tau, 0, 0, 0.5 * (PSD * tau^2)/(RN +h), 0, 0;
            0, PSD * tau, 0, 0, 0.5 * (PSD * tau^2)/((RE + h)*cos(drlatitudes(i))), 0;
            0, 0, PSD * tau, 0, 0, -0.5 * PSD * tau^2;
            0.5 * (PSD * tau^2)/(RN +h), 0, 0, (PSD * tau^3)/(3*(RN +h)^2), 0, 0;
            0, 0.5 * (PSD * tau^2)/((RE + h)*cos(drlatitudes(i))), 0, 0, (PSD * tau^3)
    /(3*(cos(drlatitudes(i)))^2*(RN +h)^2), 0;
            0, 0, -0.5 * PSD * tau^2, 0, 0, (PSD*tau^3)/3];
        %NED step 3 & 4: Propagate State and Covariance x, P
        x = psi * x;
        P = psi * P * psi' + Q;
        %NED step 5: Calculate measurement matrix H
        %H = getMeaMat (nSat, x, 1);
        H = [zeros(3,3), -eye(3);
            -eye(3), zeros(3,3)];
        %% TODONED step 6: Measurement Noise Covariance R
%         R = [eye(3) * sdPseuRanMea^2, zeros(3);
%             zeros(3), eye(3) * sdPseuRanRateMea^2]; %Measurement noise comprises only
     the GNSS errors

        sdGr = 1;
        sdGv = 0.05;
        R = [ sdGr^2 /(RN + gnssaltitude(i))^2; sdGr^2/(RE+gnssaltitude(i))^2/(cos(
    drlatitudes(i)))^2; sdGr^2; sdGv*ones(3,1)];
        R = diag(R);

        %Step 7: Calculate Kalman Gain Matrix
        K = P * H' / (H * P * H' + R);

        %% TODO Intermediate Step, find z (difference between DR & GNSSResults)
        z = [gnsslatitudes(i) - drlatitudes(i);
             gnsslongitudes(i) - drlongitudes(i);
             gnssaltitude(i);
             gnssvel_N(i) - drvel_N(i);
             gnssvel_E(i) - drvel_E(i);
             0 ];% down velocity assumed zero as 2D position tracked only.



        %NED Step 8: Measurement Innovation dz = z - H * x_pred
        dz = z - H * x;

        %Steps 9 & 10: Measurement Update x = x + K * dz; P = (I - K * H) * P
        x = x + K * dz;
        P = (eye(6) - K * H) * P;

        %% TODO Step 11:Results = difference between DR and states x
        latitudes(i) = drlatitudes(i) - x(4);
        longitudes(i) = drlongitudes(i) - x(5);
        vN(i) = drvel_N(i) - x(1);
        vE(i) = drvel_E(i) - x(2);


    end



%% Results
%Concatenate into struct and export to excel sheet
    integratedResults.times = times';
```

```
    integratedResults.latitudes = latitudes;
    integratedResults.longitudes = longitudes;
    integratedResults.vel_N = vN;
    integratedResults.vel_E = vE;
    %integratedResults.heading = heading;
    % odometry = timestamp, deadreckoning Latitude, deadreckoing
    %           Longitude, velocity North, velocity East, heading
    % No header row
    %Make struct into columns , MAY NOT BE NECESSARY!
    fns = fieldnames(integratedResults);
    for iField = 1:numel (fns)
        integratedResults.(fns{iField}) = integratedResults.(fns{iField})';
    end
    % Convert to table and export to excel file
    writetable(struct2table(integratedResults), 'Results/integratedResults.xlsx')
    display(integratedResults)
end
```

Listing 2: Integrated GNSS Kalman filter Dead Reckoning Solution

```
function plottingFunction(Method, time, longitude, latitude, vN, vE)
%% Positioning plot
figure
plot(longitude, latitude, '-m');
title(Method, ' Position');
xlabel('Longitude (  )');
ylabel('Latitude (  )');
grid on;
saveas(gcf,['Results\', Method,'_Pos.png'])

%% Velocity plot
figure
plot(time, vN, '-m', ...
    time, vE, '--b');
legend('Velocity North', 'Velocity East');
xlabel('Time (s)');
ylabel('Velocity (m/s)');
title(Method, ' Velocities');
grid on;
saveas(gcf,['Results\', Method,'_Vel.png'])

%% Quiver plot
figure
quiver(longitude,latitude,vE,vN)
grid on;
xlabel('Longitude (  )');
ylabel('Latitude (  )');
title(['Quiver Plot of ', Method ,' Results'])
saveas(gcf,['Results\', Method ,'_Quiver.png'])

%% Positioning geoplot , ensure lat, long format
figure
geoplot(latitude, longitude, '-m');
title(Method, ' GeoPlot Position');
grid on;
saveas(gcf,['Results\', Method,'_Geoplot.png'])

end
```

Listing 3: Plotting function

```
function deadReckoningResults = deadReckoning(dead_reckoning_data, GNSSResults)
%INPUTS:
%       init_longitude:
%           GNSS  initial longitude result in radians
%       init_latitude:
%           GNSS initial latitude results in radians
%       init_altitude:
%           GNSS initial height result
%
%OUTPUTS:
```

```
%       deadReckoningResults    DR results struct with fields:
%          times        Time in seconds     this must match the data files.
%          latitudes    Geodetic latitude in degrees
%          longitudes   Geodetic longitude in degrees
%          vel_N        North velocity in metres per second
%          vel_E        East velocity in metres per second
%          heading      Heading in degrees

% Everything to 15 decimal places using format long
format long;
%Load in constants
Define_Constants;

%Load in GNSSResults
init_longitude = GNSSResults.longitude(1);
init_latitude = GNSSResults.latitude(1);
init_altitude = 0; %Set initial altitude as zero
%Initialise variables
times = dead_reckoning_data(:,1);
%rear wheels are driving wheels, thus 3 and 4
wheel_speed_3 = dead_reckoning_data(:,4);
wheel_speed_4 = dead_reckoning_data(:,5);
average_wheel_velocity = 0.5.*(wheel_speed_3 + wheel_speed_4);
heading = dead_reckoning_data(:,7) * deg_to_rad;

% Calculate longitude and latitudes
% Initialise empty matrix
vel_N = zeros(size(average_wheel_velocity));
vel_E = zeros(size(average_wheel_velocity));
% Initilaise first value of Velocity East, North
vel_N(1) = 0.5 * (cos(heading(1)) + cos(heading(1))) * average_wheel_velocity(1);
vel_E(1) = 0.5 * (sin(heading(1)) + sin(heading(1))) * average_wheel_velocity(1);

for i = 2:size(average_wheel_velocity,1)
    vel_N(i) = 0.5 * (cos(heading(i)) + cos(heading(i-1))) * average_wheel_velocity(i)
    ;
    vel_E(i) = 0.5 * (sin(heading(i)) + sin(heading(i-1))) * average_wheel_velocity(i)
    ;
end

% Initialise empty matrix
latitudes = zeros(size(times));
longitudes = zeros(size(times));
[Rad_N, Rad_E] = Radii_of_curvature(init_latitude);
latitudes(1) = init_latitude + vel_N(1) * (times(1) - 0) / (Rad_N + init_altitude);
longitudes(1) = init_longitude + vel_E(1) * (times(1) - 0) / ((Rad_E + init_altitude)
    * cos(latitudes(1)));
for i = 2:size(times,1)
    [Rad_N, Rad_E] = Radii_of_curvature(latitudes(i-1));
    latitudes(i) = latitudes(i-1) + vel_N(i) * (times(i)-times(i-1)) / (Rad_N +
    init_altitude);
    longitudes(i) = longitudes(i-1) + vel_E(i) * (times(i)-times(i-1)) / ((Rad_E +
    init_altitude) * cos(latitudes(i)));
end

% TODO Calculate damped instant velocitiy

v0 = average_wheel_velocity(1);
vN = zeros(size(vel_N));
vE = zeros(size(vel_E));
psi0 = heading(1);
psi = heading;
vN(1) = v0 * cos(psi0);
vE(1) = v0 * sin(psi0);
%L3AS58
for i = 2:size(vel_N,1)
    vN(i) = cos(psi(i)) * vel_N(i) - sin(psi(i)) * vN(i-1);
    vE(i) = sin(psi(i)) * vel_E(i) + cos(psi(i)) *vE(i-1);
end
%% TODO CONVERT TO RADIANS
%TODO Concatenate into struct and export to excel sheet
```

```
deadReckoningResults.times = times;
deadReckoningResults.latitudes = latitudes;
deadReckoningResults.longitudes = longitudes;
deadReckoningResults.vel_N = vN;
deadReckoningResults.vel_E = vE;
deadReckoningResults.heading = heading;
% odometry = timestamp, deadreckoning Latitude, deadreckoing
%           Longitude, velocity North, velocity East, heading
% No header row
% Convert to table and export to excel file

writetable(struct2table(deadReckoningResults), 'Results/deadReckoningResults.xlsx')

end
```

Listing 4: Dead Reckoning Solution

```
%Second iteration of cw1_script
%Script to execute COMP0130 ROBOT VISION AND NAVIGATION
%Coursework 1: Integrated Navigation for a Robotic Lawnmower
%Authors: Yuangshen (Jason) Zhang, Joseph Rowell & Vanessa Igodifo (Equal
%contributions)

close all;
clear;
clc;

%Import useful constants
Define_Constants;
%Set long format so numbers reported to 15 d.p
format long;

%% GNSS SOLUTION
% Import GNSS data
Pseudo_ranges_data = (readmatrix("data/Pseudo_ranges.csv"));
Pseudo_rates_data = (readmatrix("data/Pseudo_range_rates.csv"));
%Compute GNSS solution
GNSSResults = GNSSSolution(Pseudo_ranges_data, Pseudo_rates_data);

%% DEAD RECKONING SOLUTION
% Import dead reckoning data
dead_reckoning_data = csvread('data/Dead_reckoning.csv');
% Compute dead reckoning Solution
deadReckoningResults = deadReckoning(dead_reckoning_data, GNSSResults);

%% INTEGRATED SOLUTION
% % Compute integrated solution
integratedResults = IntegratedSolution(dead_reckoning_data, ...
    deadReckoningResults, Pseudo_rates_data, Pseudo_rates_data, GNSSResults);
%
% %% GNSS PLOTTING
plottingFunction('GNSS', GNSSResults.timeStamp, ...
    GNSSResults.longitude, GNSSResults.latitude, GNSSResults.velNorth, GNSSResults.
    velEast);

% %% Dead Reckoning PLOTTING
plottingFunction('Dead Reckoning', deadReckoningResults.times, ...
    deadReckoningResults.longitudes, deadReckoningResults.latitudes,
    deadReckoningResults.vel_N, deadReckoningResults.vel_E);


%% INTEGRATED SOLUTION PLOTTING
plottingFunction('Integrated Solution', integratedResults.times, ...
    integratedResults.longitudes, integratedResults.latitudes, integratedResults.vel_N
    , integratedResults.vel_E);
```

Listing 5: Main Script