1 Methodology

Task 1 aims to determine the horizontal velocity, position, and heading of a lawnmower. This is achieved through the integration of dead reckoning and GNSS data with Kalman filter. The approach is split into 3 main parts:

1.1 GNSS with Kalman filter

We implemented a method that computes position and velocity of a lawnmower with GNSS data through a Kalman filter, following steps in Figure 1. This process begins with the initialization of the state vector using a least squares estimate derived from two CSV files containing pseudo-ranges and pseudo-range rates from satellites, and error covariance matrix with GNSS error specifications (noise standard deviation of pseudo-ranges, pseudo-range rates, clock offset and clock drift). Initialising these two matrices provides a crucial initial estimate of the lawnmower's position and velocity.

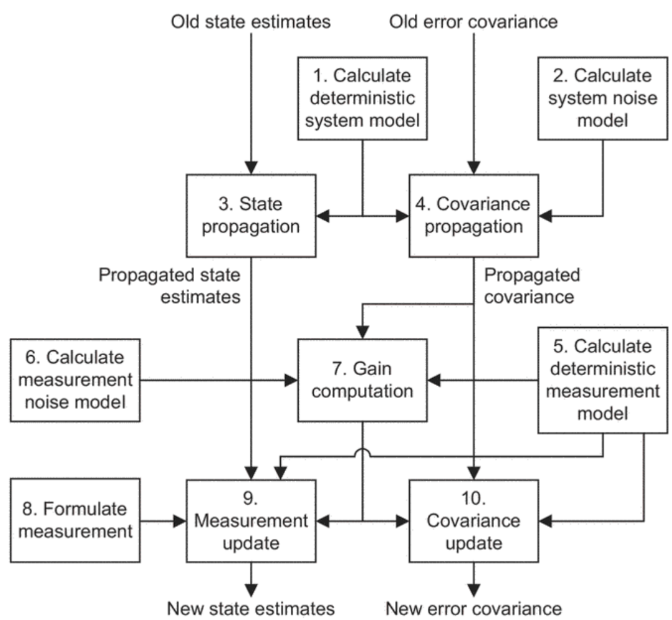

Figure 1: Kalman filter flowchart [4]

With these state vectors and error covariance matrices of position and velocity of lawnmower can be computed with two main stages of Kalman filter: state propagation and measurement update. In state propagation, the state vector and error covariance matrix are advanced in time, considering system dynamics, process noise and system noise covariance matrix defining by GNSS error specifications (power spectral density of clock offset and clock drift). The measurement update then refines these estimates using the latest GNSS measurements, taking into account measurement noise and updating the state with the Kalman gain. An essential part of this process is the robust detection and handling of `outliers` in the GNSS data, ensuring the accuracy of the filter's outputs.

Finally, the estimated position and velocity of the lawnmower, processed through the Kalman filter, are converted into North, East, Down velocities and coordinates for practical application.

1.2 Dead reckoning

The process begins with dead reckoning, as detailed in Figure 2. The formula for computing the forward and lateral speed is robust, accounting for potential errors in individual wheel speed sensors. This method improves the understanding of the lawnmower's dynamics, particularly in terms of rotational movement and lateral velocity. It's essential that the gyroscope measurements are precise for accurate velocity computation.

$$v_{\text{forward}} = \frac{1}{4}\sum_{j=1}^{4} v_{\text{wheel},j}$$

$$\begin{cases} v_{\text{lat}} = 0, & \text{if } v_{\text{forward}} = 0 \\ v_{\text{lat}} = v_{\text{forward}} \tan\left(\frac{\omega L}{v_{\text{forward}}}\right), & \text{otherwise} \end{cases}$$

Figure 2: Formula for Computing Forward and Lateral Speed

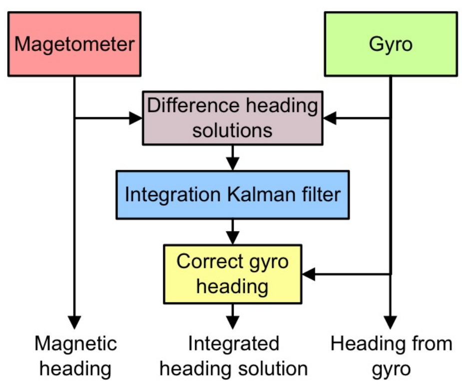

Figure 3: Flowchart of Gyro-Magnetometer Integration [4]

Once the velocity is calculated, the Gyroscopic heading is determined. We use **Gyro-Magnetometer Integration** [3] to reduce the error and bias of heading. This can be acheived as figure 3 which calculates difference between magnetic heading which is obtained from *dead_reckoning.csv* and heading that are calculated with angular rate in (1) with Sensor error Specifications (gyroscope bias standard deviation, wheel speed measurement errors and heading error variance). After that, Kalman filter is applied to determined the error and bias, and corrected the solution of each state. From this approach, we can overcome weakness of both heading with higher robustness.

$$\psi_{pb}(t) = \psi_{pb}(t_0) + \int_{t_0}^{t} \omega^{b}_{pb,z}(t')\, dt' \tag{1}$$

Dead reckoning is particularly suit for lawnmowers operating in areas where GPS signals are weak or nonexistent. This method continuously provides navigation data and is resilient to interference from external signals. However, despite its advantages, it is rely on performance of sensors which are not precise in this task due to low-cost MEMS gyroscope. Therefore, to enhance accuracy and overcome this limitation, we integrate dead reckoning with GNSS data by using a Kalman filter. This integration effectively combines the continuous data from dead reckoning with the precision of GNSS, resulting in more accurate navigation for the lawnmower.

1.3 GNSS Dead Reckoing integration with Kalman filter

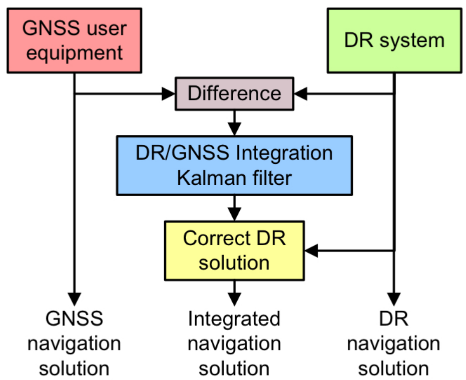

Figure 4: Flowchart of Dead reckoning-GNSS Integration [4]

The final part involves integrating dead reckoing and GNSS data with Kalman filter as depicted in Figure 4. This integration involves comparing data from the Global Navigation Satellite System (GNSS) from section 1.2 with the lawnmower's dead reckoning calculations from section 1.1. The Kalman filter adjusts for drifts or errors in the dead reckoning method, enhancing accuracy.

The implementation stages of the Kalman filter are outlined in Figure 4. The process starts with initializing the state vector defining the error between 2 methods, which is assuming to be zero, and error covariance matrix consisting noise standard deviation of each error. The transition matrix updates this state based on the lawnmower's dynamics. The Kalman filter refines predictions by incorporating GNSS measurements and employing a gain matrix to adjust the predicted state. This step significantly corrects the position and velocity estimates, reducing cumulative errors from dead reckoning.

Finally, the corrected position, velocity, and heading for each epoch provide a refined, accurate estimate of the lawnmower's state. Integrating these two methods enhances the accuracy and reliability of the navigation system, providing a robust solution against sensor errors and external factors affecting GNSS data, crucial for precise lawnmower navigation.

This approach, as mentioned in [2], combines the strengths of both dead reckoning and Kalman filtering to ensure efficient and accurate navigation of the lawnmower.

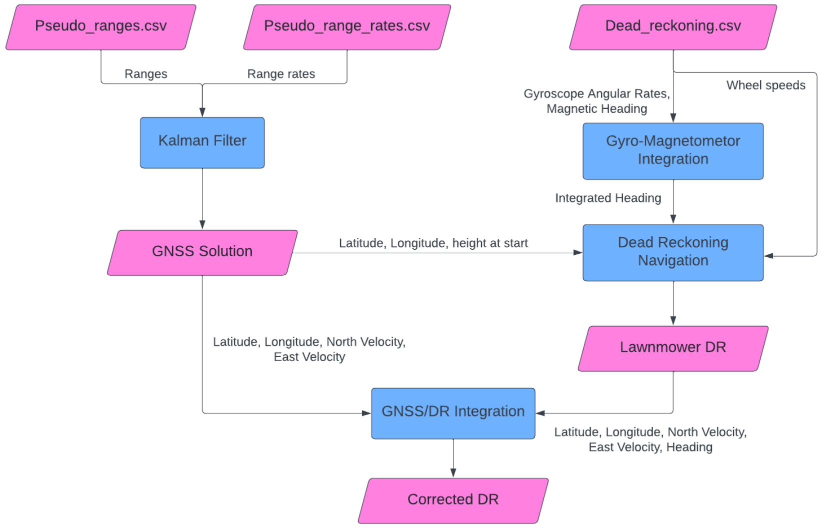

Figure 5: Flowchart of Integration process

2 Evaluation

2.1 GNSS with Kalman filter

In the analysis of GNSS data, as depicted in Figure 6, we identified certain time intervals with irregular positioning, indicative of outliers. This deviation from expected patterns could significantly impact the accuracy of GNSS-based navigation and tracking. To rectify this, we applied an outlier detection method, leading to the improved positioning shown in Figure 7, where the data points align more consistently with expected trajectories.

Figure 6: Result of GNSS with outlier handling

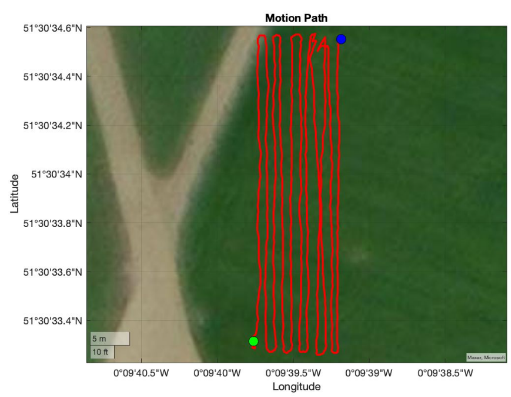

Figure 7: Result of GNSS with outlier handling

However, a residual fuzzy effect, a form of noise, remains evident in the data. This noise, potentially arising from factors like atmospheric disturbances, multipath effects, or system errors, can degrade the precision of GNSS measurements.

2.2 Dead reckoning

From the result of Dead-Reckoning (Figure 8), it visualizes a motion pattern of lawnmower zigzagging on the grass in Hyde Park. The lines between each turn are smooth and the turning motion shows a relatively clearer circle. However, the margin of longitude between each turn is lower than final result's. The motion path slightly shrink along the horizontal direction. This indicates that the errors occur when determining the longitude after each turn, because, from the graph, it turned more than 180 degree, and then turned opposite to move along the line. It can be assumed that due to low quality of gyroscope. The accuracy of gyroscope continuously decrease as the lawnmower moves.

Figure 8: Result of Dead-Reckoning

2.3 GNSS Dead Reckoing integration with Kalman filter

In Figure 9, 10, 11, 12, the graphs are the result of north velocity, east velocity, heading, and coordinates of final solution respectively.

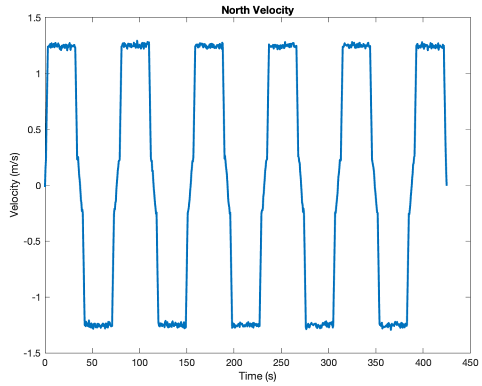

Figure 9: Results of North Velocity

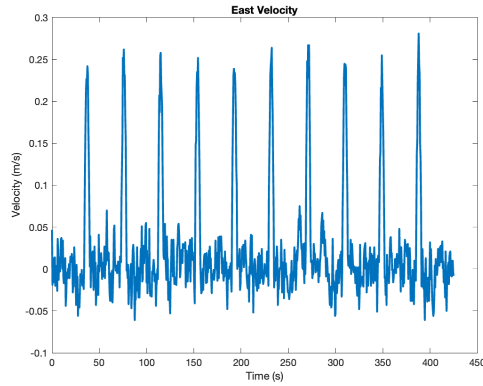

Figure 10: Results of East Velocity

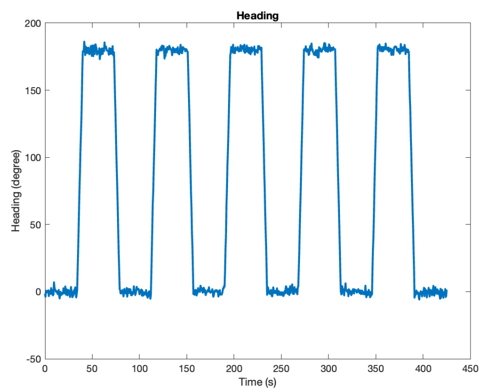

Figure 11: Results of Heading

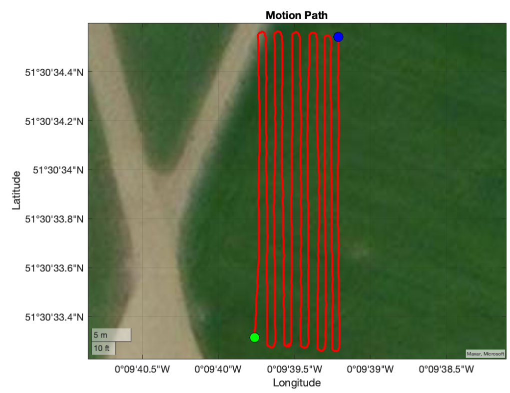

Figure 12: Result of Latitude and Longitude

The application of the Kalman filter to the integration of GNSS and Dead Reckoning data has significantly enhanced the realism and smoothness of the motion path (as illustrated in Figure 12). This advancement is particularly evident in the reduction of the noisy patterns observed in previous solutions. Additionally, the trajectory post-turns now appears more natural compared to the results obtained solely from Dead Reckoning or GNSS methods. Despite some minor sharp edges observed during the 180-degree turns made by the lawnmower, the overall solution demonstrates reliability. The map depicted in Figure 12 successfully visualizes a motion path that aligns closely with typical lawnmower behavior.

Focusing on the aspects of velocity, heading, and position (as shown in Figures 9, 10, and 11), a noticeable improvement is seen in terms of smoothness, indicative of reduced noise. This enhancement is attributed to the synergistic effect of integrating Dead Reckoning and GNSS data [1], which effectively compensates for individual errors in each system, resulting in a more robust solution. Remarkably, the heading data exhibits comparatively lower noise levels than both velocity and position, despite being computed solely from Dead Reckoning. This improved accuracy can be credited to the integration of gyroscopic and magnetometer data, which underscores the efficacy of Gyro-Magnetometer Integration in refining heading measurements.

References

[1] "de Juan, A., & Tauler, R. (2019). Data Fusion by Multivariate Curve Resolution. In Data Handling in Science and Technology (Vol. 31, pp. 205-233). Elsevier. https://doi.org/10.1016/B978-0-444-63984-4.00008-9"

[2] "J.Z. Sasiadek, P. Hartana, Sensor Fusion for Dead-Reckoning Mobile Robot Navigation, IFAC Proceedings Volumes, Volume 34, Issue 4, 2001, Pages 251-256, ISSN 1474-6670."

[3] "Ladetto, Q., & Merminod, B. (January 2002). Digital magnetic compass and gyroscope integration for pedestrian navigation. Faculté ENAC - Institut du Développement Territorial, Geodetic Laboratory (TOPO), EPFL. Retrieved from http://topo.epfl.ch."

[4] "P. D. Groves, Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems, 2nd Edition, Artech House, 2013."

[5] "Sasiadek, J. Z., & Hartana, P. (2001). Sensor Fusion for Dead-Reckoning Mobile Robot Navigation. IFAC Proceedings Volumes, 34(4), 251-256. https://doi.org/10.1016/S1474-6670(17)34304-5"

3 Appendix

3.1 GNSS with Kalman filter

```
function cw1_kalman()

    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]);

    %initialising state vector x_0 and error covaraicne matrix P_0
    [x_0, P_0] = Initialize_Pos;

    % Read pseudo-range and pseudo-range rate data
    ranges = csvread("data/Pseudo_ranges.csv"); %#ok<CSVRD>
    range_rates = csvread("data/Pseudo_range_rates.csv"); %#ok<CSVRD>
    epochs = size(ranges, 1) - 1; % Time variable

    % Threshold of outlier detection
    T = 6;

    Pos_vel_NEC = {'Time', 'Latitude (deg)', 'Longitude (deg)', 'Height',
        'Velo(N)', 'Velo(E)', 'Velo(D)'};

    % Loop through all epochs
    for epoch = 2:epochs+1

        % time interval
        i = 0.5;
        % Transition matrix (step 1)
        t = [1 0 0 i 0 0 0 0;...
             0 1 0 0 i 0 0 0;...
             0 0 1 0 0 i 0 0;...
             0 0 0 1 0 0 0 0;...
             0 0 0 0 1 0 0 0;...
             0 0 0 0 0 1 0 0;...
             0 0 0 0 0 0 1 i;...
             0 0 0 0 0 0 0 1];

        % Acceleration
        S_e_a = 5;
        % Clock phase
        S_a_c = 0.01;
        % Clock frequency
        S_a_cf = 0.04;

        % System noise covariance matrix (step 2)
        q1 = S_e_a * i^3 / 3;
```

```
        q2 = S_e_a * i^2 / 2;
        q3 = S_e_a * i;
        q4 = (S_a_c * i) + (S_a_cf * i^3 / 3);
        q5 = S_a_cf * i^2 / 2;
        q6 = S_a_cf * i;

        Q = [q1   0   0 q2   0   0   0   0;...
              0  q1   0   0 q2   0   0   0;...
              0   0  q1   0   0 q2   0   0;...
             q2   0   0 q3   0   0   0   0;...
              0  q2   0   0 q3   0   0   0;...
              0   0  q2   0   0 q3   0   0;...
              0   0   0   0   0   0 q4  q5;...
              0   0   0   0   0   0 q5  q6;];

        % State estimate (step 3)
        x_1 = t * x_0;

        % Upadate error covaricance matrix (step 4)
        P_1 = t * P_0 * t.' + Q;

        %Compute positon and Velocity of Satellite
        n_sat = size(ranges, 2) - 1;
        sat_r_arr = zeros(n_sat, 3);
        sat_v_arr = zeros(n_sat, 3);
        for i = 1:n_sat
            time = ranges(epoch,1);
            j = ranges(1, i+1);
            [sat_r_arr(i, 1:3), sat_v_arr(i, 1:3)] =
                Satellite_position_and_velocity(time, j);
        end

        r_aj_arr = zeros(n_sat, 1);
        r_aj_r_arr = zeros(n_sat, 1);
        u = zeros(n_sat, 3);

        % Predict the ranges from the approximate user position to each
            satellite
        for m = 1:n_sat
            r_aj = ranges(epoch, m+1);
            r_ej = sat_r_arr(m , 1:3).';
            v_ej = sat_v_arr(m , 1:3).';
            for n = 1:2
                q = omega_ie * r_aj / c;
                C = [1 q 0; -q 1 0; 0 0 1];

                r_2 = C*r_ej - x_1(1:3);
                r_aj = sqrt(r_2.' * r_2);
            end
            q = omega_ie * r_aj / c;
```

```matlab
            C = [1 q 0; -q 1 0; 0 0 1];

            u(m, 1:3) = (C*r_ej - x_1(1:3)) / r_aj;
            r_aj_arr(m) = r_aj;
            r_aj_r_arr(m) = u(m, 1:3) * (C * (v_ej + Omega_ie * r_ej) - (
                x_1(4:6) + Omega_ie * x_1(1:3)));
        end

        % Measurement matrix (Step 5)
        H = zeros(n_sat*2, 8);
        for m = 1:n_sat*2
            if m <= n_sat
                H(m, :) = [-u(m, 1) -u(m, 2) -u(m, 3) 0 0 0 1 0];
            else
                H(m, :) = [0 0 0 -u(m-n_sat, 1) -u(m-n_sat, 2) -u(m-n_sat
                    , 3) 0 1];
            end
        end

        % Standard deviation
        % Pseudo-range measurements
        std_p = 10;
        % Pseudo-range-rate measurements
        std_r = 0.05;

        % Update Measurement noise covariance matrix (Step 6)
        R = eye(n_sat*2);
        for m = 1:n_sat*2
            if m <= n_sat
                R(m, m) = std_p^2;
            else
                R(m, m) = std_r^2;
            end
        end


        % Update the Kalman gain matrix (Step 7)
        K = P_1 * H.' / (H * P_1 * H.' + R);

        % Form the measurement innovation vector (Step 8)
        z = zeros(n_sat*2, 1);
        for m = 1:n_sat*2
            if m <= n_sat
                z(m) = ranges(epoch, m+1) - r_aj_arr(m) - x_1(7);
            else
                z(m) = range_rates(epoch, m-n_sat+1) - r_aj_r_arr(m-n_sat
                    ) - x_1(8);
            end
        end

```

```matlab
    % ---------------- OUTLIER DETECTION - residuals vector v --------------------
    I_m = eye(16, 16);
    v = (H * inv(H' * H) * H' - I_m) * z;

    % Compute the residuals covariance matrix Cv
    sigma_p = 10;  % measurement error standard deviation
    Cv = (I_m - H * inv(H' * H) * H') * sigma_p^2;

    %Compute the normalized residuals and detect outliers
    normalized_residuals = abs(v) ./ sqrt(diag(Cv));
    outliers = normalized_residuals > T;

    % If any outliers are detected, recalculate your position without the outliers
    if any(outliers)
        H(outliers, :) = [];
        R(outliers, :) = [];
        R(:, outliers) = [];  % Remove the columns corresponding to the outliers
        z(outliers) = [];  % Remove the outlier measurements

        % Recalculate the Kalman gain matrix without outliers
        K = P_1 * H.' / (H * P_1 * H.' + R);

        % Update the state estimate without outliers (Step 9)
        x_1 = x_1 + K * z;

        % Update the error covariance matrix without outliers (Step 10)
        P_1 = (eye(size(K,1)) - K * H) * P_1;
    else


        % Update the state estimate (Step 9)
        x_1 = x_1 + K * z;

        % Update the error covariance matrix (Step 10)
        P_1 = (eye(8) - K * H) * P_1;
    end

    % Convert the state estimate to NED coordinates
    [lat,long,h,v] = pv_ECEF_to_NED(x_1(1:3),x_1(4:6));
    lat = round(lat * rad_to_deg,6);
    long = round(long * rad_to_deg,6);

    Pos_vel_NEC(end+1, :) = {(epoch-2)/2, lat, long, h, v(1), v(2), v(3)};

    % Update x_0 and P_0
```

```
            x_0 = x_1;
            P_0 = P_1;

        end

        Pos_vel_NEC(1, :) = []; % Remove the first row which is the header
        writecell(Pos_vel_NEC, 'ans/CW_GNSS_Pos_Vel.csv');

end

% ===========================================================================

%
% Initial state
function [x_k_est, P_k_est] = Initialize_Pos

    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]);

    pseudo_ranges = csvread('data/Pseudo_ranges.csv'); %#ok<CSVRD>
    pseudo_range_rates = csvread('data/Pseudo_range_rates.csv'); %#ok<CSVRD>

    % Number of satellites
    n_sat = size(pseudo_ranges, 2) - 1;

    %Compute positon and velocity of Satellite
    sat_r_arr = zeros(n_sat, 3);
    sat_v_arr = zeros(n_sat, 3);
    time = 0;
    for i = 1:n_sat
        j = pseudo_ranges(1, i+1);
        [sat_r_arr(i, 1:3), sat_v_arr(i, 1:3)] = Satellite_position_and_velocity(time, j);
    end

    r_eb_e = [0; 0; 0]; % Initialize ECEF position
    c_offset = 1E-04; % Clock offset
    v_eb_e = [0; 0; 0]; %Initialize ECEF velocity
    c_drift = 1E-04; % Clock drift

    % Predict the ranges from the approximate user position to each satellite
    % Iterate until find the solution
    while true

        % Ranges from the approximate position to each satellite
        r_aj_p = zeros(n_sat, 1);
```

```matlab
        % Range rates from the approximate velocity to each satellite
        r_aj_r = zeros(n_sat, 1);
        % Line-of-sight unit vector
        u = zeros(n_sat, 3);

        for m = 1:n_sat
            r_aj = pseudo_ranges(2, m+1);
            r_ej = sat_r_arr(m, 1:3).';
            v_ej = sat_v_arr(m, 1:3).';
            for n = 1:2
                q = omega_ie * r_aj / c;
                C = [1 q 0; -q 1 0; 0 0 1];
                r_2 = C*r_ej - r_eb_e;
                r_aj = sqrt(r_2.' * r_2);
            end
            r_aj_p(m) = r_aj;
            % Compute unit vector
            u(m, 1:3) = (C*r_ej - r_eb_e) / r_aj;
            % Compute range rates
            r_aj_r(m) = u(m, 1:3) * (C * (v_ej + Omega_ie * r_ej) - (v_eb_e + Omega_ie * r_eb_e));
        end

        % Position state
        x_r = [r_eb_e; c_offset];
        % Velocity state
        x_v = [v_eb_e; c_drift];
        % Position measurement innovation
        z_r = zeros(n_sat, 1);
        % Velocity measurement innovation
        z_v = zeros(n_sat, 1);
        % Measurement matrix
        H = ones(n_sat, 4);
        for i = 1:n_sat
            z_r(i) = pseudo_ranges(2, i+1) - r_aj_p(i) - c_offset;
            z_v(i) = pseudo_range_rates(2, i+1) - r_aj_r(i) - c_drift;
            H(i, :) = [-u(i, 1) -u(i, 2) -u(i, 3) 1];
        end

        % Update state vector
        x_r = x_r + (H.' * H) \ H.' * z_r;
        x_v = x_v + (H.' * H) \ H.' * z_v;

        % Get new ECEF position and velocity, clock offset, clock drift
        r_eb_e_z = x_r(1:3);
        c_offset = x_r(4);
        v_eb_e_z = x_v(1:3);
        c_drift = x_v(4);

        % Limit the error
```

```
        limit = 0.1;
        % If the new ones is not much different from previous ones, leave
        % the loop
        if sqrt((r_eb_e(1) - r_eb_e_z(1))^2 +...
                (r_eb_e(2) - r_eb_e_z(2))^2 +...
                (r_eb_e(3) - r_eb_e_z(3))^2) < limit
            break
        end

        % Update ECEF position and velocity, clock offset, clock drift
        r_eb_e = r_eb_e_z;
        v_eb_e = v_eb_e_z;
    end

    std_p = 10; % noise standard deviation of pseudo-range
    std_v = 0.05; % noise standard deviation of pseudo-range rate
    std_co = 100000; % Clock offset standard deviation
    std_cd = 200; % Clock drift standard deviation

    % Initialize state
    x_k_est = [r_eb_e_z; v_eb_e_z; c_offset; c_drift];
    P_k_est = [std_p^2 0       0       0       0       0       0
        0;...
               0       std_p^2 0       0       0       0       0
                   0;...
               0       0       std_p^2 0       0       0       0
                   0;...
               0       0       0       std_v^2 0       0       0
                   0;...
               0       0       0       0       std_v^2 0       0
                   0;...
               0       0       0       0       0       std_v^2 0
                   0;...
               0       0       0       0       0       0       std_co^2
                   0;...
               0       0       0       0       0       0       0
                   std_cd^2;];

end
```

3.2 Dead reckoning

```
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;
e = 0.0818191908425; %WGS84 eccentricity

```

```matlab
% Read GNSS result and dead reckoning data
dead_reckoning = csvread("data/Dead_reckoning.csv"); %#ok<CSVRD>
gnss = csvread("ans/CW_GNSS_Pos_Vel.csv"); %#ok<CSVRD>

% Number of epoches
epochs = size(dead_reckoning, 1);

lat = gnss(1, 2); % initial lattitude
long = gnss(1, 3); % initial longtitude
h = gnss(1, 4); % Geodetic height

% Calculate latitude, longitude, velocity and heading from initial position
% and dead reckoning
lawnmower_dr = Dead_Reckoning(lat, long, h, dead_reckoning);

filename = 'ans/Lawnmower_DR.csv';
writematrix(lawnmower_dr, filename);

% ==========================================================================
%
% Calculate latitude, longitude, velocity and heading from initial position
% and dead reckoning
function converted_dr = Dead_Reckoning(lat, long, h, dead_reckoning)

    deg_to_rad = 0.01745329252; % Degrees to radians conversion factor
    rad_to_deg = 1/deg_to_rad; % Radians to degrees conversion factor

    L = 0.5; % Wheel Base

    % Number of epoches
    epoches = size(dead_reckoning, 1);

    % Define an array of dead reckoning solution
    converted_dr = zeros(epoches, 6);

    % Convert latitude and longitude from degree to radian
    lat = lat * deg_to_rad;
    long = long * deg_to_rad;

    % Convert heading with Gyro-Magnetometor Integration
    heading = Gyro_Integration(dead_reckoning(:,6), dead_reckoning(:,7));


    for i = 1:epoches
        % Average wheel speeds for forward speed
        wheel_speeds = dead_reckoning(i, 2:5);
```

```
        v_forward = mean(wheel_speeds);
        % Compute lateral speed
        if v_forward == 0
            v_lat = 0;
        else
            delta = (dead_reckoning(i, 6) * L) / v_forward;
            v_lat = v_forward * tan(delta);
        end
        % Compute north velocity and east velocity
        v_n = v_forward * cos(heading(i)) - v_lat * sin(heading(i));
        v_e = v_forward * sin(heading(i)) + v_lat * cos(heading(i));

        % Compute north and east curvature
        [R_N,R_E] = Radii_of_curvature(lat);
        % Update latitude and longitude
        lat = lat + (v_n * 0.5 / (R_N + h));
        long = long + (v_e * 0.5 / ((R_E + h) * cos(lat)));

        % Save the data in the solution
        converted_dr(i, 1) = dead_reckoning(i, 1);
        converted_dr(i, 2) = lat * rad_to_deg;
        converted_dr(i, 3) = long * rad_to_deg;
        converted_dr(i, 4) = v_n;
        converted_dr(i, 5) = v_e;
        converted_dr(i, 6) = heading(i) * rad_to_deg;
    end

end


% =========================================================================
%
% Convert heading with Gyro-Magnetometor Integration
function h_integrated = Gyro_Integration(gyro_rate, mag_heading)

    deg_to_rad = 0.01745329252; % Degrees to radians conversion factor
    rad_to_deg = 1/deg_to_rad; % Radians to degrees conversion factor

    % Number of epoches
    epoches = size(gyro_rate, 1);
    % Define an array of heading solution
    h_integrated = zeros(epoches, 1);
    % Define an array of heading from gyroscope
    gyro_heading = zeros(epoches, 1);

    t = 0.5; % time interval
    S_rg = 3E-06; % PSD of gyroscope measurement errors
    S_bgd = 0; % PSD of gyroscope bias errors
```

```
    std_b = 1 * deg_to_rad; % bias standard deviation
    std_g = 3E-06; % gyroscope heading error variance
    std_m = 3E-06; % magnetometor heading error variance

    % Compute heading from gyroscope
    gyro_heading(1) = mag_heading(1) * deg_to_rad;
    for epoch = 2:epoches
        gyro_heading(epoch) = gyro_heading(epoch - 1) + gyro_rate(epoch)
            * t;
    end

    % transition matrix
    T = [1 t;
         0 1];

    %  system noise covariance matrix
    Q = [(S_rg*t)+(S_bgd*t^3)/3 (S_bgd*t^2)/2;
         (S_bgd*t^2)/2           S_bgd*t];

    % Measurement Matrix
    H = [-1   0;
          0  -1];

     % Measurement Noise Covariance Matrix
    R = [std_m^2 0;
         0       std_m^2];

    % Initialize state filter
    x = [0; 0];
    % Initialize state estimation error covariance matrix
    P = [std_g^2 0;
         0       std_b^2];

    % Kalman filter measurement
    for epoch = 1:epoches

        % Propagate state
        x = T * x;
        P = T * P * T.' + Q;

        % Kalman gain matrix
        K = P * H.' \ (H * P * H.' + R);

        % Measurement innovation
        z = [(mag_heading(epoch)*deg_to_rad - gyro_heading(epoch)); 0] -
            H * x;

        % Update state
        x = x + K * z;
        P = (eye(2) - K * H) * P;
```

```
        % Save the data in the solution
        h_integrated(epoch) = (gyro_heading(epoch) - x(1));
    end

end
```

3.3 GNSS Dead Reckoing integration with Kalman filter

```
function cw1_dr_kalman()

    gnss = csvread("ans/CW_GNSS_Pos_Vel.csv"); %#ok<CSVRD>
    lawnmower_dr = csvread("ans/Lawnmower_DR.csv"); %#ok<CSVRD>

    deg_to_rad = 0.01745329252; % Degrees to radians conversion factor
    rad_to_deg = 1/deg_to_rad; % Radians to degrees conversion factor

    % Number of epoches
    epoches = size(gnss, 1);

    t = 0.5; % time interval
    u_r = 10; % initial position uncertainty
    u_v = 0.1; % initial velocity uncertainty
    e_gr = 5; % GNSS position std
    e_gv = 0.02; % velocity std
    S_DR = 0.2; % DR velocity error power spectral density

    % Define an array of dead reckoning solution
    dr_solution = zeros(epoches, 6);

    lat_G = gnss(1, 2) * deg_to_rad; % Initial lattitude
    long_G = gnss(1, 3) * deg_to_rad; % Initial longtitude
    h = gnss(1, 4); % Initial height

    % Compute north and east curvature
    [R_N,R_E] = Radii_of_curvature(lat_G);

    % state filter
    x = [0; 0; 0; 0];
    % state estimation error covariance matrix
    P = [u_v^2 0      0                  0;
         0     u_v^2  0                  0;
         0     0      (u_r/(R_N + h))^2  0;
         0     0      0                  (u_r/((R_E + h)*cos(lat_G)))^2];


    for i = 1:epoches

        % transition matrix
        T = [1          0                         0  0;
```

```
            0            1                       0  0;
            t/(R_N + h)  0                       1  0;
            0            t/((R_E + h)*cos(lat_G)) 0  1];

        %  system noise covariance matrix
        q1 = (S_DR * t^2) / (2 * (R_N + h));
        q2 = (S_DR * t^2) / (2 * (R_E + h) * cos(lat_G));
        q3 = (S_DR * t^3) / (3 * (R_N + h)^2);
        q4 = (S_DR * t^3) / (3 * ((R_E + h) * cos(lat_G))^2);
        Q = [S_DR*t  0       q1  0;
             0       S_DR*t  0   q2;
             q1      0       q3  0;
             0       q2      0   q4];

        % Data from GNSS
        lat_G = gnss(i, 2) * deg_to_rad;
        long_G = gnss(i, 3) * deg_to_rad;
        h = gnss(i, 4);
        v_n_G = gnss(i, 5);
        v_e_G = gnss(i, 6);
        % Data from dead reckoning
        lat_D = lawnmower_dr(i, 2) * deg_to_rad;
        long_D = lawnmower_dr(i, 3) * deg_to_rad;
        v_n_D = lawnmower_dr(i, 4);
        v_e_D = lawnmower_dr(i, 5);

        % Compute north and east curvature
        [R_N,R_E] = Radii_of_curvature(lat_G);

        % Propagate state
        x = T * x;
        P = T * P * T.' + Q;

        % measurement matrix
        H = [ 0   0  -1   0;
              0   0   0  -1;
             -1   0   0   0;
              0  -1   0   0];

        % measurement noise covariance matrix
        R = [(e_gr/(R_N + h))^2 0                                0       0;
             0                  (e_gr/((R_E + h)*cos(lat_G)))^2 0       0;
             0                  0                                e_gv^2  0;
             0                  0                                0       e_gv
            ^2];

        % Kalman gain matrix
        K = P * H.' / (H * P * H.' + R);

        % Measurement innovation
```

```
        z = [lat_G - lat_D; long_G - long_D; v_n_G - v_n_D; v_e_G - v_e_D
            ];
        z = z - H * x;

        % Update state
        x = x + K * z;
        P = (eye(4) - K * H) * P;

        % Save the data in the solution
        dr_solution(i, 1) = gnss(i, 1);
        dr_solution(i, 2) = round((lat_D - x(3)) * rad_to_deg, 6);
        dr_solution(i, 3) = round((long_D - x(4)) * rad_to_deg, 6);
        dr_solution(i, 4) = round(v_n_D - x(1), 3);
        dr_solution(i, 5) = round(v_e_D - x(2), 3);
        dr_solution(i, 6) = lawnmower_dr(i, 6);

    end

    filename = 'ans/Corrected_DR.csv';
    writematrix(dr_solution, filename);
end
```