COMP0130: Robot Vision and Navigation Coursework 1

Team: 13

Student Numbers: 23219913, 23204577, 23223992

ucabria@ucl.ac.uk, ucaba78@ucl.ac.uk, ucabaf0@ucl.ac.uk

February 8th 2024

CONTENTS

1 Provided Data **2**

2 Principles and Practices **4**

2.1 Methodology 4

2.2 Constants 5

2.3 Conventions 5

3 Independent GNSS solution **6**

3.1 Position in Epoch 0 6

3.2 Velocity in Epoch 0 7

3.3 Position and Velocity after Epoch 0 8

4 Independent Dead reckoning solution **10**

4.1 Preprocessing 10

4.2 Dead reckoning solution calculation 10

5 Integrating GNSS and DR **14**

5.1 Initialisation and calculation at Epoch 0 14

5.2 Epoch 1 and later 15

Appendix **17**

References **41**

1. PROVIDED DATA

We have been provided with three data files, all in comma-separated variable (CSV) format:

1. In the pseudo-ranges file, the first column contains the time tag in seconds and the first row contains the satellite numbers. The remaining rows and columns contain the corresponding pseudo-range measurements in m

	5	6	7	...	15	30
0	20900805.52	21303508.29	24665467.83	...	22243173.47	24180826.31
0.5	20900686.22	21303595.57	24665530.38	...	22243515.65	24180507.75
1	20900565.71	21303685.11	24665592.12	...	22243860.55	24180191.29

2. In the pseudo-range rates file, the first column contains the time in seconds and the first row contains the satellite numbers. The remaining rows and columns contain the corresponding pseudo-range rate measurements in ms^{-1}

	5	6	7	...	15	30
0	-239.9727047	176.7756164	125.6414328	...	691.0606862	-632.4830396
0.5	-239.96835	176.8871874	125.6176636	...	691.2458095	-632.3350689
1	-239.8468923	176.9181996	125.5792242	...	691.3256075	-632.2345457

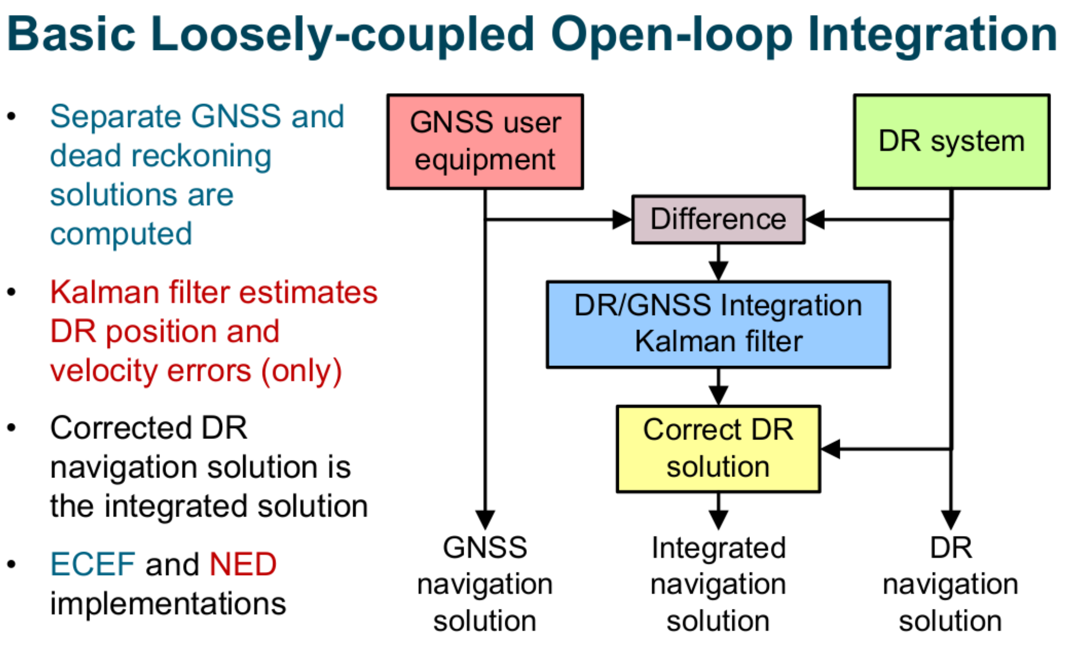
3. In the dead-reckoning file, column 1 contains time in seconds, columns 2 to 5 contain the wheel-speed measurements from sensors 1 to 4 in ms^{-1}. Column 6 contains the gyroscope angular rate measurements in radians per second. Column 7 contains the heading measurements in degrees from the magnetic compass

time	Wheel 1	Wheel 2	Wheel 3	Wheel 4	Gyroscope angular rate	Heading
0	0	0	0	0	0	-1.542892291
0.5	0.06	0.04	-0.02	0	-0.014	-3.797893258
1	0.16	0.14	0.1	0.16	-0.008	0.7524558213

2. PRINCIPLES AND PRACTICES

2.1. METHODOLOGY

Here is how we'll do it (Fig. 2.1). Image credits Week3 Lecture 3B slide 20. Our solution and design was inspired heavily from the material in [1] and Workshops 1 2 and 3



Basic Loosely-coupled Open-loop Integration

- Separate GNSS and dead reckoning solutions are computed
- Kalman filter estimates DR position and velocity errors (only)
- Corrected DR navigation solution is the integrated solution
- ECEF and NED implementations

Figure 2.1: Basic Loosely-coupled Open-loop Integration

The whole report can be broken up into 3 main chunks

1. Computing GNSS position and velocity solution at all epochs using pseudo range and pseudo range-rate measurements Workshop 1 Task 1B, 3, 4 and Workshop 2 Task 2A and 2B

2. Computing DR position and velocity solution at every epoch using odometer and compass observations Workshop 3 Task 1

3. Computing an integrated horizontal only DR/GNSS solution using Kalman filtering Workshop 3 Task 2

2.2. CONSTANTS

Apart from the constants defined in *Define_Constants.m* we use a number of other pre defined values

1. $T = 6$ in outlier detection threshold for Epoch 0 GNSS position calculation
2. Acceleration power spectral density $S_a^e = 5m^2s^{-3}$
3. Pseudo range measurements standard deviation $\sigma_\rho = 10m$
4. Pseudo range rate measurements standard deviation $\sigma_{\dot{\rho}} = 0.05ms^{-1}$

2.3. CONVENTIONS

1. Algorithms and steps utilised from the workshops are referenced like this Workshop Task 1A
2. Custom Matlab scripts used are referenced like this Single_Epoch_position.m . All the scripts referenced can be found in 5.2

3. INDEPENDENT GNSS SOLUTION

The entire procedure for deriving GNSS position and velocity solutions is divided into two principal segments: the initial epoch (Epoch 0) and subsequent epochs 3.1. Epoch 0 is defined as the moment when the elapsed time equals 0 seconds, marking the commencement of the measurement sequence.

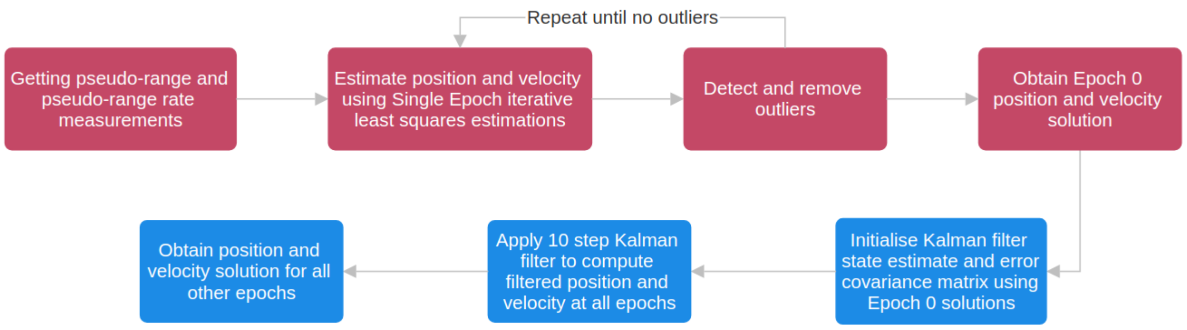

Repeat until no outliers
Getting pseudo-range and pseudo-range rate measurements
Estimate position and velocity using Single Epoch iterative least squares estimations
Detect and remove outliers
Obtain Epoch 0 position and velocity solution
Initialise Kalman filter state estimate and error covariance matrix using Epoch 0 solutions
Apply 10 step Kalman filter to compute filtered position and velocity at all epochs
Obtain position and velocity solution for all other epochs

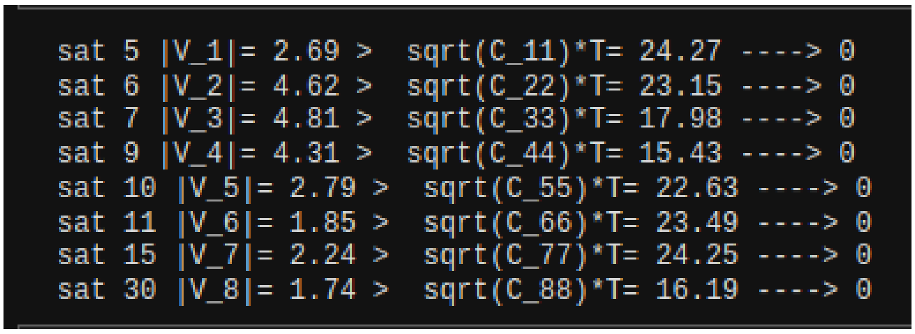
Figure 3.1: Flowchart for computing GNSS position and velocity estimates

3.1. POSITION IN EPOCH 0

The GNSS Solution for Epoch 0 is calculated through a **unweighted least-squares** method, using the equation 5 in task A utilizing the Earth Centered-Earth Fixed (ECEF) coordinate system for determining the position of a lawn mower. The MATLAB function Single_Epoch_Position.m employs an iterative process to ascertain the antenna receiver's position at a specific time instant, leveraging pseudo-range data from satellites without necessitating prior knowledge of the receiver's location.

This procedure involves reading pseudorange data from the provided CSV file, computing ECEF positions and velocities for the satellites involved, and employing these calculations to refine the receiver's position via the unweighted least squares method. The iteration persists until the improvement in the receiver's positional accuracy diminishes to less than 10 cm in the x, y, and z directions. Incorporating Sagnac correction, the algorithm updates both the position estimate and clock deviation with each iteration, striving for the requisite precision. The method is designed to enhance efficiency and numerical stability, incorporating optimizations to circumvent direct matrix inversions and validating input accuracy.

Moreover, the initial iteration includes a check for outliers in the range measurements from the lawn mower to the satellites, discarding any detected outliers and recomputing the position solution. Detailed algorithm and equations can be found in Workshop 1 Task 3 with $T = 6$. It should be noted that **no outliers** were flagged for epoch 0 pseudo range measurements. Fig 3.2 shows logical values for Workshop 1 Task 3 Eq. 8 where a value of 0 signifies measurment not flagged as an outlier.



```
sat 5 |V_1|= 2.69 >  sqrt(C_11)*T= 24.27 ----> 0
sat 6 |V_2|= 4.62 >  sqrt(C_22)*T= 23.15 ----> 0
sat 7 |V_3|= 4.81 >  sqrt(C_33)*T= 17.98 ----> 0
sat 9 |V_4|= 4.31 >  sqrt(C_44)*T= 15.43 ----> 0
sat 10 |V_5|= 2.79 >  sqrt(C_55)*T= 22.63 ----> 0
sat 11 |V_6|= 1.85 >  sqrt(C_66)*T= 23.49 ----> 0
sat 15 |V_7|= 2.24 >  sqrt(C_77)*T= 24.25 ----> 0
sat 30 |V_8|= 1.74 >  sqrt(C_88)*T= 16.19 ----> 0
```

Figure 3.2: Results from Outlier detection

Detailed equations and algorithm for this part can be found in Workshop 1 Task 1B and 3

3.2. VELOCITY IN EPOCH 0

The function Single_Epoch_Velocity.m calculates velocities in x, y, z directions in ECEF frame for a given epoch using unweighted least squared error methodology. It reads satellite pseudo range rate data from a CSV file, then calculates the positions and velocities of all satellites for the given time.

Utilizing the calculated position in Epoch 0, it computes line-of-sight vectors and predicted range rates to satellites. Through an iterative process, it adjusts the predicted state vector, comprising velocities and clock drift, to minimize the difference between measured and predicted satellite range rates. The iteration continues until changes in the estimated velocities fall below a 0.01 m/s threshold in all 3 axes, ensuring precise velocity determination.

Detailed equations and algorithm for this part can be found in Workshop 1 Task 4

3.3. POSITION AND VELOCITY AFTER EPOCH 0

The position as well as the velocity solutions at Epoch 0 are amalgamated into a eight-column vector, with the initial three entries representing the positional coordinates (x, y, z), the subsequent three detailing the velocities (V_x, V_y, V_z) as and a single entry for both of clock offset and clock drift as provided in the vector formula 13 in task 2A . This vector serves as the initial state estimator for the Kalman Filter. Additionally an intial estimate of the error covariance matrix is also set using the pre-defined constants. This initialisation bit is done in Initialise_GNSS_KF.m

This state estimator, along with pseudo measurement inputs for each epoch, drives the Kalman Filter through cycles of prediction, measurement update, and state correction. The filter forecasts the system's future state, adjusts this forecast based on incoming measurements, and corrects the state estimate to minimize error covariance. This process produces accurate, real-time position and velocity estimates, essential for precision navigation and tracking. The Kalman Filter's efficiency in handling measurement noise and system dynamics ensures reliable performance in critical applications.

Detailed equations and algorithm for this part can be found in Workshop 2 Task 2A and 2B

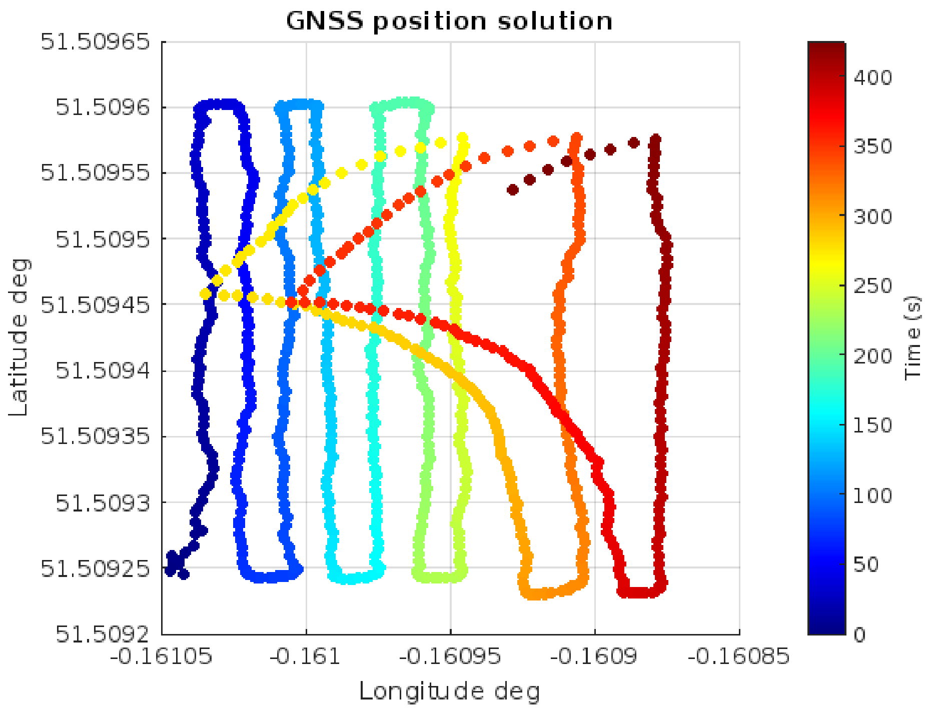


Figure 3.3: Change of GNSS position solutions [Lat, Long] over time

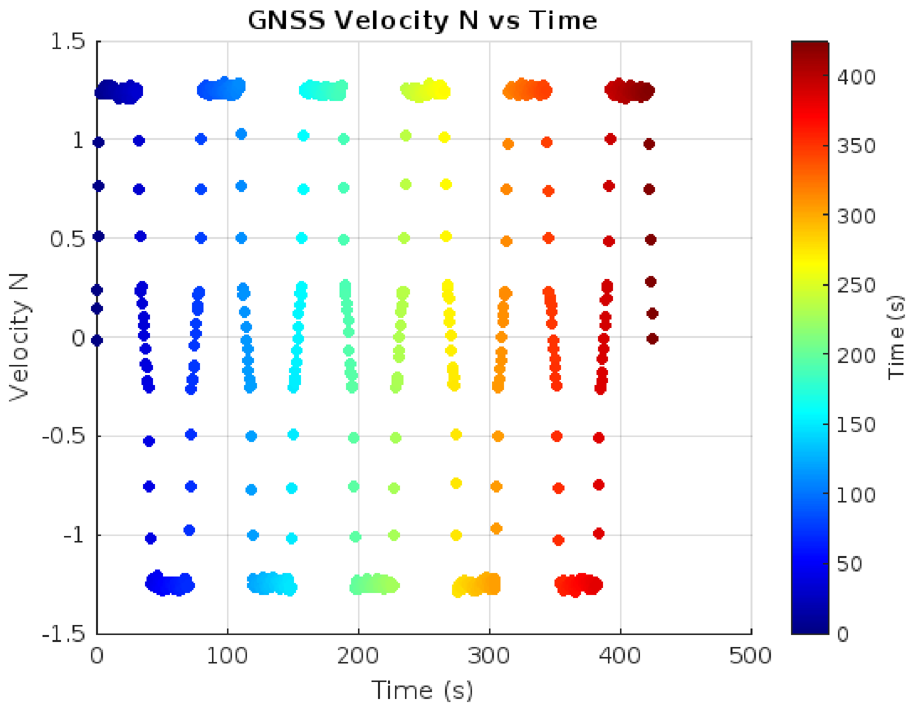


Figure 3.4: Change of Velocity North over time

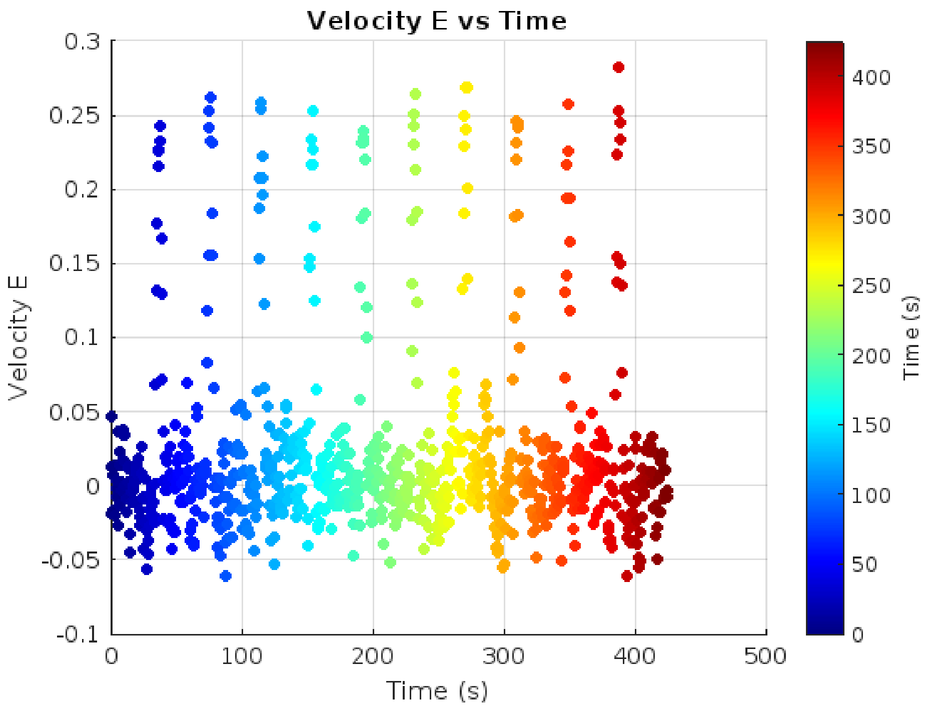


Figure 3.5: Change of Velocity East over time

4. INDEPENDENT DEAD RECKONING SOLUTION

4.1. PREPROCESSING

The provided MATLAB function, **correctHeadingWithGyroscopeKalman**, employs a Kalman Filter to refine heading estimates by fusing data from a gyroscope and a compass. It takes a matrix, ‘data‘, where gyroscope angular rates (in radians per second) and compass headings (in degrees) are columns 6 and 7, respectively. The function converts compass headings to radians for consistency, initializes state vectors for heading and heading change rate, and sets up covariance matrices reflecting measurement and process noise. Through iterative predictions and updates, it integrates gyroscope rates to anticipate heading changes, then corrects these predictions with compass measurements. The result is a series of corrected headings, adjusted for gyroscope drift and compass inaccuracies, converted back to degrees for practical interpretation. This approach allows for dynamically balanced, accurate heading estimation over time (correctHeadingWithGyroscopeKalman in Appendix).

4.2. DEAD RECKONING SOLUTION CALCULATION

This section of the reports highlights the Dead Reckoning integration for the purpose of calculating the lawnmower’s DR positioning and velocity. Dead reckoning technique was implemented in this project in order to estimate the lawnmower’s new position (latitude, longitude, height) and velocity (north, east and down) for all epochs, based on previously measured GNSS data, throughout a continuous integration of the velocity measurements [2].

After initialising the known GNSS position and velocity of the lawnmower, the estimated position gets updated in each epoch, using the following equations:

$$L_k = L_{k-1} + \frac{\overline{V}_{N,k}(t_k - t_{k-1})}{R_N + h}$$

$$\lambda_k = \lambda_{k+1} + \frac{\overline{V}_{E,k}(t_k - t_{k-1})}{(R_E + h)\cos L_k}$$

Where

- L_k Latitude at current epoch in radians
- λ_k Longitude at previous epoch in radians
- R_N Earth's meridian radius of curvature
- R_E Earth's transverse radius of the curvature
- h Height at current epoch (estimated through GNSS)
- $\overline{V}_{N,k}$ North velocity of lawnmower in NED coordinate system
- $\overline{V}_{E,k}$ East velocity of lawnmower in NED coordinate system

Subsequently, the north, east and down velocities also gets updated for the next epoch by assigning the received GNSS velocity measurements to the corresponding velocity component.

Detailed equations and algorithm for this part can be found in Workshop 3

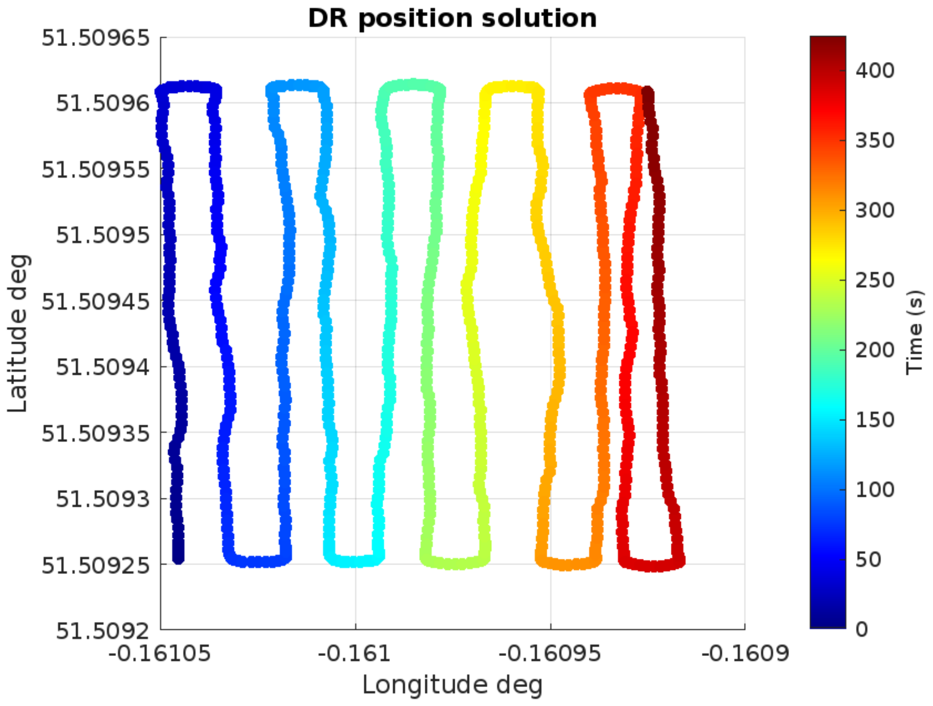


Figure 4.1: Change of DR position solution over time

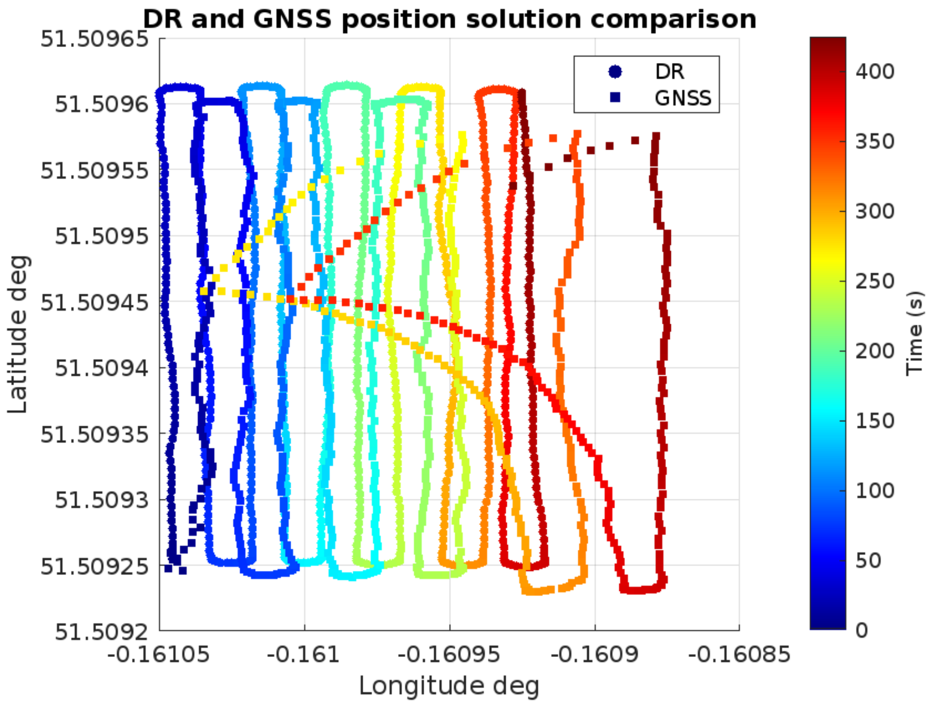


Figure 4.2: Comparing DR and GNSS position solutions over time

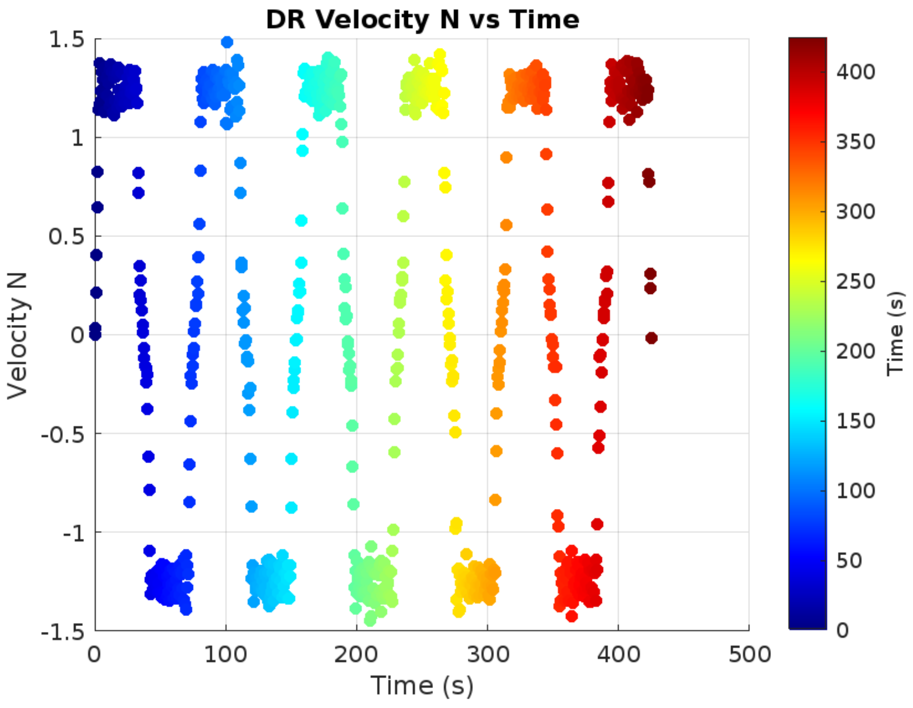


Figure 4.3: Change in DR Velocity North over time

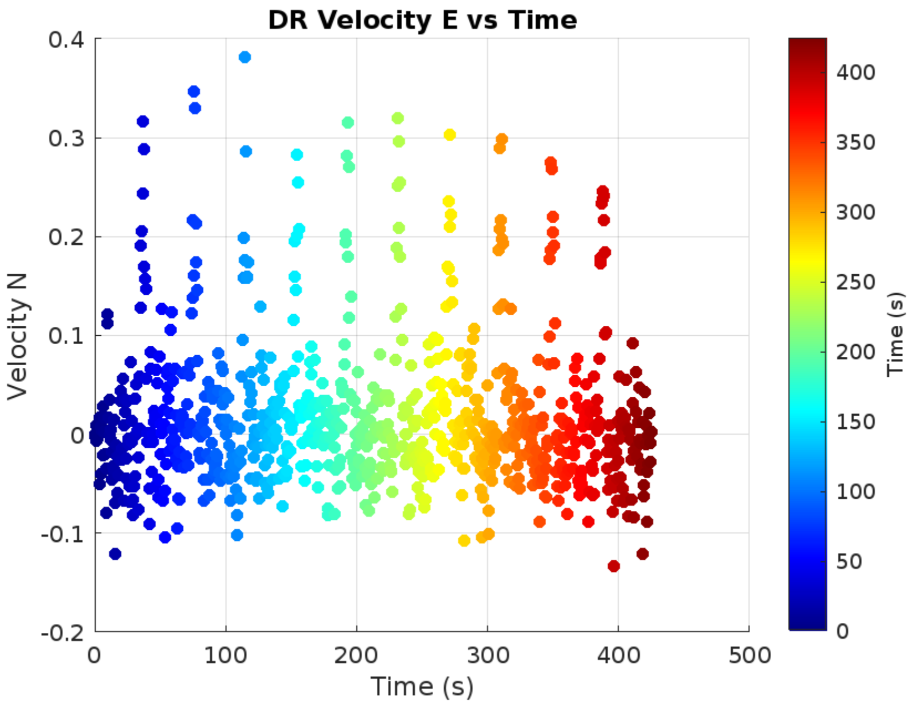


Figure 4.4: Change in DR Velocity East over time

5. INTEGRATING GNSS AND DR

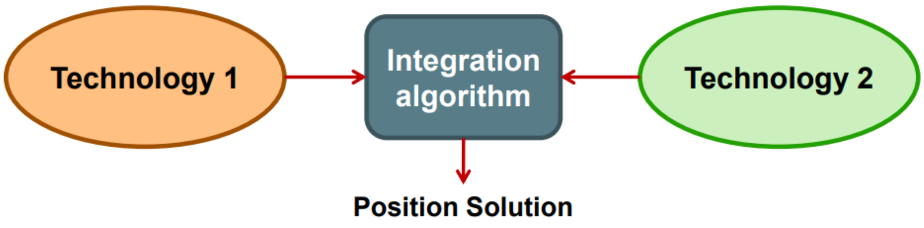


Figure 5.1: Integrating solutions from 2 different technlogies (Week3 Lecture 3B p. 4)

To date, we have obtained position and velocity solutions from two distinct systems. Our next step involves implementing a 4-state Kalman filter to estimate errors in north and east dead reckoning (DR) velocity, as well as DR latitude and longitude errors. Comprehensive equations and the algorithm for this phase are detailed in Workshop 3 Task 2

The state vector is defined as:

$$\mathbf{x} = \begin{bmatrix} \delta V_N \\ \delta V_E \\ \delta L \\ \delta \lambda \end{bmatrix}$$

Similarly a 4x4 matrix is defined for the state estimation error covariance matrix.

5.1. INITIALISATION AND CALCULATION AT EPOCH 0

The state vector is initialised with all zeros and the state estimation error covariance matrix is initialised as

$$P_0^+ = \begin{bmatrix} \sigma_v^2 & 0 & 0 & 0 \\ 0 & \sigma_v^2 & 0 & 0 \\ 0 & 0 & \frac{\sigma_r^2}{(R_N+h_o)^2} & 0 \\ 0 & 0 & 0 & \frac{\sigma_r^2}{(R_E+h_0)^2 \cos^2 L_o} \end{bmatrix}$$

Assuming the position measurements proceed the velocity measurements we apply Kalman filtering and estimate the **corrected DR solution**. The value of the estimated state vector and estimated error covariance matrix at Epoch 0 is treated as the predicted state vector and predicted error covariance matrix at Epoch 1.

5.2. EPOCH 1 AND LATER

The estimated state vector and estimated error covariance matrix from previous Epoch are treated as the predicted state vector and predicted error covariance matrix for current Epoch. A 10 step Kalman integration algorithm is applied to estimate corrected DR solution for velocity and position following Workshop 3 Task 2.

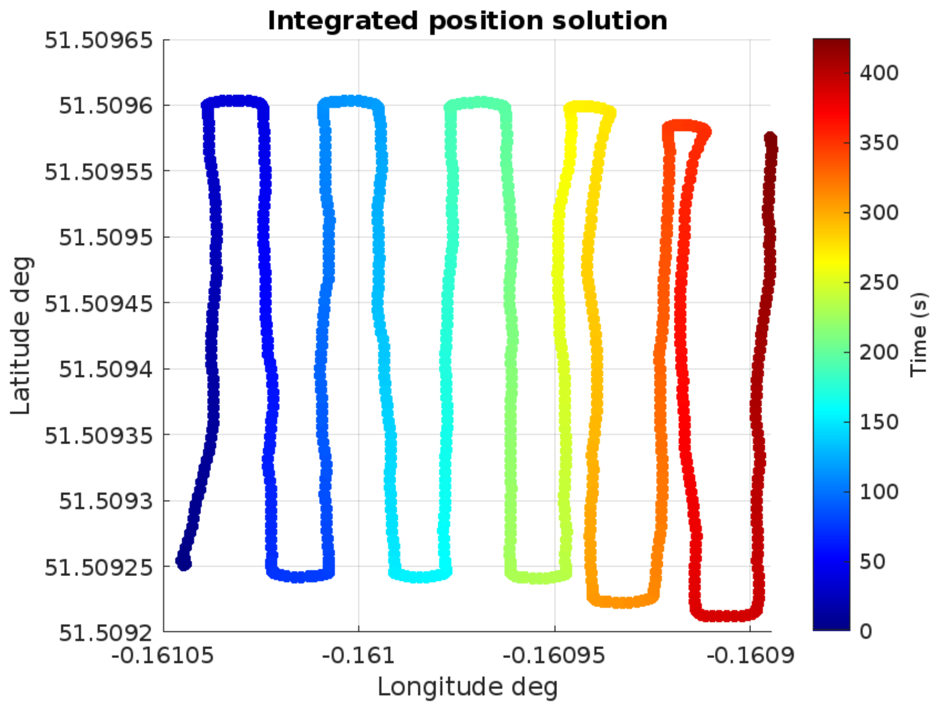


Figure 5.2: Integrated position solution graph

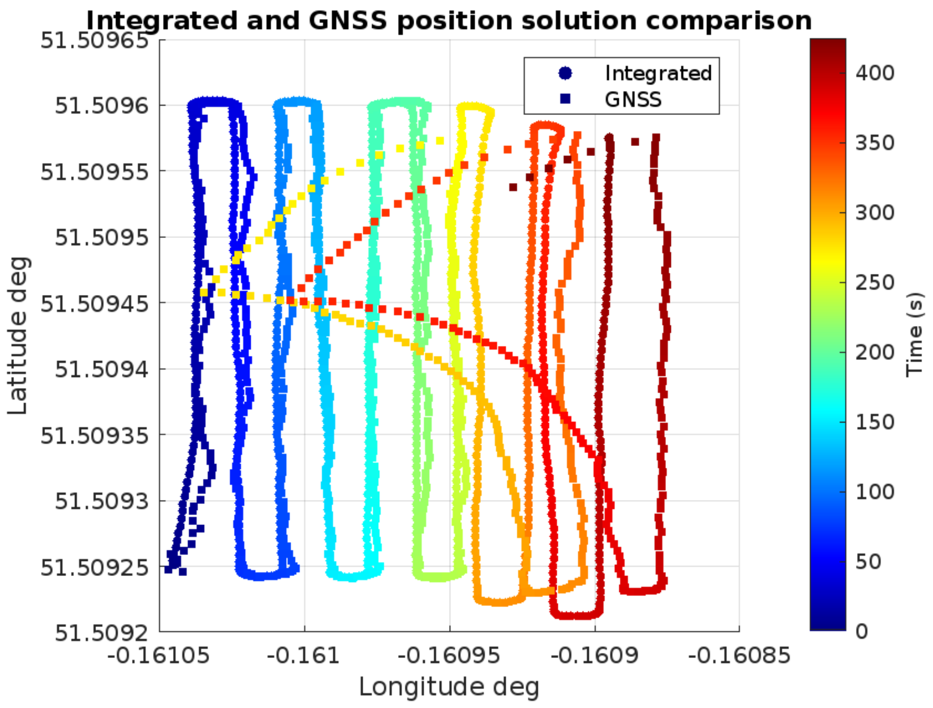


Figure 5.3: Comparing the Integrated position solution with the GNSS position solution

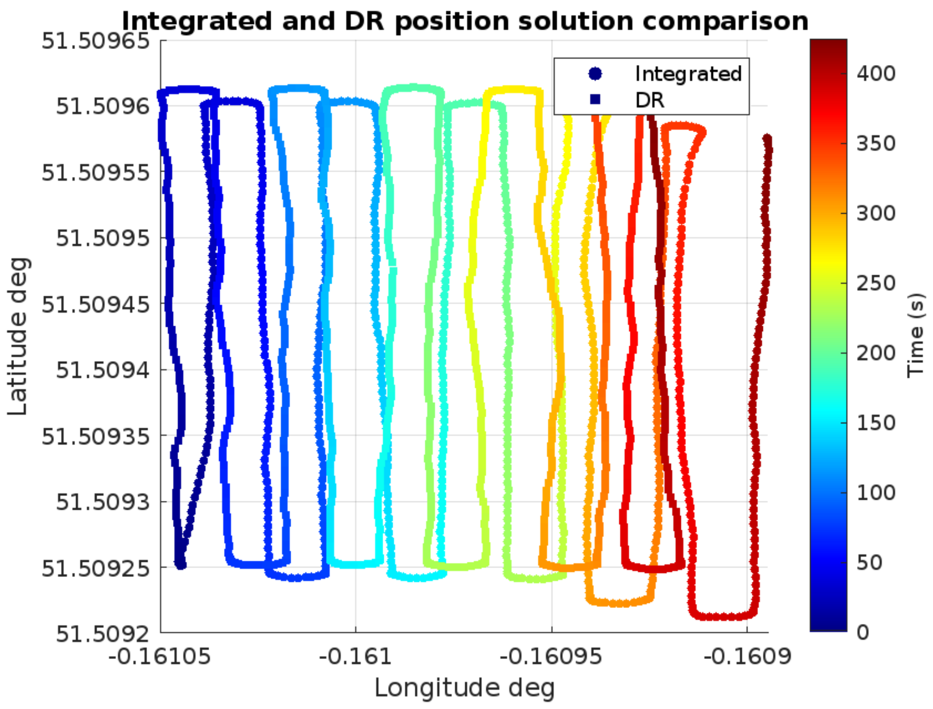


Figure 5.4: Comparing the Integrated position solution with the DR position solution

APPENDIX

Listing 5.1: Estimating antenna postition for a single epoch

```
function [est_state] = Single_Epoch_position(time, ...
    pred_pos_e_ea, ~, sat_index, obs_prange_a_sati)
%
%
% Function to get single epoch position solution with OR without any
   priors
% Iterates until position solution does not improve by more than 10cm
    in x y z axis in ECEF coordinate system
%
%
%
% Inputs :
% time : time in seconds (1, 1)
% pred_pos_e_ea : prior position of the antenna. If not available
   then coordinates for centre of the Earth is passed (3, 1)
% ~ : velocity vector not required (3, 1)
% sat_index : index number of satellites (n,) where n is the number
   of satellites
% obs_prange_a_sati : observed pseudo range measurements from all
   satellites (n, 1)
%
%
%
% Returns
% est_state : position of the antenna in ECEF coordinate system and
   clock
% offset
Define_Constants
pred_clock_offset= 1;
sat_count= numel(sat_index);
% total number of satellites
est_pos_e_esati= zeros(sat_count, 3);
% position of all satellites in ECEF
est_vel_e_esati= zeros(sat_count, 3);
% Velocity of all satellites in ECEF
for index = 1: sat_count
    [est_pos_e_esati(index, :) , est_vel_e_esati(index, :)]= ...
        Satellite_position_and_velocity(time, sat_index(index));
end
while 1>0
% infinite loop unless be break out
    pred_range_asati= zeros(sat_count, 1);
    % predicted pseudo ranges from current estimate of antenna
       position and
    % satellites
    u_e_asati= zeros(sat_count, 3);
    % line-of-sight unit vector from current estimate of antenna
       position and
    % satellites
    for index = 1: numel(sat_index)
        [pred_range_asati(index, 1), u_e_asati(index, :)]= ...
            line_of_sight_vector(est_pos_e_esati(index, :)',
               pred_pos_e_ea);
    end
    pred_state= [pred_pos_e_ea; pred_clock_offset];
    % predicted state vector [ pos_x, pos_y pos_z clock offset]
    pred_meas_innov= obs_prange_a_sati- pred_range_asati -
       pred_clock_offset;
    % predicted measurement innovation vector
    meas_mat= horzcat(-u_e_asati, ones( sat_count, 1));
    % measurement matrix
    A= meas_mat'* meas_mat;
    B= inv(A)* meas_mat';
    C= B* pred_meas_innov;
    est_state= pred_state + C;
    % estimated state using unweighted least-squares
    x_change= abs(est_state(1)- pred_state(1));
    y_change= abs(est_state(2)- pred_state(2));
    z_change= abs(est_state(3)- pred_state(3));
    % absolute value of change in position of antenna in ECEF
    resultx = x_change < 0.01;
    resulty = y_change < 0.01;
    resultz = z_change < 0.01;
    % checking if absolute change in position in all axes is less
       than 10cm
    if resultx==1 && resulty==1 && resultz==1
    % If the estimated position solution has not moved by more than
       10cm
    % in any axis we break out and return the current state estimates
        break
    end
    pred_pos_e_ea= est_state(1:3, 1);
    pred_clock_offset= est_state(4, 1);
    % setting estimated values as predicted values that will be used
       in
    % next iteration
end
```

Listing 5.2: Correct Heading using the Gyroscope

```
function correctedHeading = correctHeadingWithGyroscopeKalman(data)
    % Extract relevant data
    gyroRates = data(:, 6);  % Gyroscope angular rate in rad/s
    compassHeadings = deg2rad(data(:, 7)); % Compass headings in
       radians

    % Constants
    deltaTime = 0.5;  % Time step in seconds, adjust based on your
       data sampling rate
    gyro_error_variance = 3 * 10^(-6);  % Gyro measurement error
       variance in rad^2/s^3
    compass_noise_variance = 10^(-8); % Compass measurement noise
       variance in rad^2

    % Initialize state and covariance matrix
    states = [compassHeadings(1); 0]; % Initial heading and rate of
       change of heading (rad/s)
    P = [compass_noise_variance, 0;
         0, gyro_error_variance]; % Initial error covariance matrix

    % System noise covariance matrix (Q)
    Q = [1/4 * deltaTime^4, 1/2 * deltaTime^3;
         1/2 * deltaTime^3, deltaTime^2] * gyro_error_variance;

    % Measurement matrix (H) for heading update
    H = [1, 0];

    % Measurement noise covariance matrix (R)
    R = compass_noise_variance;

    % Allocate space for correctedHeading
    correctedHeading = zeros(size(compassHeadings));
    correctedHeading(1) = compassHeadings(1); % Set initial corrected
        heading

    for i = 2:length(compassHeadings)
        % Prediction using gyro rates
        phi = [1, deltaTime; 0, 1]; % State transition model
```

```
        gyroRate = gyroRates(i-1); % Gyro rate at previous timestep
        predictedRateOfChange = gyroRate; % Assuming gyroRate is the
           rate of change of heading
        states = phi * states + [0; predictedRateOfChange]; % Predict
            next state with gyro adjustment
        P = phi * P * phi' + Q; % Predict next covariance

        % Update using compass heading
        K = P * H' / (H * P * H' + R); % Kalman gain
        z = compassHeadings(i); % New measurement from compass
        y = z - H * states; % Measurement residual
        states = states + K * y; % Update state estimate with compass
            measurement
        P = (eye(2) - K * H) * P; % Update covariance estimate

        correctedHeading(i) = states(1); % Store corrected heading
    end

    % Convert corrected headings back to degrees
    correctedHeading = rad2deg(correctedHeading);
end
```

Listing 5.3: Kalman Filter for GNSS

```
function [est_state_new, est_error_cov_mat_new]= ...
    GNSS_Kalman_filter(est_state_last, est_error_cov_mat_last,...
    sat_index, sat_count, simulation_time, tau_s,...
    meas_range_asati, meas_range_rate_asati)

% Function to implement GNSS based Kalman filter
% Incorporates the pseudo range and pseudo range rate measurements
   from all satellites
%
% Inputs :
% est_state_last : estimated state vector of k-1 epoch
% est_error_cov_mat_last : estimated error covariance matrix of k-1
```

```
   epoch
% sat_index : list of index of all satellites from which measurements
    are available
% sat_count : lenght of sat_index
% simulation_time : time in seconds (required to get correct position
    and velocity of all satellites)
% tau_s : propogation interval in seconds (required to compute the
   transition and system noise covariance matrix
% meas_range_asati : pseudo range measurements from all satellites
% meas_range_rate_asati : pseudo range rate measurements from all
   satellites
%
%
% Returns:
% est_state_new : estimated state vecor for epoch k
% est_error_cov_mat_new : estimated error covariance matrix at epoch
   k


if size(meas_range_asati, 1) > size(meas_range_asati, 2)
    % no. of rows > no. of col
    % col vector
    % NOT compatible with the rest of the code
    meas_range_asati = meas_range_asati';
end

if size(meas_range_rate_asati, 1) > size(meas_range_rate_asati, 2)
    % no. of rows > no. of col
    % col vector
    % NOT compatible with the rest of the code
    meas_range_rate_asati = meas_range_rate_asati';
end
% if ~exist('prop_time','var')
%       prop_time = 0.5;
% end




% Only some constatns from Define_Constants.m are needed
omega_ie = 7.292115E-5;  % Earth rotation rate in rad/s
Omega_ie = Skew_symmetric([0,0,omega_ie]);
c = 299792458; % Speed of light in m/s




% Kalman filter Step 1)
% Transisition matrix last

trans_mat_last = eye(8, 8);


trans_mat_last(1:3, 4:6)= eye(3)* tau_s;
trans_mat_last(7, 8)= tau_s;




% Kalman filter Step 2)
% Compute system noise covariance matrix


acc_pow_spec_dens= 5; % can be added as function argument
```

```
clock_phase_dens= 0.01; % can be added as function argument
clock_freq_dens= 0.04; % can be added as function argument


noise_cov_mat_last= zeros(8 , 8);


noise_cov_mat_last(1:3, 1:3)= (1/3)*acc_pow_spec_dens*(tau_s^3)*eye
    (3);

noise_cov_mat_last(1:3, 4:6)= (1/2)*acc_pow_spec_dens*(tau_s^2)*eye
    (3);

noise_cov_mat_last(4:6, 1:3)= (1/2)*acc_pow_spec_dens*(tau_s^2)*eye
    (3);

noise_cov_mat_last(4:6, 4:6)= acc_pow_spec_dens*tau_s*eye(3);

noise_cov_mat_last(7, 7) = clock_phase_dens*tau_s + (1/3)*
    clock_freq_dens*(tau_s^3) ;
noise_cov_mat_last(7, 8) = (1/2)*clock_freq_dens*(tau_s^2);
noise_cov_mat_last(8, 7) = (1/2)*clock_freq_dens*(tau_s^2);
noise_cov_mat_last(8, 8) = clock_freq_dens*tau_s;




% Step 3)
% propogate state estimates pred_state_new using transisiton matrix

pred_state_new= trans_mat_last* est_state_last;
```

```
% Step 4)
% pred_error_cov_mat_new
% propogate the error covariance matrix

pred_error_cov_mat_new= trans_mat_last* est_error_cov_mat_last*
   trans_mat_last' + noise_cov_mat_last;



% Getting position and velocity of all satellites

pred_r_esati= zeros(sat_count, 3);
pred_v_esati= zeros(sat_count, 3);


for index = 1: numel(sat_index)
    [pred_r_esati(index, :) , pred_v_esati(index, :)]= ...
    Satellite_position_and_velocity(simulation_time, sat_index(index)
       );
end


% Compute psudo ranges and line-of-sight unit vector from antenna to
   all satellites

pred_range_asati= zeros(sat_count, 1);
pred_u_asati= zeros(sat_count, 3);

for index=1:sat_count
   [pred_range_asati(index, 1), pred_u_asati(index, :)]= ...
       line_of_sight_vector(pred_r_esati(index, :)', pred_state_new
          (1:3, 1));

end
```

```
% Compute range rates from approx antenna position to satellite

pred_range_rate_asati= zeros(sat_count, 1);

for index=1:sat_count

    C_I_e= eye(3); % (3, 3)
    C_I_e(1, 2)= omega_ie*pred_range_asati(index, 1)/c;
    C_I_e(2, 1)= -omega_ie*pred_range_asati(index, 1)/c;

    A= pred_v_esati(index, :)'+ Omega_ie*pred_r_esati(index, :)'; %
       (3, 1)

    B= pred_state_new(4:6, 1)+ Omega_ie* pred_state_new(1:3, 1); %
       (3, 1)

    pred_range_rate_asati(index, 1)= pred_u_asati(index, :)*((C_I_e*
       A)- B);
end



% Step 5) Compute measurement matrix

meas_mat_new = zeros(sat_count*2, 8);

meas_mat_new(1:sat_count, 1:3)= -pred_u_asati;
meas_mat_new(1:sat_count, 7)= 1;

meas_mat_new(sat_count+1: sat_count*2, 4:6)= -pred_u_asati;
meas_mat_new(sat_count+1: sat_count*2, 8)= 1;
```

```
% Step 6)
%  Compute measurement noise covariance matrix

pseudo_range_sd= 10;
pseudo_range_rate_sd= 0.05;


meas_noise_cov_mat_new= eye(sat_count*2, sat_count*2);

meas_noise_cov_mat_new(1: sat_count, 1: sat_count)= eye(sat_count,
    sat_count)* pseudo_range_sd^2;

meas_noise_cov_mat_new(sat_count+1: 2*sat_count, sat_count+1: 2*
    sat_count)= ...
     eye(sat_count, sat_count)* pseudo_range_rate_sd^2;


% Step 7)
% Compute kalman gain matrix
%  Kalman_gain_mat_new

A= meas_mat_new* pred_error_cov_mat_new * meas_mat_new';

B= A+ meas_noise_cov_mat_new;

C= pinv(B);



kal_gain_mat_new = pred_error_cov_mat_new* meas_mat_new' * C;
```

```
% Step 8)
% Predicted measurement innovation vector pred_meas_inov_new
pred_meas_inov_new= zeros(sat_count*2, 1);

pred_meas_inov_new(1: sat_count, 1)= meas_range_asati'-
   pred_range_asati- pred_state_new(7, 1);
pred_meas_inov_new(sat_count+1: 2*sat_count, 1)= ...
    meas_range_rate_asati'- pred_range_rate_asati- pred_state_new(8,
       1);


% Step 9)
%  Update state vector

est_state_new= pred_state_new+ kal_gain_mat_new* pred_meas_inov_new;


% Step 10)
%  Update error covariance matrix

A= kal_gain_mat_new* meas_mat_new ;
B= eye(size(A, 1));

est_error_cov_mat_new = (B- A)* pred_error_cov_mat_new;


end
```

Listing 5.4: Initialise GNSS

```
function [x_est,P_matrix] = Initialise_GNSS_KF_CW
%Initialise_GNSS_KF - Initializes the GNSS EKF state estimates and
   error
%covariance matrix for Workshop 2
```

```
%
% This function created 30/11/2016 by Paul Groves
%
% Outputs:
%   x_est                   Kalman filter 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

% Copyright 2016, Paul Groves
% License: BSD; see license.txt for details

% Begins

% Initialise state estimates
x_est = [  3.977851131656157e+06; -1.118086981210384e+04;
    4.969033742600406e+06 ;...% positon
     0.006799278282820; 0.047183610241202;  -0.017330817394829;...%
        velocity
     1.000880095199524e+04; 1.000123511027067e+02]; % clock offset ,
        clock drift

% Initialise error covariance matrix
P_matrix =  zeros(8);
P_matrix(1,1) = 10^2;
P_matrix(2,2) = 10^2;
P_matrix(3,3) = 10^2;
P_matrix(4,4) = 0.05^2;
P_matrix(5,5) = 0.05^2;
P_matrix(6,6) = 0.05^2;
P_matrix(7,7) = 100000^2;
P_matrix(8,8) = 200^2;
```

```
% Ends
```

Listing 5.5: Integrate GNSS and DR

```
function [corr_DR_speed_N_d, corr_DR_speed_E_d, corr_DR_lat,
    corr_DR_long,...
     est_state_new, est_state_err_cov_mat_new]=...
     Integrate_GNSS_DR_soln(est_state_last, est_state_err_cov_mat_last
        ,...
     GNSS_lat_rad, GNSS_long_rad, GNSS_height, GNSS_vel_N, GNSS_vel_E
        ,...
     DR_lat_rad, DR_long_rad, DR_vel_N_d, DR_vel_E_d,...
     prop_time, S_DR, GNSS_pos_sd, GNSS_vel_sd)

% Function to integrate GNSS and DR solutions derived independently
%
% Inputs :
% est_state_new : estimated state vector
% est_state_err_cov_mat_last : estimated state error covariance
    matrix
%
% GNSS_lat_rad : GNSS derived latitude in radians
% GNSS_long_rad : GNSS derived longitude in radians
% GNSS_heights : GNSS derived height in m
% GNSS_vel_N : GNSS derived velocity in N direction
% GNSS_vel_E : GNSS derived velocity in E direction
%
% DR_lat_rad : DR latitude solution in radians
% DR_long_rad : DR longitude solution in radians
% DR_vel_N_d : DR velocity (damped) in N direction
% DR_vel_E_d : DR velocity (damped) in E direction
%
%
% Optional inputs
%
```

```
% prop_time : propogation time (default value in 0.5 s)
% S_DR : Power spectral density (default value in 0.2)
% GNSS_pos_sd : sd of GNSS position solution
% GNSS_vel_sd : sd of GNSS velocity solution
%
%
% Outputs :
% corr_DR_speed_N_d : Corrected DR speed in N direction
% corr_DR_speed_E_d : Corrected DR speed in E direction
% corr_DR_lat : corrected DR latitude in radians !!!
% corr_DR_long : corrected DR longitude in radians !!!
% est_state_new : estimated state vector for current epoch
% est_state_err_cov_mat_last : estimated state error covariance
   matrix for currepnt epoch
%
%
if ~exist('prop_time','var')
      prop_time = 0.5;
end
if ~exist('S_DR','var')
      S_DR = 0.2;
end
if ~exist('GNSS_pos_sd','var')
      GNSS_pos_sd = 10;
end
if ~exist('GNSS_vel_sd','var')
      GNSS_vel_sd = 0.05;
end
% Kalman filter
% Step 1)
% define the transition matrix
[R_N, R_E]= Radii_of_curvature(GNSS_lat_rad);
h= GNSS_height;
trans_mat_last= eye(4);
```

```
trans_mat_last(3, 1)= prop_time/(R_N+ h );
trans_mat_last(4, 2)= prop_time/((R_E+ h )*cos(GNSS_lat_rad));
% Step 2)
% System error covariance matrix
sys_noise_cov_mat_last= zeros(4, 4);
sys_noise_cov_mat_last(1,1)= S_DR* prop_time;
sys_noise_cov_mat_last(2,2)= S_DR* prop_time;
sys_noise_cov_mat_last(3,3)= (1/3* S_DR* prop_time^3)/(R_N+ h)^2;
sys_noise_cov_mat_last(4,4)= (1/3* S_DR* prop_time^3) / ((R_E+ h) *
   cos(GNSS_lat_rad))^2;
sys_noise_cov_mat_last(3, 1)= (1/2 * S_DR* prop_time^2)/(R_N+ h);
sys_noise_cov_mat_last(1, 3)= (1/2 * S_DR* prop_time^2)/(R_N+ h);
sys_noise_cov_mat_last(4, 2)= (1/2 * S_DR* prop_time^2)/((R_E+ h) *
   cos(GNSS_lat_rad));
sys_noise_cov_mat_last(2, 4)= (1/2 * S_DR* prop_time^2)/((R_E+ h) *
   cos(GNSS_lat_rad));
% Step 3)
% Propogate state estimates
pred_state_new= trans_mat_last* est_state_last;
% Step 4)
% Propogate error covariance matrix
pred_state_err_cov_mat_new= trans_mat_last* ...
    est_state_err_cov_mat_last* trans_mat_last' +
       sys_noise_cov_mat_last;
% Step 5)
% Measuerment matrix
meas_mat_new= zeros(4, 4);
meas_mat_new(3:4, 1:2)= -eye(2, 2);
meas_mat_new(1:2, 3:4)= -eye(2, 2);
% Step 6)
% Measurement noise covariance matrix
meas_noise_cov_mat_new= zeros(4, 4);
meas_noise_cov_mat_new(1, 1)= GNSS_pos_sd^2 / (R_N+ h)^2;
meas_noise_cov_mat_new(2, 2)= GNSS_pos_sd^2 / ((R_E+ h)* cos(
   GNSS_lat_rad))^2;
meas_noise_cov_mat_new(3, 3)= GNSS_vel_sd^2;
meas_noise_cov_mat_new(4, 4)= GNSS_vel_sd^2;
% Step 7)
% Calculate Kalman gain matrix
A= meas_mat_new* pred_state_err_cov_mat_new * meas_mat_new';
B= A+ meas_noise_cov_mat_new;
C= inv(B);
kal_gain_mat_new = pred_state_err_cov_mat_new* meas_mat_new' * C;
% Step 8)
% Forumlate the measurement innovation
% a) Initialising with GNSS -DR values
pred_meas_inov_new= zeros(4, 1);
pred_meas_inov_new(1) = GNSS_lat_rad- DR_lat_rad;
pred_meas_inov_new(2) = GNSS_long_rad- DR_long_rad;
pred_meas_inov_new(3) = GNSS_vel_N- DR_vel_N_d;
pred_meas_inov_new(4) = GNSS_vel_E- DR_vel_E_d;
% c) subtracting the  measurement innov* state vector
pred_meas_inov_new= pred_meas_inov_new- (meas_mat_new* pred_state_new
   );
% Step 9) Update state estimates
est_state_new= pred_state_new+ kal_gain_mat_new* pred_meas_inov_new;
% Step 10) Update state error covariance matrix
est_state_err_cov_mat_new= (eye(4)- kal_gain_mat_new* meas_mat_new)*
   pred_state_err_cov_mat_new;
% Use Kalman filter estimates to correct the DR solution at each
   epoch
corr_DR_speed_N_d= DR_vel_N_d- est_state_new(1);
corr_DR_speed_E_d= DR_vel_E_d- est_state_new(2);
corr_DR_lat= DR_lat_rad- est_state_new(3);
corr_DR_long= DR_long_rad- est_state_new(4);
end
```

Listing 5.6: Line of Sight Vector Calculation

```
function [pred_range_asati, pred_u_asati] = line_of_sight_vector(
   pred_r_esati , ...
    pred_r_ea)
% Calculate range and line of sight vector unit vector
%
%
% Input :
% pred_r_esati predicted r vector from center of earth to satellite
   (3, 1)
% pred_r_ea predicted r vector from centre of earth to antenna (3, 1)
%
%
% Returns :
% pred_range_asati - predicted range from antenna to satellite (1, 1)
% pred_u_asati - line-of-sight unit vector (3, 1)
%
%


c = 299792458; % Speed of light in m/s
omega_ie = 7.292115E-5;  % Earth rotation rate in rad/s


C= eye(3);

pred_range_asati = sqrt((C*pred_r_esati - pred_r_ea)' * (C*
   pred_r_esati - pred_r_ea)); % (1, 1)

C(2, 1)= -omega_ie* pred_range_asati/c;
C(1, 2)= omega_ie* pred_range_asati/c;

pred_range_asati = sqrt((C*pred_r_esati - pred_r_ea)' * (C*
   pred_r_esati - pred_r_ea)); % (1, 1)
```

```
pred_u_asati= (C*pred_r_esati - pred_r_ea)/pred_range_asati ; %(3, 1)
    matrix

end
```

Listing 5.7: Single Epoch Position

```
function [est_state] = Single_Epoch_position(time, ...
    pred_pos_e_ea, ~, sat_index, obs_prange_a_sati)
%
%
% Function to get single epoch position solution with OR without any
   priors
% Iterates until position solution does not improve by more than 10cm
    in x y z axis in ECEF coordinate system
%
%
%
% Inputs :
% time : time in seconds (1, 1)
% pred_pos_e_ea : prior position of the antenna. If not available
   then coordinates for centre of the Earth is passed (3, 1)
% ~ : velocity vector not required (3, 1)
% sat_index : index number of satellites (n,) where n is the number
   of satellites
% obs_prange_a_sati : observed pseudo range measurements from all
   satellites (n, 1)
%
%
%
% Returns
% est_state : position of the antenna in ECEF coordinate system and
   clock
% offset
Define_Constants
pred_clock_offset= 1;
```

```
sat_count= numel(sat_index);
% total number of satellites
est_pos_e_esati= zeros(sat_count, 3);
% position of all satellites in ECEF
est_vel_e_esati= zeros(sat_count, 3);
% Velocity of all satellites in ECEF
for index = 1: sat_count
    [est_pos_e_esati(index, :) , est_vel_e_esati(index, :)]= ...
        Satellite_position_and_velocity(time, sat_index(index));
end
while 1>0
% infinite loop unless be break out
    pred_range_asati= zeros(sat_count, 1);
    % predicted pseudo ranges from current estimate of antenna
       position and
    % satellites
    u_e_asati= zeros(sat_count, 3);
    % line-of-sight unit vector from current estimate of antenna
       position and
    % satellites
    for index = 1: numel(sat_index)
        [pred_range_asati(index, 1), u_e_asati(index, :)]= ...
            line_of_sight_vector(est_pos_e_esati(index, :)',
               pred_pos_e_ea);
    end
    pred_state= [pred_pos_e_ea; pred_clock_offset];
    % predicted state vector [ pos_x, pos_y pos_z clock offset]
    pred_meas_innov= obs_prange_a_sati- pred_range_asati -
       pred_clock_offset;
    % predicted measurement innovation vector
    meas_mat= horzcat(-u_e_asati, ones( sat_count, 1));
    % measurement matrix
    A= meas_mat'* meas_mat;
    B= inv(A)* meas_mat';
    C= B* pred_meas_innov;
    est_state= pred_state + C;
    % estimated state using unweighted least-squares
    x_change= abs(est_state(1)- pred_state(1));
    y_change= abs(est_state(2)- pred_state(2));
    z_change= abs(est_state(3)- pred_state(3));
    % absolute value of change in position of antenna in ECEF
    resultx = x_change < 0.01;
    resulty = y_change < 0.01;
    resultz = z_change < 0.01;
    % checking if absolute change in position in all axes is less
       than 10cm
    if resultx==1 && resulty==1 && resultz==1
    % If the estimated position solution has not moved by more than
       10cm
    % in any axis we break out and return the current state estimates
        break
    end
    pred_pos_e_ea= est_state(1:3, 1);
    pred_clock_offset= est_state(4, 1);
    % setting estimated values as predicted values that will be used
       in
    % next iteration
end
```

Listing 5.8: Single Epoch Celocity

```
function [est_state]= Single_epoch_velocity(time, ...
    est_pos_e_ea, sat_index, osb_range_rate_asati)
%
% Function to estimate velocity of antenna using unweighted
% least squares estimation
%
% Inputs
% time : time in seconds (1, 1)
% est_pos_e_ea : estimated position on antenna in ECEF (3, 1)
% sat_index :index number of satellites (n,) where n is the number of
    satellites
% osb_range_rate_asati : observed pseudo range rate measurements from
    all satellites (n, 1)
%
% Returns
% est_state : velocity of the antenna and clock drift
%
Define_Constants;
% Getting position and velocity of all satellites
sat_count= numel(sat_index);
pred_r_esati= zeros(sat_count, 3);
% position of all satellites in ECEF
pred_v_esati= zeros(sat_count, 3);
% velocity of all satellites in ECEF
for index = 1: sat_count
    [pred_r_esati(index, :) , pred_v_esati(index, :)]= ...
    Satellite_position_and_velocity(time, sat_index(index));
end
% starting with zero velocities for the antenna
pred_v_ea=  zeros(1, 3);
clock_drift= 0;
% intial velocity in NED frame
[~, ~, ~, pred_vel_ea_resolved]= pv_ECEF_to_NED(est_pos_e_ea,
   pred_v_ea');
% predicted state vector
pred_state= zeros(4, 1);
pred_state(1:3, 1)= pred_v_ea';
pred_state(4, 1)= clock_drift;
pred_range_rate_asati= zeros(8, 1);
while 1>0

    pred_range_asati= zeros(sat_count, 1);
    % predicted pseudo ranges from current estimate of antenna
   position and
% satellites
pred_u_asati= zeros(sat_count, 3);
% line-of-sight unit vector from current estimate of antenna
   position and
% satellites
for index=1: sat_count
   [pred_range_asati(index, 1), pred_u_asati(index, :)]= ...
       line_of_sight_vector(pred_r_esati(index, :)', est_pos_e_ea
          );
end
for index = 1: sat_count
    % iterating over all satellites
    C_I_e= eye(3); % (3, 3)
    C_I_e(1, 2)= omega_ie*pred_range_asati(index, 1)/c;
    C_I_e(2, 1)= -omega_ie*pred_range_asati(index, 1)/c;
    A= pred_v_esati(index, :)'+ Omega_ie*pred_r_esati(index, :)';
        % (3, 1)
    B= pred_v_ea'+ Omega_ie* est_pos_e_ea; % (3, 1)
    pred_range_rate_asati(index, 1)= pred_u_asati(index, :)*((
       C_I_e* A)- B);
    % predicted range rate measurement for a satellite
end
% predicted measurement innovation vector
pred_meaus_innov= osb_range_rate_asati- pred_range_rate_asati-
   pred_state(4, 1);
% measurement matrix
H= ones(8, 4);
H(:, 1:3)= -pred_u_asati;
% estimated state using unweighted least-squares
est_state= pred_state+ inv(H'*H)*H'*pred_meaus_innov;
% resolving velocity in NED frame
[~, ~, ~, est_v_ea_resolved]= pv_ECEF_to_NED(est_pos_e_ea,
   est_state(1:3, 1));
```

```
    % absolute value of change in velocity of antenna in NED
    Delta_vx = abs(est_v_ea_resolved(1, 1)- pred_vel_ea_resolved(1,
       1));
    Delta_vy = abs(est_v_ea_resolved(2, 1)- pred_vel_ea_resolved(2,
       1));
    Delta_vz = abs(est_v_ea_resolved(3, ...
        1)- pred_vel_ea_resolved(3, 1));
    % checking if absolute change in velocity in all axes is less
       than 10cm/s
    if Delta_vx < 0.01 && Delta_vy < 0.01 && Delta_vz < 0.01
        break
    end
    % setting estimated values as predicted values that will be used
       in
    % next iteration
    pred_v_ea=  est_state(1:3, 1)';
    clock_drift= est_state(4, 1);
    pred_state= est_state;
    pred_vel_ea_resolved= est_v_ea_resolved;
end
end
```

BIBLIOGRAPHY

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

[2] M. Winton, Ask the quexperts: What is dead reckoning?, last accessed 07-Feb-2024.
URL `https://www.quectel.com/what-is-dead-reckoning-iot`