Localization Challenge

Hoomaan Moradimaryamnegari

December 13, 2023

Abstract

In this challenge, the objective is to develop a feature-based localization engine for determining the position of a mobile robot within a computer-simulated 2D soccer field. The robot's state is defined by a three-dimensional vector, encompassing both its position and orientation on the playing field. The primary aim is to implement a localization engine using C++ that accurately estimates the state of the robot at each time step. This work explores and details two distinct methods, namely Weighted Average and Kalman Filter, for effectively estimating the robot's position on the soccer field.

1 Introduction

In this report, we explore two different methods for estimating the states of a mobile robot: Weighted Average and Kalman Filter. The first method employs the inverse of the distance between the robot and markers as weights to calculate the Weighted Average of all observations. In the second method, the Kalman Filter is applied to fuse observations from both markers and the mobile robot's odometry, mitigating jumps caused by noise. Subsequent sections provide comprehensive explanations of these localization methods.

2 Weighted Average

A Weighted Average is a computation method that considers the varying degrees of importance associated with numbers in a data set. During the calculation of a weighted average, each number in the data set is multiplied by a predetermined weight before the final computation. This approach can yield a more precise result compared to a simple average, where all numbers in the data set are assigned an identical weight. To illustrate, consider two signals, S_{1_i} and S_{2_i} at time step i, each with variances V_{1_i} and V_{2_i}, respectively. Weights can be established based on the inverse of the variance. Consequently, the fusion results can be formulated as follows:

$$F_i = \frac{S_{1_i} V_{1_i}^{-1} + S_{2_i} V_{2_i}^{-1}}{V_{1_i}^{-1} + V_{2_i}^{-1}} \tag{1}$$

In localization problems, defining the distance between the mobile robot and the marker can be simplified and effectively represented as a time-varying variance. As a result, the Weighted Average for our systems, considering a maximum of four observations O from markers at a distance d from the robot, can be expressed as follows:

$$F_i = \frac{O_{1_i} d_{1_i}^{-1} + O_{2_i} O_{2_i}^{-1} + O_{3_i} d_{3_i}^{-1} + O_{4_i} O_{4_i}^{-1}}{d_{1_i}^{-1} + d_{2_i}^{-1} + d_{3_i}^{-1} + d_{4_i}^{-1}} \tag{2}$$

When at least one observation is available, the performance of the Weighted Average method is desirable. However, in instances where, for any reason, no marker observations are available at a given sampling time, relying solely on odometry can lead to a degradation in estimation performance. A viable solution involves initializing the odometry sensor with the result of the Weighted Average at each sampling time when marker observations are accessible. This approach ensures that in the case of lost marker observations, the odometry begins with the most recently estimated location obtained

through the Weighted Average method. Consequently, any drift in the odometry sensor is mitigated until the last sampling time when the latest marker observation is available.

To comprehensively assess the algorithm, the code functions are elucidated step by step:

Firstly, as depicted in the code below, it is imperative to initialize the robot state in both the robot body frame and the global frame. In terms of orientations, it is crucial to ensure that angles are expressed in radians and constrained within the range of $-\Pi$ to Π. To achieve this constraint, a function named *angLimit* has been defined, which takes the angle as a reference and maps it within the specified range of $-\Pi$ to Π. The implementation of this function is also provided below. Additionally, the positions of the markers are stored in the *landmarkLocations* variable for utilization in the *sensorUpdate()*.

```
void myinit(RobotState robotState, RobotParams robotParams,
            FieldLocation markerLocations[NUM_LANDMARKS])
{
    //Initialize the robot sates
    pose.x = robotState.x;
    pose.y = robotState.y;
    pose.theta = robotState.theta;
    poseGlob.x = robotState.x*cos(robotState.theta);
    poseGlob.y = robotState.y*sin(robotState.theta);

    // Assuming that the angles are given in degrees
    robotParams.angle_fov = robotParams.angle_fov * M_PI / 180;
    angLimit(robotParams.angle_fov);   //limit the angle within [-pi, pi]

    robotParams.odom_noise_rotation_from_rotation =
    robotParams.odom_noise_rotation_from_rotation * M_PI / 180;
    angLimit(robotParams.odom_noise_rotation_from_rotation);

    robotParams.odom_noise_rotation_from_translation =
    robotParams.odom_noise_rotation_from_translation * M_PI / 180;
    angLimit(robotParams.odom_noise_rotation_from_translation);

    robotParams.sensor_noise_orientation =
    robotParams.sensor_noise_orientation * M_PI / 180;
    angLimit(robotParams.sensor_noise_orientation);

    // Initialize marker locations with respect to origin
    for (size_t i = 0; i < NUM_LANDMARKS; i++) {
       landmarkLocations[i].x = markerLocations[i].x;
       landmarkLocations[i].y = markerLocations[i].y;
   };

}
```

```
// Function to limit the angle within [-pi, pi]
void angLimit(double& ang) {
  while (ang > M_PI) ang -= 2. * M_PI;
  while (ang < -M_PI) ang += 2. * M_PI;
}
```

In the subsequent phase, leveraging data from the odometry sensor, we proceed to update the position and orientation of the mobile robot. The function *motionUpdate(RobotState delta)* is designed to accumulate the displacement at each sampling time, subsequently updating the position and orientation (pose) from the previous sampling time. Due to the presence of constant noise, relying solely on this sensor for navigation can result in a drift in the robot's position.

Since the pose is computed by the odometry installed on the robot, the measurements are calculated in the robot body frame (noting that, as the robot moves forward in the $X.c$ direction, *delta.y* remains consistently zero). To determine the position of the robot in the global frame, a transformation from the robot body frame to the global frame is necessary, as illustrated in Figure 1. The equation for this transformation is as follows:

$$X_{global_{odo}} = X_{robot}.\cos(\theta) \tag{3}$$

$$Y_{global_{odo}} = Y_{robot}.\sin(\theta) \tag{4}$$

In the final line, we set the *flag* to *true*. If the *flag* remains *true*, it indicates a lack of access to any marker observations. Consequently, the estimation is solely reliant on the odometry results. However, it's important to note that the position of the robot has already been initialized with the last available observation. This preemptive initialization serves to mitigate the drift induced by the accumulated odometry in the preceding time steps.

```
1  void motionUpdate(RobotState Δ)
2  {
3
4      //Update the pose of the robot in the body frame of the robot originated at the ...
           inital Pose of the robot measured by odometry with sensor noise
5      pose.x += Δ.x + robotParams.odom_noise_translation_from_translation;
6      pose.y += Δ.y + robotParams.odom_noise_translation_from_translation;
7      pose.theta += Δ.theta + robotParams.odom_noise_rotation_from_rotation;
8
9      angLimit(pose.theta);
10
11     //Update the global Pose of the robot with respect to the origin measured by ...
           odometry
12     poseGlob.x = pose.x*cos(pose.theta);
13     poseGlob.y = pose.y*sin(pose.theta);
14
15     flag = true;    //To update the odometry with the last Weighted Average result
16
17    }
```

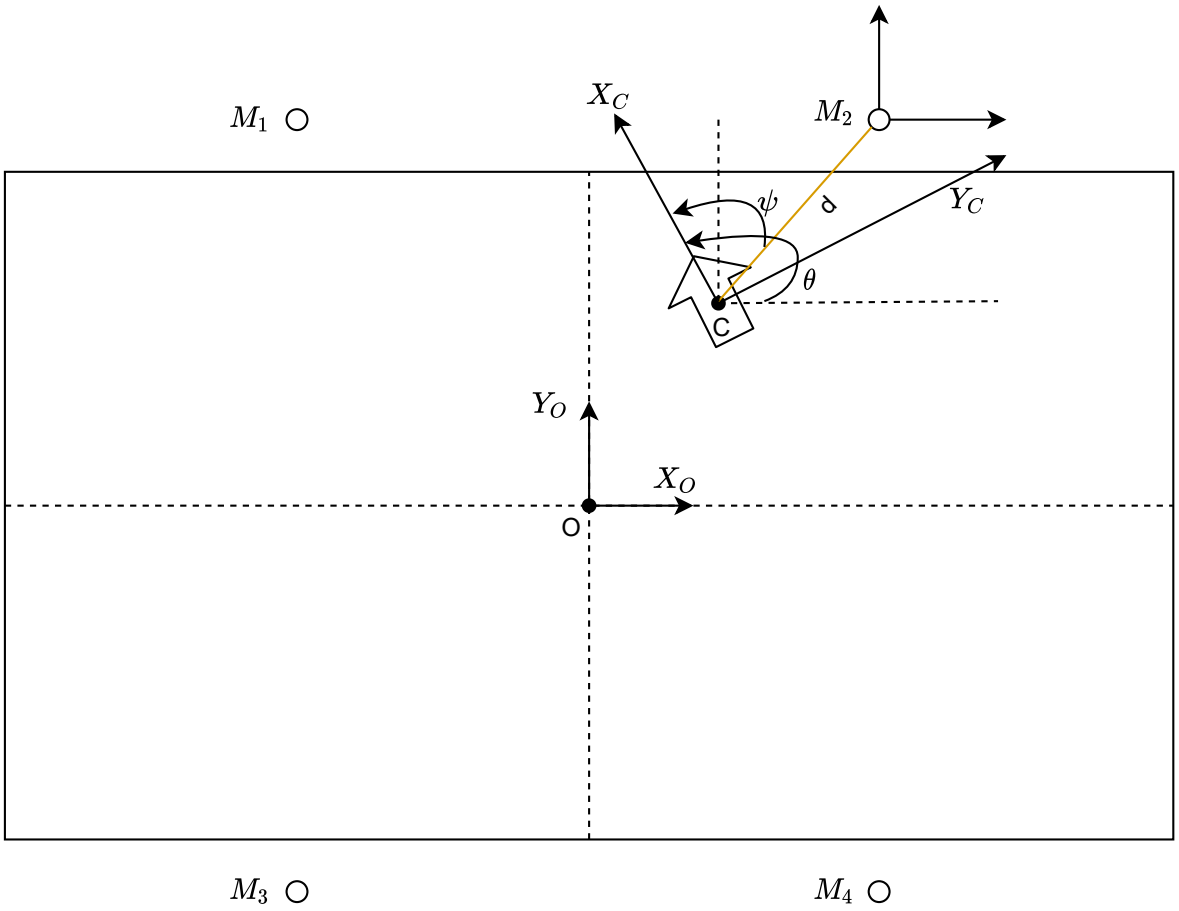

Figure 1: The schematic of positions and orientation of the mobile robot with respect to the origin and markers.

In the subsequent phase, the calculation of the global position of the robot based on observations from markers is performed in the *sensorUpdate()*, outlined below. Initially, noise is introduced to the *distance* (d) and *orientation* (ψ) variables, which are received as observations from each marker. Subsequently, the global position of the robot must be computed.

As the *distance* and *orientation* are provided in the robot frame, as illustrated in Fig.1, a transformation to the marker frame is necessary. To begin, we project the *distance* into the body frame of the robot, and the process is as follows:

$$XX_C = d.\cos(\psi) \tag{5}$$

$$YY_C = d.\sin(\psi) \tag{6}$$

Furthermore, another projection to the marker frame is carried out, and the procedure is as follows:

$$XX_M = XX_C.\cos(\theta) - YY_C.\sin(\theta) \tag{7}$$

$$YY_M = XX_C.\sin(\theta) + YY_C.\cos(\theta) \tag{8}$$

Then, as the position needs to be calculated with respect to the global origin, the transfer of the robot position from the marker origin to the global origin is executed as follows:

$$X_{global_{mark}} = X_M - XX_M \tag{9}$$

$$Y_{global_{mark}} = Y_M - YY_M \tag{10}$$

This approach enables us to obtain the robot position in the global frame through observations from each marker. Subsequently, we can proceed to apply the Weighted Average method to fuse the available observations. As previously mentioned, we calculate each weight based on the inverse of its observation distance using the *computeWeight()* function, as illustrated below. The cumulative weights for all observations are then computed to derive the Weighted Average based on Equation 2. When we have at least one observation from markers, the *flag* is set to *false* in the loop, signifying that the estimation is conducted based on the Weighted Average method. Additionally, in this scenario, the initial position of the odometry is set to the Weighted Average result to mitigate drift as long as we have access to at least one marker observation.

```
1  void sensorUpdate(std::vector<MarkerObservation> observations)
2  {
3      double totalWeight = 0.0;   //Sum of the weights
4
5      //Pose of the robot in global frame based on Average Weight result
6      weightedPose.x = 0.0;
7      weightedPose.y = 0.0;
8
9      // Loop through each observed marker to obtain the Weighted Average
10     for (const auto& observation : observations) {
11
12         //Add noise to the observations obtained by markes
13         markerOutput.distance  = observation.distance + ...
               robotParams.sensor_noise_distance;
14         markerOutput.orientation  = observation.orientation + ...
               robotParams.sensor_noise_orientation;
15
16         //Update the pose of the robot in the marker frame obtained by marker ...
               observations
17         markerPose.x = markerOutput.distance * ...
               cos(markerOutput.orientation)*cos(pose.theta)- markerOutput.distance * ...
               sin(markerOutput.orientation)*sin(pose.theta);
18         markerPose.y = markerOutput.distance * cos(markerOutput.orientation )* ...
               sin(pose.theta) + markerOutput.distance * sin(markerOutput.orientation ...
               )* cos(pose.theta);
19
20         //Update the pose of the robot in origin frame obtained by marker observations
21         markerPoseGlob.x = landmarkLocations[observation.markerIndex].x - markerPose.x;
22         markerPoseGlob.y = landmarkLocations[observation.markerIndex].y - markerPose.y;
23
```

```
        //Store the pose of the robot in the global frame obtained by marker ...
            observation for plots
        markerPoseGlobFig[observation.markerIndex].x = markerPoseGlob.x;
        markerPoseGlobFig[observation.markerIndex].y = markerPoseGlob.y;

        double weight = computeWeight(observation);   //Calculate the weight of each ...
            observation
        totalWeight += weight;     // Accumulate the weightes

        ////Calculate the weightd pose of each observation
        weightedPose.x += weight * markerPoseGlob.x;
        weightedPose.y += weight * markerPoseGlob.y;

        flag = false;   //To apply the Weighted Average to the estimation
    }

    if (totalWeight > 0.0) {

        // Update the estimation of the robot's pose based on weighted average
        weightedPose.x /= totalWeight;
        weightedPose.y /= totalWeight;

        // Update the robot's pose based on weighted average for the odometry
        poseGlob.x = weightedPose.x;
        poseGlob.y = weightedPose.y;
        pose.x = poseGlob.x/(cos(pose.theta)+0.00001);
        pose.y = poseGlob.y/(sin(pose.theta)+0.00001);
    }
}
```

```
double computeWeight(const MarkerObservation& observation) {
    const double minDistance = 0.000000000001; // Avoid division by zero
    return 1.0 / std::max(minDistance, observation.distance);
}
```

In the final section, *getRobotPositionEstimate()* is called to retrieve the ultimate estimation, as outlined below. As previously indicated, if the *flag* is set to *false*, the estimations are derived from the Weighted Average method. Conversely, if the *flag* is set to *true*, it signifies the absence of any marker observations. In such cases, the odometry output, initialized with the Weighted Average method in the previous sampling time, is employed to estimate the position of the robot.

```
void getRobotPositionEstimate(RobotState& estimatePosn)
{

  if (!flag) {                                  //if at least we can have one ...
      observation from the markers
  estimatePosn.x = weightedPose.x;
  estimatePosn.y = weightedPose.y;
  }
  else {                                        //if there is not any observation from ...
      the markers
  estimatePosn.x = poseGlob.x;
  estimatePosn.y = poseGlob.y;
  }
  estimatePosn.theta = pose.theta;
}
```

In *mydisplay()*, marker observations stored in *sensorUpdate()*, corrected odometry stored in *motionUpdate()*, and the output of the Weighted Average method stored in *sensorUpdate()* are saved in a text file, as follows. This enables them to be plotted later.

```
void mydisplay()
{
    // Write the snesor data as well as the Weighted Average result in a text file ...
        for plots
    std::ofstream outputFile("Figures.txt", std::ios::app);
```

```
    if (outputFile.is_open()) {
        // Write header line if the file is empty
        if (outputFile.tellp() == 0) {
            outputFile << "poseodom.x poseodom.y markerPose1.x markerPose1.y ...
                markerPose2.x markerPose2.y markerPose3.x markerPose3.y ...
                markerPose4.x markerPose4.y PoseAver.x PoseAver.y" << std::endl;
        }

        // Append varibles values
        outputFile << poseGlob.x << " " << poseGlob.y <<
                " " << markerPoseGlobFig[1].x << " " << markerPoseGlobFig[1].y <<
                " " << markerPoseGlobFig[2].x << " " << markerPoseGlobFig[2].y <<
                " " << markerPoseGlobFig[3].x << " " << markerPoseGlobFig[3].y <<
                " " << markerPoseGlobFig[4].x << " " << markerPoseGlobFig[4].y <<
                " " << weightedPose.x << " " << weightedPose.y << std::endl;

        outputFile.close();

    } else {
        std::cerr << "Unable to open file for writing!" << std::endl;
    }

}
```

The outputs of marker observations, the proposed Weighted Average method, and the Corrected Odometry are compared in Figure 2. It is evident that the Weighted Average method exhibits more desirable estimation performance compared to individual marker observations. Even in scenarios with some lost marker observations, the Weighted Average output functions effectively (when the value for the marker observation is consistently zero, it indicates the loss of that observation). Additionally, the Corrected Odometry output demonstrates a lack of drift, thanks to the updating strategy implemented by the Weighted Average. The sole drawback of this method is that drift accumulates if the absence of all observations persists for an extended duration.

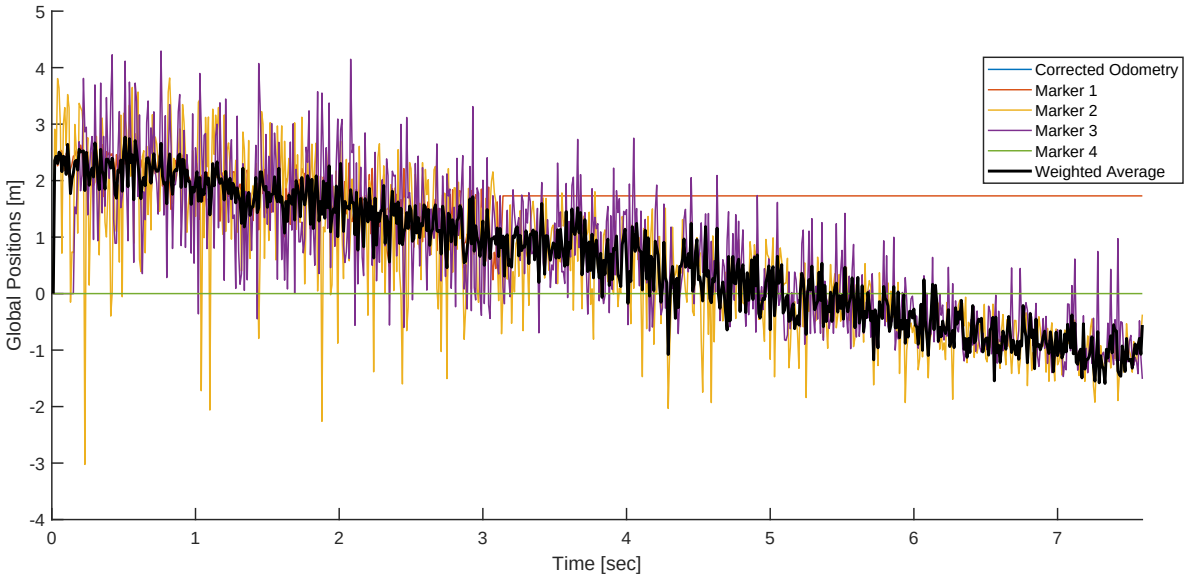

Figure 2: Output of the Markers observations, corrected odometry, and the proposed Weighted Average method.

3 Kalman Filter

Kalman filtering leverages a system's dynamic model, incorporating factors such as the physical laws of motion, known control inputs, and multiple sequential measurements from sensors. This integration allows the Kalman filter to generate an improved estimate of the system's varying quantities (its state)

compared to relying on a single measurement in isolation. Consequently, it serves as a widely employed algorithm for sensor fusion and data fusion.

The Kalman Filter adeptly addresses the challenges posed by the uncertainty inherent in noisy sensor data. By synthesizing the system's predicted state with a new measurement using a weighted average, the Kalman filter produces a state estimate that serves as a compromise between the predicted and measured states. Notably, the weights assigned to values are determined by the covariance, reflecting the estimated uncertainty of the system's state prediction. Values with lower estimated uncertainty receive higher trust, influencing the weighted average. The outcome is a refined state estimate that strikes a balance between the predicted and measured states, exhibiting a more accurate estimated uncertainty than either value in isolation. This process is repeated at every time step, with the newly derived estimate and its covariance guiding the subsequent prediction in the iterative cycle. The prediction equations are as follows:

1. Prediction Step

State Prediction:

$$\hat{x}_k^- = A \cdot \hat{x}_{k-1} + B \cdot u_k \tag{11}$$

Error Covariance Prediction:

$$P_k^- = A \cdot P_{k-1} \cdot A^T + Q \tag{12}$$

and the update step functions as follows:

2. Update Step

Kalman Gain Calculation:

$$K_k = P_k^- \cdot H^T \cdot (H \cdot P_k^- \cdot H^T + R)^{-1} \tag{13}$$

State Update:

$$\hat{x}_k = \hat{x}_k^- + K_k \cdot (z_k - H \cdot \hat{x}_k^-) \tag{14}$$

Error Covariance Update:

$$P_k = (I - K_k \cdot H) \cdot P_k^- \tag{15}$$

where $\hat{x}_k^-$ is the predicted state at time k, A is the state transition matrix, $\hat{x}_{k-1}$ is the previous state estimate, B is the control-input matrix (if control input u_k is present), u_k is the control input at time k, P_k^- is the predicted error covariance matrix at time k, Q is the process noise covariance matrix, K_k is the Kalman Gain at time k, H is the measurement matrix, R is the measurement noise covariance matrix, $\hat{x}_k$ is the updated state estimate, z_k is the measurement at time k, I is the identity matrix.

To illustrate how the Kalman Filter operates, the code is explained as follows:

In *myinit()*, the initialization process is carried out similarly to the Weighted Average method, with the addition of initializing the Kalman matrices in the last line. The constant matrices Q, R, A, and h, along with the covariance matrix P, are initialized using this function as outlined below:

```
1  void initializeMatrices() {
2
3    // Initialize Q, R, A, H, P matrices with appropriate values
4    Q << 0.4, 0, 0,
5         0, 0.4, 0,
6         0, 0, 0.01;
7
8    R << 0.1, 0,
9         0, 0.1;
10
11   A << 1, 0, 0,
12        0, 1, 0,
13        0, 0, 1;
```

```
14
15    H << 1, 0, 0,
16         0, 1, 0;
17
18    P << 1, 0, 0,
19         0, 1, 0,
20         0, 0, 1;
21  }
```

In the subsequent phase within the *motionUpdate()*, similar to the Weighted Average method, the global position of the robot is computed based on odometry measurements. Additionally, in the last line, the covariance matrix is updated according to Equation 12.

```
1  void motionUpdate(RobotState Δ)
2  {
3      //Update the pose of the robot in the body frame of the robot originated at the ...
           inital Pose of the robot measured by odometry with sensor noise
4      pose.x += Δ.x + robotParams.odom_noise_translation_from_translation;
5      pose.y += Δ.y + robotParams.odom_noise_translation_from_translation;
6      pose.theta += Δ.theta + robotParams.odom_noise_rotation_from_rotation;
7
8      // Limit the angle to the specified range
9      angLimit(pose.theta);
10
11     //Update the global Pose of the robot with respect to the origin measured by ...
           odometry
12     poseGlob.x = pose.x*cos(pose.theta);
13     poseGlob.y = pose.y*sin(pose.theta);
14
15     //Store the global Pose of the robot with respect to the origin measured by ...
           odometry for plots
16     poseGlobFig.x = poseGlob.x;
17     poseGlobFig.y = poseGlob.y;
18
19     // Update Kalman Filter prediction step
20     P = A * P * A.transpose() + Q;
21    }
```

In the subsequent phase, akin to the aforementioned Weighted Average method, the positions of the robot measured by the markers are acquired in the global frame. Subsequently, the Kalman gain is computed in accordance with Equation 13 and applied in Equation 14 to update the odometry measurements. In the final line, the covariance matrix P is updated based on the obtained Kalman gain and the previous covariance matrix in the prediction step.

It is worth noting that since we have a maximum of four observations from the markers at each sampling time, the prediction step must be iterated for every marker observation. This necessitates computing the prediction step in a loop for all available observations at each sampling time. In other words, in the case of having four independent measurements from markers, the prediction and update steps are as follows:
prediction step based on odometry in the first sampling time
update step based on marker 1 in the first sampling time
update step based on marker 2 in the first sampling time
update step based on marker 3 in the first sampling time
update step based on marker 4 in the first sampling time
prediction step based on odometry in the second sampling time
update step based on marker 1 in the second sampling time
update step based on marker 2 in the second sampling time
update step based on marker 3 in the second sampling time
update step based on marker 4 in the second sampling time
...

```
1  void sensorUpdate(std::vector<MarkerObservation> observations)
2  {
3      // Loop through each observed marker and update the Kalman Filter
```

```
    for (const auto& observation : observations) {

        //Add noise to the observations obtained by markes
        markerOutput.distance  = observation.distance + ...
            robotParams.sensor_noise_distance;
        markerOutput.orientation  = observation.orientation + ...
            robotParams.sensor_noise_orientation;

        //Update the pose of the robot in the marker frame obtained by marker ...
            observations
        markerPose.x = markerOutput.distance * ...
            cos(markerOutput.orientation)*cos(pose.theta)- markerOutput.distance * ...
            sin(markerOutput.orientation)*sin(pose.theta);
        markerPose.y = markerOutput.distance * cos(markerOutput.orientation )* ...
            sin(pose.theta) + markerOutput.distance * sin(markerOutput.orientation ...
            )* cos(pose.theta);

        //Update the pose of the robot in origin frame obtained by marker observations
        markerPoseGlob.x = landmarkLocations[observation.markerIndex].x - markerPose.x;
        markerPoseGlob.y = landmarkLocations[observation.markerIndex].y - markerPose.y;

        //Store the pose of the robot in the global frame obtained by marker ...
            observation for plots
        markerPoseGlobFig[observation.markerIndex].x = markerPoseGlob.x;
        markerPoseGlobFig[observation.markerIndex].y = markerPoseGlob.y;

        // Update the measurement vector of the markers
        z_marker << markerPoseGlob.x, markerPoseGlob.y;

        // Update the measurement vector of the markers odometry
        z_odo << poseGlob.x, poseGlob.y;

        // Compute Kalman gain
        Eigen::Matrix<double, 3, 2> K = P * H.transpose() * (H * P * H.transpose() + ...
            R).inverse();

        // Compute innovation
        Eigen::Vector3d innovation = K * (z_marker - z_odo);

        //Update the global robot position of odometry by Kalman Filter
        poseGlob.x += innovation(0);
        poseGlob.y += innovation(1);

       // Update Kalman Filter covariance matrix
       P = (Eigen::Matrix3d::Identity() - K * H) * P;
    }
}
```

In the final phase, *getRobotPositionEstimate* receives the Kalman Filter estimations as input, as follows:

```
void getRobotPositionEstimate(RobotState& estimatePosn)
{
  // Return the current estimated robot position
  estimatePosn.x = poseGlob.x;
  estimatePosn.y = poseGlob.y;
  estimatePosn.theta = pose.theta;
}
```

The outputs of the marker observations, odometry, and the Kalman Filter are compared in Figure 3. It is evident that the odometry exhibits a drift, attributed to the presence of noise. For the marker observations, the domains of jumps become large due to the noise. In contrast, the Kalman Filter estimation shows no drift and demonstrates less noisy behavior, even in instances where some observations are lost during the simulation.

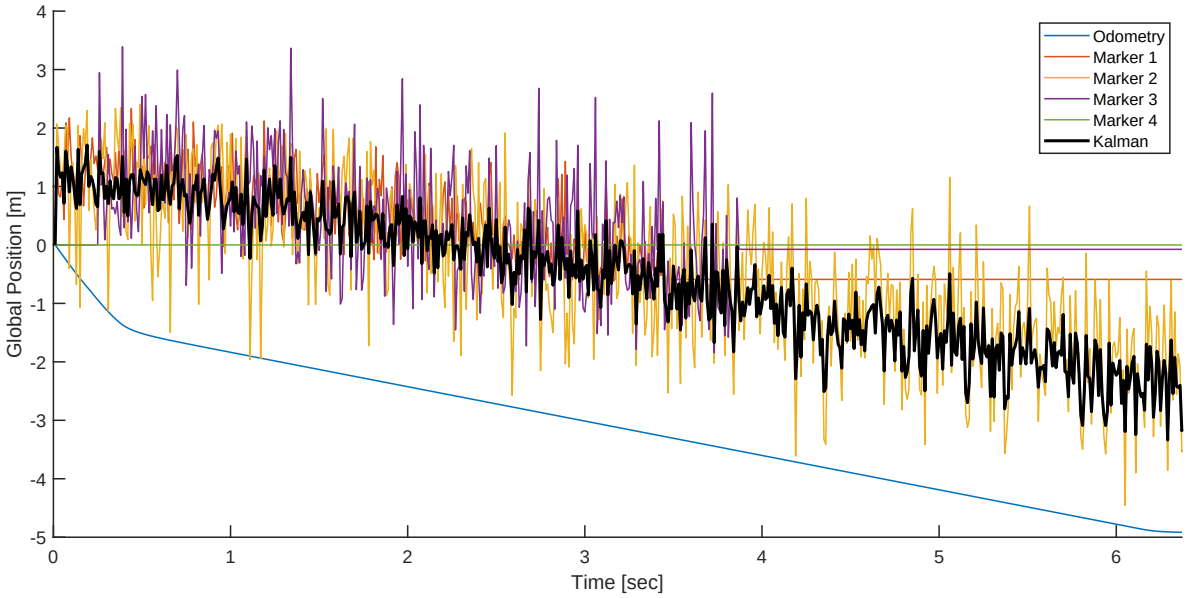

Figure 3: Output of the Markers observations, the odometry, and the Kalman Filter.