Hands-on Localization

Lab #1 - Feature EFK SLAM

Delivered by:

Rihab Laroussi, u1100330

Davina Sanghera, u1100402

Supervisor:

Ridao Rodriguez, Pedro

Date of Submission:

17/03/2025

Table of Contents

1 Introduction.....3
2 Part I: Localization without feature measurements..... 4
3 Part II: SLAM with a priori-known features..... 6
4 Part III: Full SLAM.....7
5 Issues..... 12
6 Conclusion..... 12

1 Introduction

In this lab, we extend a map-based EKF localisation system into a feature-based EKF SLAM system. We achieve this by incorporating feature estimates into the robot's state, which expands as new, unmapped features are detected. Several methods and variables from the previous lab have to be modified to support this functionality, with key changes being made to the prediction method, the observation model and its associated Jacobians. Once the extended FEFKSLAM class is implemented, we incrementally evaluate the system's performance by introducing updates and adding new features throughout this three-part lab.

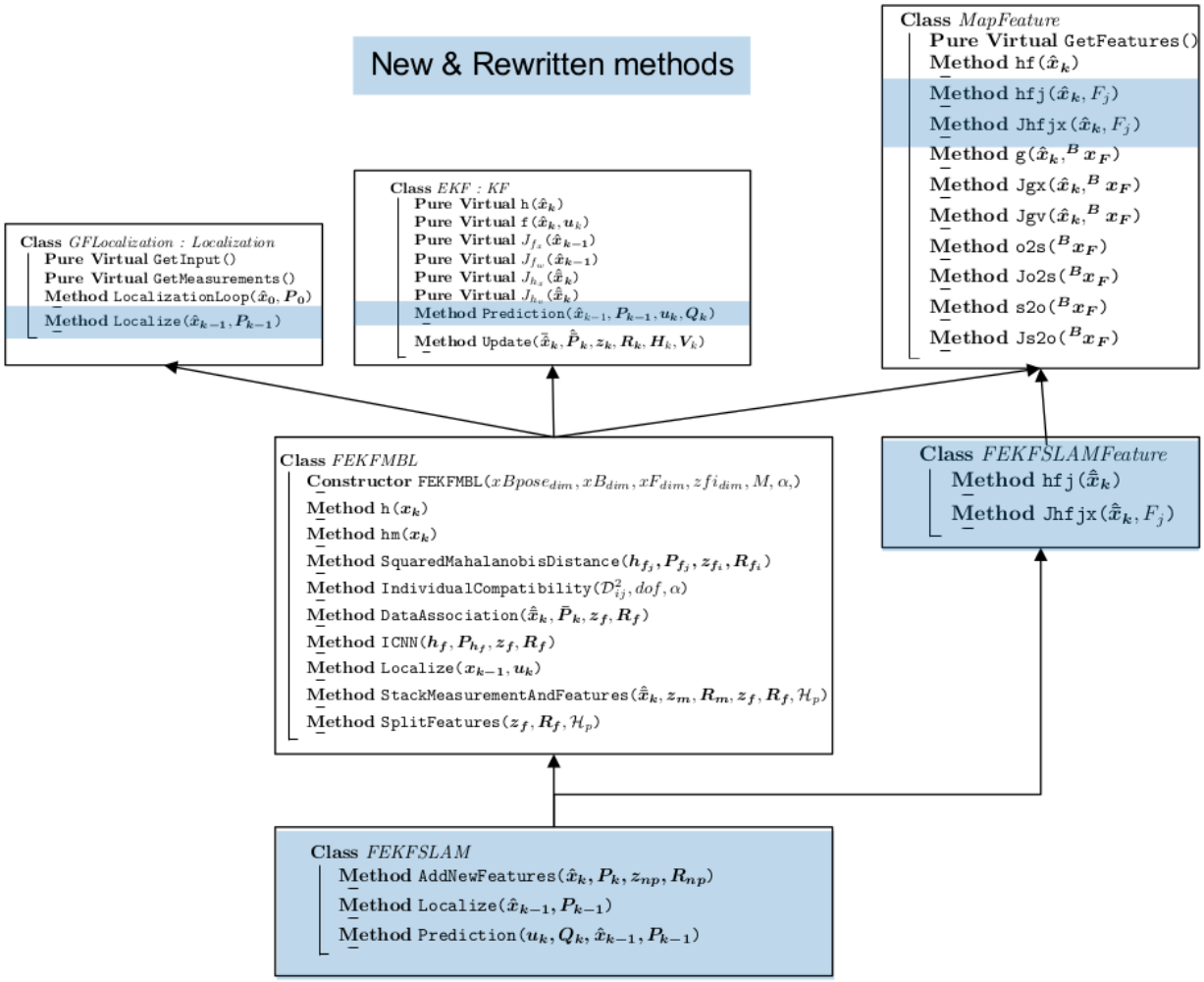

Fig1. SLAM Class Hierarchy

2 Part I: Localization without feature measurements

To move from a map-based EKF localisation system to a feature-based EKF SLAM system, the first step was to update the prediction function so that it considers feature mapping. Previously, the state vector and its associated covariance matrix only included the robot pose, therefore we had to expand these variables in order to include the positions of features in the map.

The prediction step updates the estimated state and its associated covariance using the following formulae:

$$
\begin{aligned}
{}^{N}\hat{\bar{x}}_{B_k} &= f\left({}^{N}\hat{x}_{B_{k-1}}, u_k, 0\right) \\
{}^{N}\bar{P}_{B_k} &= J_{f_x} P_{B_{k-1}} J_{f_x}^T + J_{f_w} Q J_{f_w}^T
\end{aligned}
$$

The motion model **f()** operates the same as before, taking the previous state and the control input to produce the state prediction. However, including features within the state vector, introduces dependencies between the robot's pose and the feature positions. These dependencies are captured within the covariance matrix, where the first term ($J_{fx}P_{k-1}J_{fx}^T$) defines how the uncertainty of the previous state estimate propagates through the motion model and the second term ($J_{fw}QJ_{fw}^T$) adds on the contribution of the process noise to the uncertainty. The structure of the updated covariance matrix is shown below:

$$
\left[\begin{array}{c|c}
J_{f_x} P_{B_k} J_{f_x}^T + J_{f_w} Q J_{f_w}^T & J_{f_x} P_{B_k F_1 \ldots F_n} \\
\hline
{}^{N}P_{B_k F_1 \ldots F_n}^T J_{f_x}^T & {}^{N}P_{F_1 \ldots F_n}
\end{array}\right]
$$

Since the actual feature positions are fixed but their estimated positions depend on previous estimates of the robot's pose, any uncertainty in the robot's motion propagates to the feature position estimates. This dependency between the uncertainty of the robot pose and the uncertainty of the feature positions is encoded in the cross-covariance terms of the matrix.

In the above matrix:

- The top left block represents the predicted uncertainty of the robot's pose.
- The top right block represents the correlation between the uncertainty of the robot's pose and the feature estimates.
- The bottom left block is the transpose of the top right block and therefore represents the same correlation.
- The bottom right block represents the uncertainty of the feature positions, which doesn't change during the prediction step.

During the update step of the EKF algorithm, these uncertainties are jointly corrected since new observations are used to refine both the robot's pose and the feature positions.

The implementation of the prediction step is given below:

```
def Prediction(self, uk, Qk, xk_1, Pk_1):

    # predict the state vector mean and covariance at time step k
    # we have xBk_1, xFi and Pk_1 , we calculate xk_bar and Pk_bar while xFi remained unchanged

    self.xk_1 = xk_1
    self.Pk_1 = Pk_1
    self.uk = uk
    self.Qk = Qk

    # xk_bar = [f(xk_1, uk, 0) xF1 ..... xFn]T
    xk_bar = np.array(xk_1)
    xk_bar[0:3] = self.f(xk_1[0:3], uk)

    xBk_1 = Pose3D(xk_1[0:3])

    # Covariance Prediction: Pk_bar = F1*Pk_1*F1.T + F2*Qk*F2.T
    # Covariance Matrix: Pk_bar = np.block([ Pxx, Pxf], [Pfx, Pff])
    # Pxx is the covariance of the robot pose
    J_fx = self.Jfx(xBk_1)
    J_fw = self.Jfw(xBk_1)

    NPk = Pk_1[0:3,0:3]
    Pxx = J_fx @ NPk @ J_fx.T + J_fw @ Qk @ J_fw.T
    # Pxf is the correlation between the robot and the features
    Pxf = J_fx @ Pk_1[0:3,3:]
    # Pfx is the correlation between the features and the robot
    Pfx = Pxf.T
    # Pff is the covariance of the features
    Pff = Pk_1[3:,3:]
    Pk_bar = np.block([[Pxx,Pxf], [Pfx,Pff]])

    return xk_bar, Pk_bar
```

The **FEKFSLAM_3DOFDD_InputVelocityMM_2DCartesianFeatureOM.py** file tests these modifications using a 3DOF differential drive robot and 2D cartesian features. Since only the prediction method is implemented, the state and feature updates are excluded from the **Localize()** method which performs a single iteration of the SLAM algorithm.

3 Part II: SLAM with a priori-known features

With the prediction step updated to accommodate for feature mapping, we next modify the update step. In **FEKFSLAMFeature.py**, both the observation model (**hfj**) and its Jacobian (**Jhfjx**) are overridden so that the features are obtained from the state vector as opposed to the predefined map.

The implementation of **hfj()** is given below:

```
def hfj(self, xk_bar, Fj):  # Observation function for zf_i and x_Fj

    NxB = xk_bar[0:3,0].reshape((3,1))
    start_index = 3 + Fj * self.xF_dim
    NxF = CartesianFeature(xk_bar[start_index:start_index+self.xF_dim].reshape((self.xF_dim, 1)))
    hfj = self.s2o(NxF.boxplus(Pose3D.ominus(NxB)))
    return hfj
```

This method extracts both the robot pose (**NxB**) and the stored position of the feature **Fj** from the state vector. It then obtains the expected measurement (**zf**) of the feature by applying the boxplus operation to the feature position and the ominus operation to the robot pose.

The implementation of **Jhfjx()** is given below:

```
def Jhfjx(self, xk, Fj):  # Observation function for zf_i and x_Fj

    # Jacobian of the observation model for a single feature with respect to the state vector
    # It represents how small changes in the state vector x k  affect the predicted observation of a specific feature F j
    NxB = xk[0:3,0].reshape((3,1))
    start_index = 3 + Fj * self.xF_dim
    NxF = CartesianFeature(xk[start_index:start_index+self.xF_dim].reshape((self.xF_dim, 1)))
    J1 = self.J_s2o(NxF.boxplus(Pose3D.ominus(NxB)))
    J2 = NxF.J_1boxplus(Pose3D.ominus(NxB))
    J3 = Pose3D.J_ominus(NxB)
    J4 = NxF.J_2boxplus(Pose3D.ominus(NxB))

    J = np.zeros((2,xk.shape[0]))   # 2 rows, len(xk) = xk.shape[0] number of columns

    J[:2, :3] = J1 @ J2 @ J3  # first two rows, first three colums
    J[:2, start_index:start_index + self.xF_dim] = J1 @ J4   # first two rows, feature columnc
    return J
```

This method computes the Jacobian of the observation model (**hfj**) with respect to the state vector (**xk**). This represents how changes in the state vector (both the robot's pose and the feature positions) affect the expected observation. The first three columns of the matrix capture the effect of the robot's pose on the expected observation, using the Jacobian of s2o, the Jacobian of J1boxplus and the Jacobian of ominus. The feature position columns of the matrix capture the effect of the feature's position on the expected observation, using the Jacobian of s2o and the Jacobian of J2boxplus.

This Jacobian is then used within the update step of the EKF, where it is used to compute the Kalman gain and to refine the state estimate.

The **FEKFSLAM_3DOFDD_InputVelocityMM_2DCartesianFeatureOM.py** file is executed again, this time with the update step included within the **Localize()** method. Additionally, the initial uncertainty of the features in the state vector is increased to demonstrate the effectiveness of the update step in refining the state estimate.

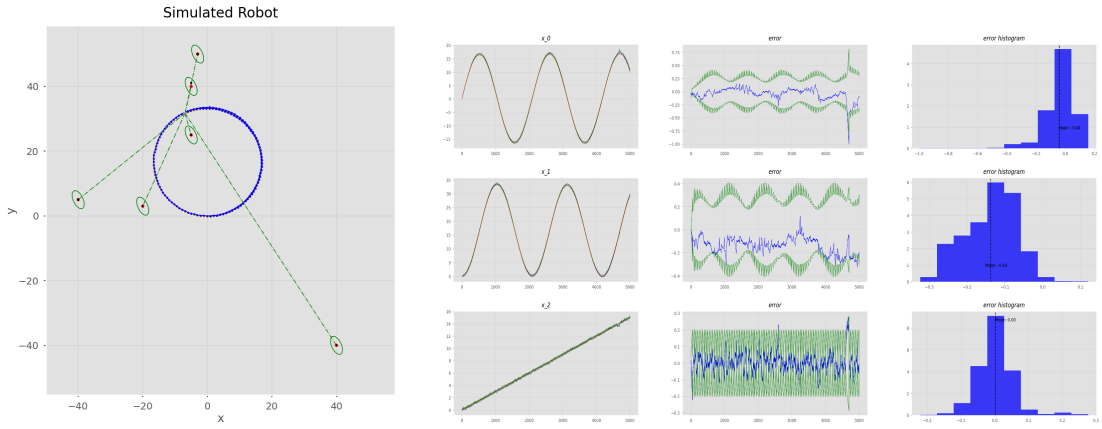

Fig2. Simulation result including performing updates

Compared to Part I, where only the prediction step was used, including the update step, allows the system to continually refine its estimate based on the observations of the map features. Therefore, as expected, the error is significantly reduced and the estimated state aligns much closer with the ground truth.

4 Part III: Full SLAM

Finally, we implement the full SLAM algorithm by augmenting the state vector when new, unmapped features are detected. We set the initial state vector to contain only the robot state **[x, y, yaw]**.

First, we implemented the inverse observation equation **g()** and its jacobians **Jgx** & **Jgv** in the **MapFeature** class.

1. **g()**: Inverse Observation Modelclass.

The inverse observation model computes the position of a new feature in the global frame

(N-Frame) based on the robot's pose **(NxB)** and observation **(BxFj)**.

```
def g(self, xk, BxFj):

     # Extract robot pose (NxB) from the state vector
      NxB = Pose3D(xk[0:3, 0].reshape((3, 1)))
      # Convert the feature observation to the global frame using boxplus
      NxFj =(self.o2s(BxFj)).boxplus(NxB)
      return NxFj
```

2. **Jgx():** Jacobian with respect to the state vector

This method computes the Jacobian of the inverse observation model with respect to the state vector **xk.**

```
 def Jgx(self, xk, BxFj):

        # Extract robot pose (NxB) from the state vector
        NxB = Pose3D(xk[0:3,0].reshape((3,1)))
        # Compute Jacobian of boxplus with respect to the robot's pose
        Jp = (self.o2s(self.Feature(BxFj))).J_1boxplus(NxB)     #Jacobian matrix of the feature
with respect to the robot's pose.
        xF_dim = np.shape(BxFj)[0]     # Number of rows in the feature observation
        Jnp =  np.zeros((xF_dim, self.xB_dim - self.xBpose_dim))  # zero matrix , Jnp = 0 : xF_dim
rows and (self.xB_dim - self.xBpose_dim) columns
        J = np.block([Jp,Jnp])  # Full Jacobian
        return J
```

3. **Jgv():** Jacobian with respect to the observation noise

This method computes the Jacobian of the inverse observation model with respect to the observation noise **vk**.

```
  def Jgv(self, xk, BxFj):

        # Extract robot pose (NxB) from the state vector
        NxB = Pose3D(xk[0:3,0].reshape((3,1)))
        # Compute Jacobian of boxplus with respect to the observation
        J = (self.o2s(self.Feature(BxFj))).J_2boxplus( NxB) @ self.J_o2s(BxFj)
        return J
```

Next, we implemented the **AddNewFeatures** function, to add newly detected features to the state vector and covariance matrix.

The function iterates over the non-paired feature observations in **znp**. For each new feature observation, it computes the feature's position in the global frame (N-Frame) using the **g()** method, which implements the inverse observation model. This position is then appended to the grown state vector **xk_plus**, effectively expanding the state vector to include the new feature.

$$\hat{\boldsymbol{x}}_{\boldsymbol{k}}^{+} = \left[\hat{\boldsymbol{x}}_{\boldsymbol{k}}^{\boldsymbol{T}} \; \boldsymbol{g}\left(\hat{\boldsymbol{x}}_{\boldsymbol{B_k}}, \boldsymbol{z_{np}}\right)^{\boldsymbol{T}}\right]^{T};$$

Next, the function computes the grown covariance matrix **Pk_plus**. This involves calculating two things: the cross-covariance between the robot's pose and the new feature, and the covariance of the new feature itself. The cross-covariance captures the relationship between the robot's uncertainty and the new feature's position, while the new feature's covariance accounts for the uncertainty introduced by the observation noise as shown in the equation below.

$$\boldsymbol{P}_{\boldsymbol{k}}^{+} = \left[\begin{array}{c|c} \boldsymbol{P}_{\boldsymbol{k}} & \boldsymbol{P}_{\boldsymbol{B}_k\boldsymbol{F}_1\ldots\boldsymbol{F}_n}^{\boldsymbol{T}}\boldsymbol{J}_{\boldsymbol{g}_x}^{\boldsymbol{T}} \\ \hline \boldsymbol{J}_{\boldsymbol{g}_x}\boldsymbol{P}_{\boldsymbol{B}_k\boldsymbol{F}_1\ldots\boldsymbol{F}_n} & \boldsymbol{J}_{\boldsymbol{g}_x}\boldsymbol{P}_{\boldsymbol{R}}\boldsymbol{J}_{\boldsymbol{g}_x}^{\boldsymbol{T}} + \boldsymbol{J}_{\boldsymbol{g}_v}\boldsymbol{R}_{\boldsymbol{f}}\boldsymbol{J}_{\boldsymbol{g}_v}^{\boldsymbol{T}} \end{array}\right];$$

After processing all new feature observations, the function returns the updated state vector **xk_plus** and covariance matrix **Pk_plus**, which now include the newly mapped features and their associated uncertainties.

$$[\hat{\boldsymbol{x}}_{\boldsymbol{k}}, \boldsymbol{P}_{\boldsymbol{k}}] = [\hat{\boldsymbol{x}}_{\boldsymbol{k}}^{+}, \boldsymbol{P}_{\boldsymbol{k}}^{+}]$$

The code:

```
def AddNewFeatures(self, xk, Pk, znp, Rnp):

    xk_plus = xk            # Start with the current state vector
    Pk_plus = np.array(Pk)   # Start with the current covariance matrix

    zfi_dim = self.xF_dim  # dimension of the new feature
    xB_dim = self.xB_dim  # dimension of the robot pose

    xBk = Pose3D(xk[0:xB_dim]) # robot pose in the N frame
    PBk = Pk[0:xB_dim, 0:xB_dim] # covariance of the robot pose

   # Iterate over each new feature observation in znp
    for i in range(len(znp)):

        # calculate the new feature position in the N frame
        xfi = self.g(xBk, znp[i])

        # append the new feature to the state vector mean
        xk_plus = np.vstack([xk_plus, xfi])

        # Pk_plus = G1 * Pk_1 * G1.T + G2 * Rk * G2.T
        # Pk_plus = [Pk Pcross.T ; Pcross Pfi]

        # calculate the Jacobians
        Jgxi = self.Jgx(xBk, znp[i])
        Jgvi = self.Jgv(xBk, znp[i])

        # compute the covariance of the new feature
        Pfi = Jgxi @ PBk @ Jgxi.T + Jgvi @ Rnp[i] @ Jgvi.T

        # compute the cross-covariance between the robot's pose and the new feature
        Pk_B = Pk[0:xB_dim, 0:xB_dim]       # covariance of robot's pose with itself
        Pk_BF = Pk[0:xB_dim, xB_dim:]       # cross-covariance with existing features
        Pcross =  np.block([Jgxi @ Pk_B,Jgxi @ Pk_BF])

        # update the state vector covariance
        Pk_plus = np.block([[ Pk_plus, Pcross.T], [Pcross, Pfi]])
        Pk = Pk_plus
        self.nf +=1 # increment the number of features

    return xk_plus, Pk_plus
```

We then upgraded the **Localize** method to add new features without performing updates. The function performs the prediction step and dynamically adds new features to the state vector but does not update the robot's pose or feature positions using measurements. Feature measurements

are first retrieved using the **GetFeatures** function, which is then processed by the **DataAsdociation** function, which determines whether they correspond to known features or represent new

unmapped ones, which later the **StackMeasuremntsAndFeatures** organizes them. The detected features are added to the state vector via the inverse observation model but remain static since no corrections are applied.

```
    def Localize(self, xk_1, Pk_1):

        uk, Qk = self.GetInput()
        xk_bar, Pk_bar  = self.Prediction(uk, Qk, xk_1, Pk_1)
        zm, Rm, Hm, Vm  = self.GetMeasurements()
        zf, Rf  = self.GetFeatures()
        Hp = self.DataAssociation(xk_bar, Pk_bar, zf, Rf)
        [zk, Rk, Hk, Vk, znp, Rnp] = self.StackMeasurementsAndFeatures(xk_bar, zm, Rm, Hm, Vm, zf,
Rf, Hp)
        #If there are enough new, unmatched features, add them to the state representation
        if(len(znp) >= self.xF_dim):
             xk, Pk = self.AddNewFeatures(xk_bar, Pk_bar,znp,Rnp)
        else:
             xk, Pk = xk_bar, Pk_bar

        self.xk, self.Pk = xk, Pk
        # Use the variable names zm, zf, Rf, znp, Rnp so that the plotting functions work
        self.Log(self.robot.xsk, self.GetRobotPose(self.xk), self.GetRobotPoseCovariance(self.Pk),
                 self.GetRobotPose(self.xk), zm)  # log the results for plotting

        return self.xk, self.Pk, zf, Rf, znp, Rnp
```

We also added a new method called **LocalizationLoop().** It is similar to the Map-based localizationLoop function, except this one has the AddNewFeatures function included. It integrates the prediction, update, and feature augmentation steps to localize the robot and build a map of the environment. By iteratively updating the state vector and covariance matrix, the method ensures accurate localization and mapping, even as new features are detected.

```
 def LocalizationLoop(self, x0, P0, usk):

     xk_1 = x0
     Pk_1 = P0

     xsk_1 = self.robot.xsk_1

     zf, Rf = self.GetFeatures()

     # add new features to the state vector and covariance matrix
     xk_1, Pk_1 = self.AddNewFeatures(xk_1, Pk_1, zf, Rf)

     for self.k in range(self.kSteps):

         xsk = self.robot.fs(xsk_1, usk)  # simulate the robot motion
         xk, Pk, zf, Rf, znp, Rnp = self.Localize(xk_1, Pk_1)  # localize the robot
         zf = np.array(zf).reshape(len(zf) * self.xF_dim ,1) # zf becomes fJ rows and 1 column
         Rf = block_diag(*Rf) # take Rf and pass its elements as arg to block_diag to craete a
block diagonal matrix
         self.PlotUncertainty(zf, Rf, znp, Rnp)

         xsk_1 = xsk  # current state becomes previous state for next iteration
         xk_1 = xk
         Pk_1 = Pk

     self.PlotState()  # plot the state estimation results
     plt.show()
```

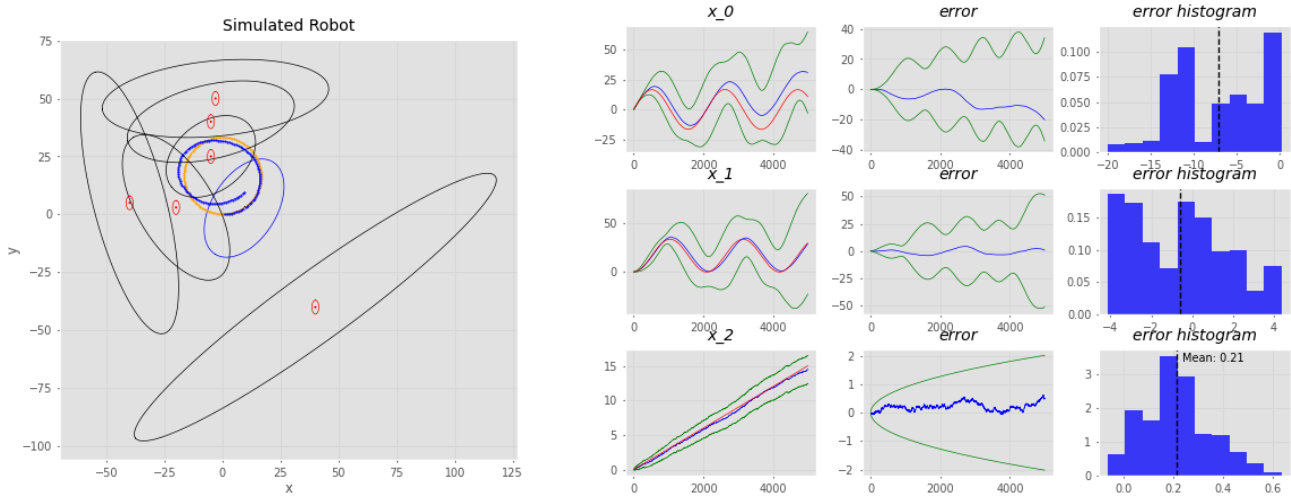

Fig3. Simulation result without performing updates

The SLAM algorithm, when implemented without updates, lacks the ability to update its estimates. As seen from the output, the estimated robot trajectory deviates from the ground truth over time with the error continuously increasing, causing a significant drift. Also, the uncertainty ellipses for the robot's pose and features keep growing.

Then we upgraded the **Localize** method to perform updates. We added the function **update()** with the following line.

```
xk, Pk = self.Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
```

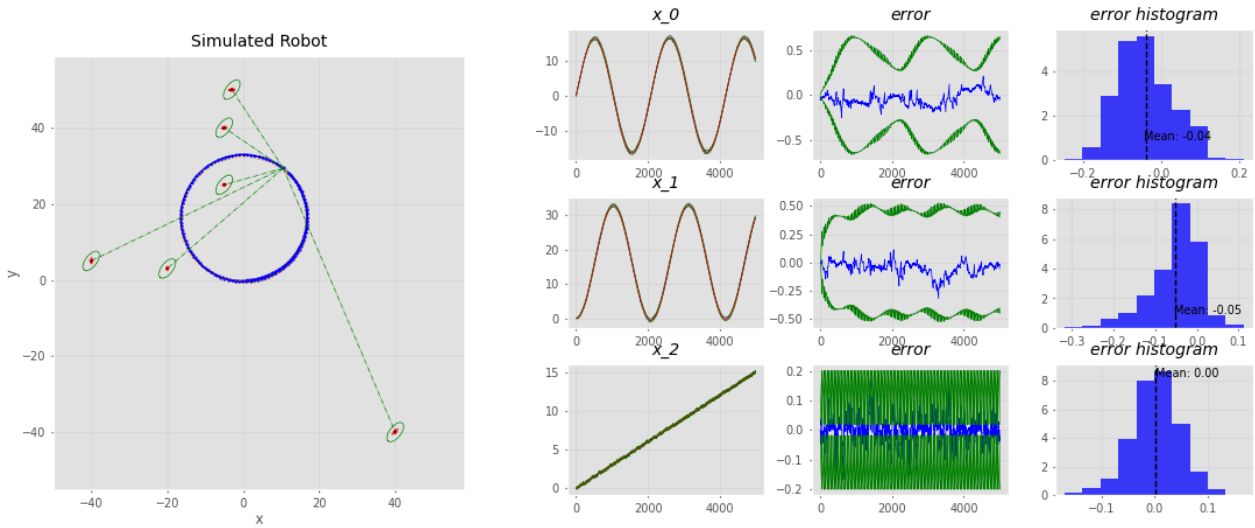

Fig4. Full SLAM simulation result

After implementing the full SLAM algorithm, we observe that the robot's trajectory is closely aligned with the ground truth, and the feature positions are updated iteratively. The output has a shrinked and stabilized uncertainty which reflects the improved estimations.

5 Issues

The main issue is that we have is an AttributeError when attempting to use the boxplus operation on the output of the o2s function. As for now, this issue remains, and it causes the simulation to terminate unexpectedly sometimes.

6 Conclusion

In conclusion, this lab provided a comprehensive introduction to the implementation of full SLAM algorithm and its challenges. By breaking the lab into 3 parts, we were able to gain a more thorough understanding of SLAM key components and their interplay in achieving an accurate localization and mapping.