NOTE- I started working on this problem statement on Saturday, as I was busy with some work previously(normal working days)
Date of Submission - 30th Oct 2023

Task-2

NOTE: <u>Read Task-1 documentation first, as this is made as a follow-up for the previous code. Things added to the previous code are documented and displayed over here, this is done to keep the document short and to the point.</u>

Aim -

- To Implement a sensor fusion pipeline to correct the wheel odometry drifts whenever the robot measures a landmark.
- Plot/record odometry-only localizations along with the fused-odometry localizations to compare the effectiveness of landmarks.
- The actual path that the robot traveled, odometry-only trajectory, and fused-odometry trajectory have to be plotted in overlap as a result.

Approach/Assumption -

- At first, as given in the problem statement the bot must measure relative landmarks within the 0.3m range and must implement a sensor fusion algorithm to correct the odometry drifts.
- However, if we see, the motion of the bot wasn't specified as such so I have assumed the motion to be based on the mathematical model of a differential drive robot, which assumes no error in the odometry.
- So, as the motion of the bot will be the same to that in task-1, so in between the point the bot will detect landmarks within 0.3 m of range.
- Moving on to the flow of the code, At first We can observe that there's a new function called error_odom, that is being called inside the move to goal, function.
- As previously mentioned the robot will follow the trajectory which has no error in odom values, for the sake of simpler understanding.
- Hence, we are calculating error Odom alongside Odom to store and compare the values and also pass it into the sensor fusion pipeline.
- Hence, As soon as the robot detects a landmark, it calculates it's position w.r.t the landmarks along with the error in it, and then finds the transform between the global frame and the robot's odom frame using the transformation between robot and landmark and landmark and global frame.

- It then passes these values to the Sensor fusion algorithm, here in this code we have used the Kalman filter to filter out the odometry drifts of the system.
- Further, we have plotted all the values given according to the problem statement.
- **Note:** At the time of writing due to submission deadlines, the code has some errors which need to be looked at, the code is almost done, a bit more time will solve the error. Due to the deadline for the submission, I am submitting this.

```
# Note - This code has been integrated and works properly, some
parts of it are yet to be connected or fixed otherwise
everything must work just fine you can also print the filtered
Odom data by uncommeting lines- 297,298. The codes work fine
it's just that the matplot lib needs to be integrated properly
but due to the submission deadline, I am submitting it on time,
up to how much I was able to complete in the given time frame.
Kalman filter works correctly in this, Yay!!
# close the program by closing the graph it will automatically
print the filtered Odom data and close :)
```

Code/Asummptions/Description -

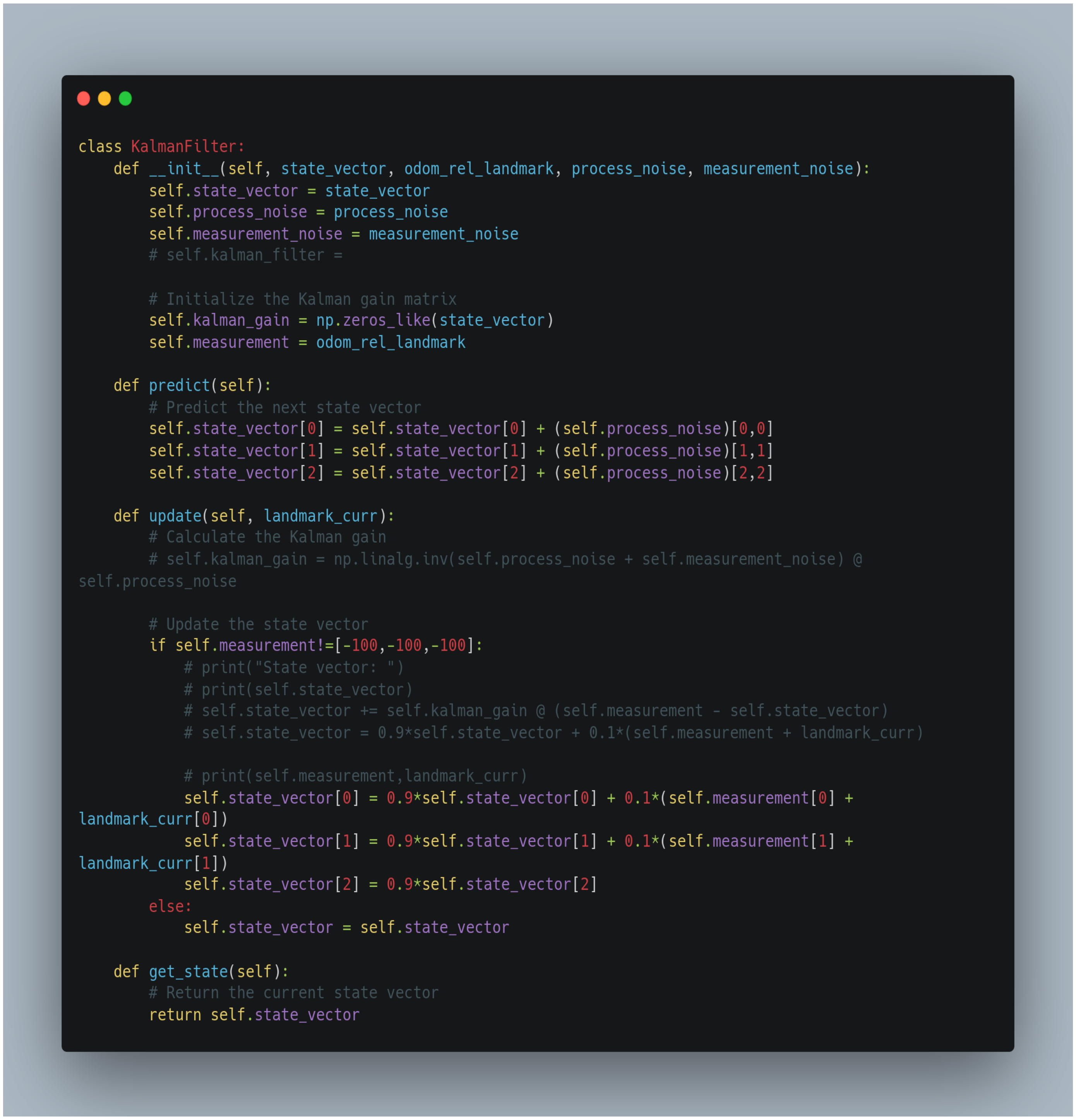
Kalman Filter Class-

code defines a KalmanFilter class that implements a simple Kalman filter for state estimation. Here's an overview of how this class works:

1. Initialization: In the constructor (__init__), the class is initialized with the following parameters:

   state_vector: The initial state vector representing the robot's state.
   odom_rel_landmark: The measurement representing the relative position of a landmark.
   process_noise: A matrix representing the process noise covariance.
   measurement_noise: A matrix representing the measurement noise covariance.
   The Kalman gain matrix (kalman_gain) is initialized with zeros.
2. Prediction step: The Kalman filter predicts the next state of the system using the system dynamics model.
3. Update step: The Kalman filter updates the predicted state of the system using the noisy measurements.

4. The prediction step is relatively straightforward. The Kalman filter simply uses the system dynamics model to predict how the state of the system will evolve over time.
5. The update step is more complex. The Kalman filter uses a weighted average of the predicted state and the noisy measurements to produce an updated estimate of the state of the system. The weights are determined by the Kalman gain, which is a matrix that is calculated based on the system dynamics model and the measurement noise.
6. The Kalman filter is a recursive algorithm, which means that it can be applied to a series of measurements to produce an estimate of the state of the system at each time step.



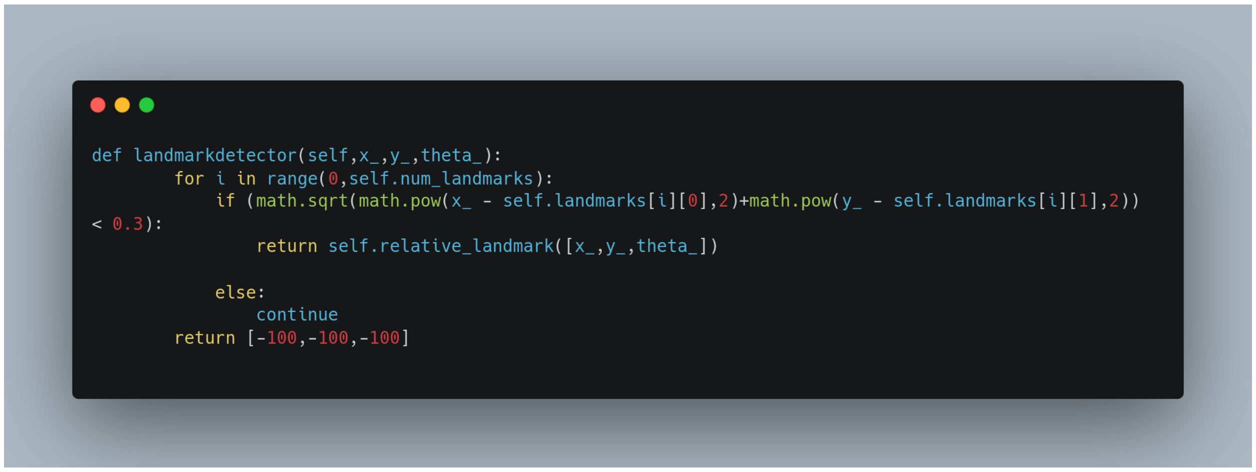
```python
class KalmanFilter:
    def __init__(self, state_vector, odom_rel_landmark, process_noise, measurement_noise):
        self.state_vector = state_vector
        self.process_noise = process_noise
        self.measurement_noise = measurement_noise
        # self.kalman_filter =

        # Initialize the Kalman gain matrix
        self.kalman_gain = np.zeros_like(state_vector)
        self.measurement = odom_rel_landmark

    def predict(self):
        # Predict the next state vector
        self.state_vector[0] = self.state_vector[0] + (self.process_noise)[0,0]
        self.state_vector[1] = self.state_vector[1] + (self.process_noise)[1,1]
        self.state_vector[2] = self.state_vector[2] + (self.process_noise)[2,2]

    def update(self, landmark_curr):
        # Calculate the Kalman gain
        # self.kalman_gain = np.linalg.inv(self.process_noise + self.measurement_noise) @
self.process_noise

        # Update the state vector
        if self.measurement!=[-100,-100,-100]:
            # print("State vector: ")
            # print(self.state_vector)
            # self.state_vector += self.kalman_gain @ (self.measurement - self.state_vector)
            # self.state_vector = 0.9*self.state_vector + 0.1*(self.measurement + landmark_curr)

            # print(self.measurement,landmark_curr)
            self.state_vector[0] = 0.9*self.state_vector[0] + 0.1*(self.measurement[0] +
landmark_curr[0])
            self.state_vector[1] = 0.9*self.state_vector[1] + 0.1*(self.measurement[1] +
landmark_curr[1])
            self.state_vector[2] = 0.9*self.state_vector[2]
        else:
            self.state_vector = self.state_vector

    def get_state(self):
        # Return the current state vector
        return self.state_vector
```

Landmarkdetector-

code defines a method called relative_landmark, which appears to calculate the relative position of a landmark with respect to the robot's instantaneous position and orientation. Here's a breakdown of how this method works:

1. It initializes an empty list rel_odom_instantaneous to store the relative position of the landmark.

2. It calculates the centroid (average) position of all landmarks by iterating through the list of landmarks and accumulating the x and y coordinates. This centroid calculation assumes that the landmarks are stored in the class.
3. It calculates the relative position of the landmark with respect to the robot's instantaneous position and orientation using the following formulas:
4. The relative x coordinate is the difference between the x centroid and the robot's x position, with added Gaussian noise.
5. The relative y coordinate is the difference between the y centroid and the robot's y position, with added Gaussian noise.
6. The relative orientation (theta) is calculated using math.atan2, taking into account both x and y differences. Gaussian noise is added to the result.
7. The calculated relative position is appended to a list rel_odom and also to self.robot_error_odom.
8. A Kalman filter is initialized with an initial state vector [robot.x_e, robot.y_e, robot.theta_e] (assuming these variables represent the robot's estimated position and orientation), process noise covariance, and measurement noise covariance. The prediction method of the Kalman filter is called.
9. The method returns the calculated relative position.



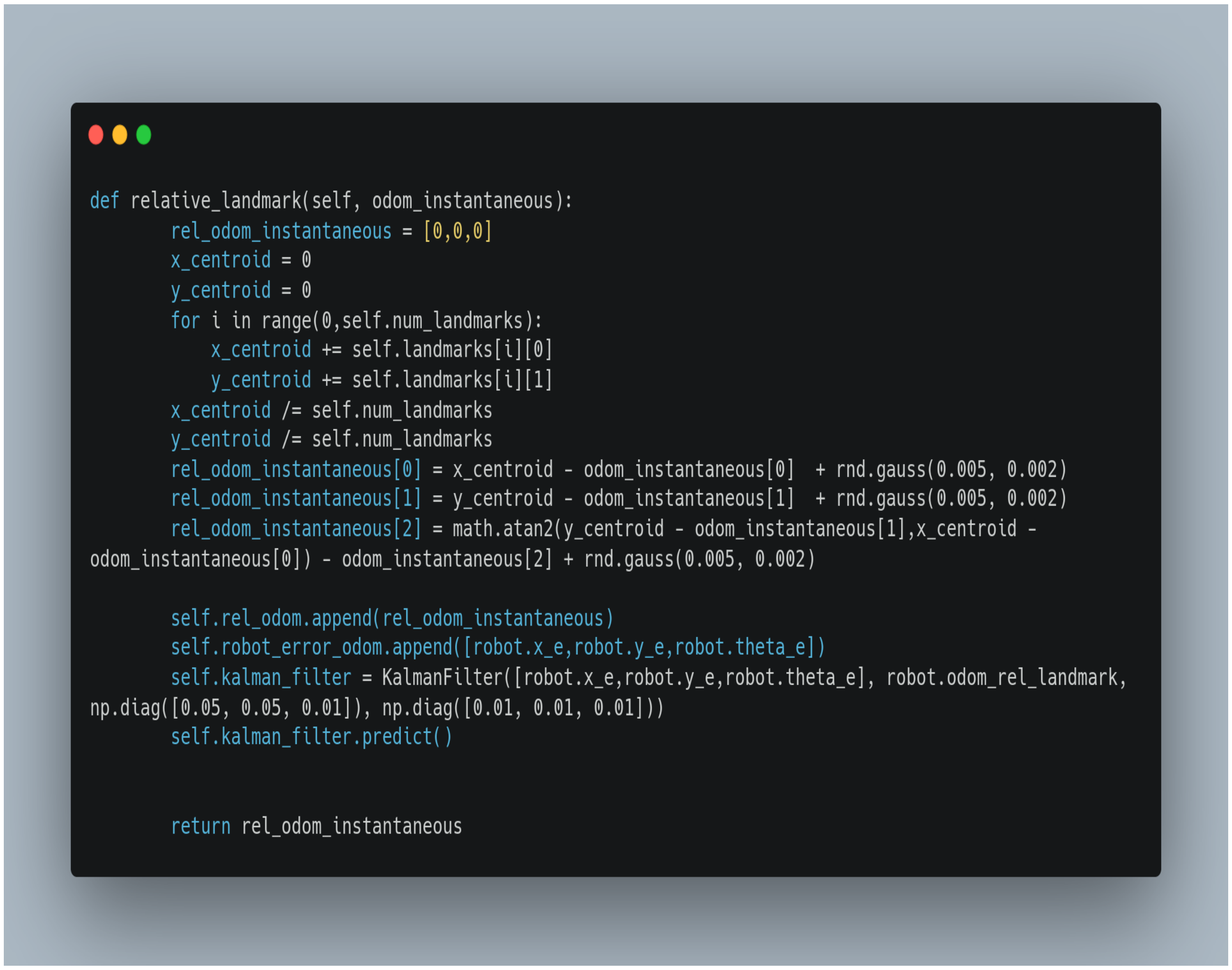
```
def landmarkdetector(self,x_,y_,theta_):
        for i in range(0,self.num_landmarks):
            if (math.sqrt(math.pow(x_ - self.landmarks[i][0],2)+math.pow(y_ - self.landmarks[i][1],2))
< 0.3):
                return self.relative_landmark([x_,y_,theta_])

            else:
                continue
        return [-100,-100,-100]
```

<u>Relative_landmark-</u>

code defines a method called relative_landmark, which appears to calculate the relative position of a landmark with respect to the robot's instantaneous position and orientation. Here's a breakdown of how this method works:

1. It initializes an empty list rel_odom_instantaneous to store the relative position of the landmark.
2. It calculates the centroid (average) position of all landmarks by iterating through the list of landmarks and accumulating the x and y coordinates. This centroid calculation assumes that the landmarks are stored in the class.
3. It calculates the relative position of the landmark with respect to the robot's instantaneous position and orientation using the following formulas:
   The relative x coordinate is the difference between the x centroid and the robot's x position, with added Gaussian noise.
   The relative y coordinate is the difference between the y centroid and the robot's y position, with added Gaussian noise.
   The relative orientation (theta) is calculated using math.atan2, taking into account both x and y differences. Gaussian noise is added to the result.
   The calculated relative position is appended to a list rel_odom and also to self.robot_error_odom.

4. A Kalman filter is initialized with an initial state vector [robot.x_e, robot.y_e, robot.theta_e] (assuming these variables represent the robot's estimated position and orientation), process noise covariance, and measurement noise covariance. The prediction method of the Kalman filter is called.
5. The method returns the calculated relative position.



```
def relative_landmark(self, odom_instantaneous):
        rel_odom_instantaneous = [0,0,0]
        x_centroid = 0
        y_centroid = 0
        for i in range(0,self.num_landmarks):
            x_centroid += self.landmarks[i][0]
            y_centroid += self.landmarks[i][1]
        x_centroid /= self.num_landmarks
        y_centroid /= self.num_landmarks
        rel_odom_instantaneous[0] = x_centroid - odom_instantaneous[0]  + rnd.gauss(0.005, 0.002)
        rel_odom_instantaneous[1] = y_centroid - odom_instantaneous[1]  + rnd.gauss(0.005, 0.002)
        rel_odom_instantaneous[2] = math.atan2(y_centroid - odom_instantaneous[1],x_centroid -
odom_instantaneous[0]) - odom_instantaneous[2] + rnd.gauss(0.005, 0.002)

        self.rel_odom.append(rel_odom_instantaneous)
        self.robot_error_odom.append([robot.x_e,robot.y_e,robot.theta_e])
        self.kalman_filter = KalmanFilter([robot.x_e,robot.y_e,robot.theta_e], robot.odom_rel_landmark,
np.diag([0.05, 0.05, 0.01]), np.diag([0.01, 0.01, 0.01]))
        self.kalman_filter.predict()


        return rel_odom_instantaneous
```

The Rest is the code for plotting the graph :)

Limitations and scope of future work-

- The data input for plotting the Graph needs to be corrected.
- The Data visualization part needs to be worked on a bit more time into will solve that issue.

This Document Henceforth provides the full documentation of the Task-2 code, assumptions made, available functionalities, limitations, and scope of future Work.

Thanks for reading you can find the simulation code in the code folder, and the simulation recording in the ss/recordings folder named according to their respective task name