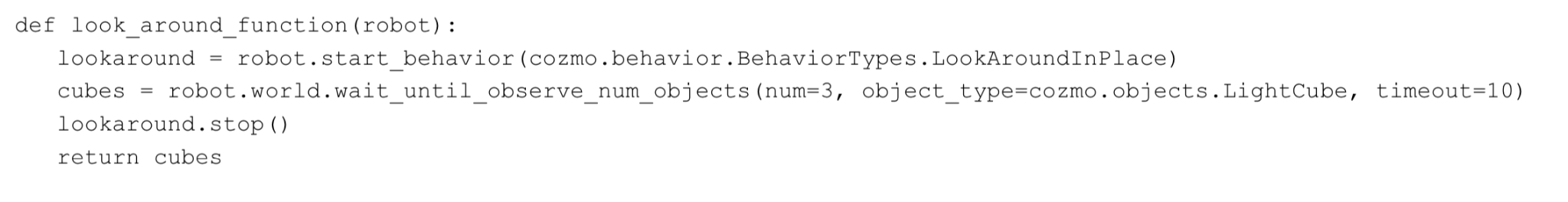
Search And Rescue Robot Report

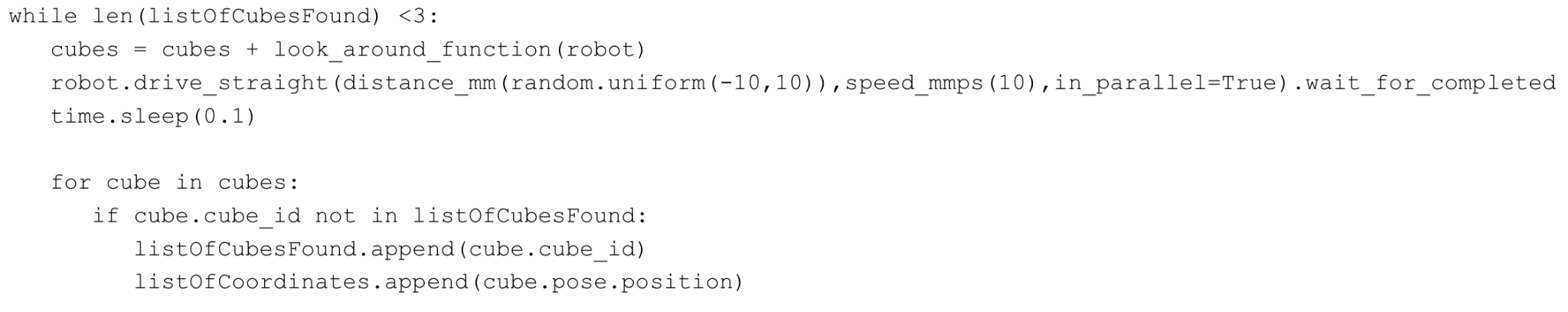
Exploration

The exploration process starts with a look around in place and observing all the objects and remembering the cubes only. It will try and look for 3 cubes for 10 seconds, when the function times-out it will return all the cubes if found if any.

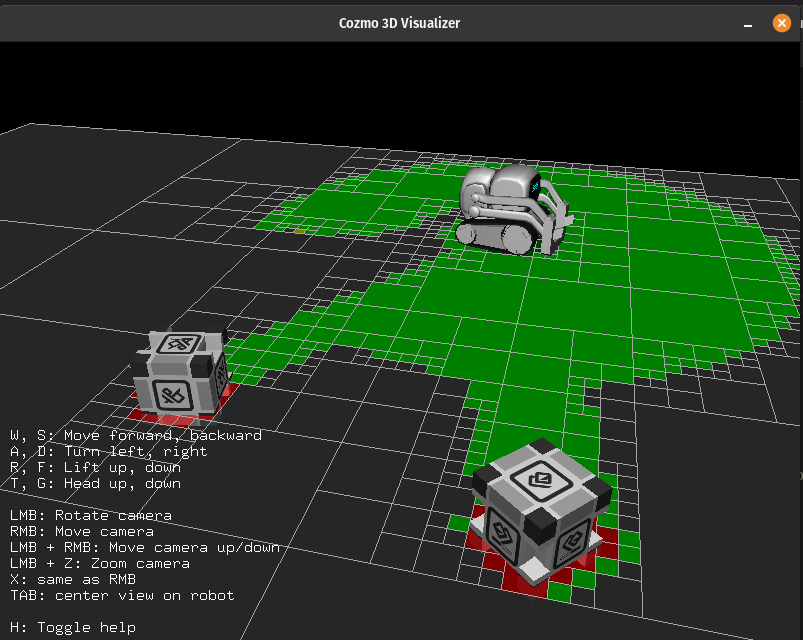
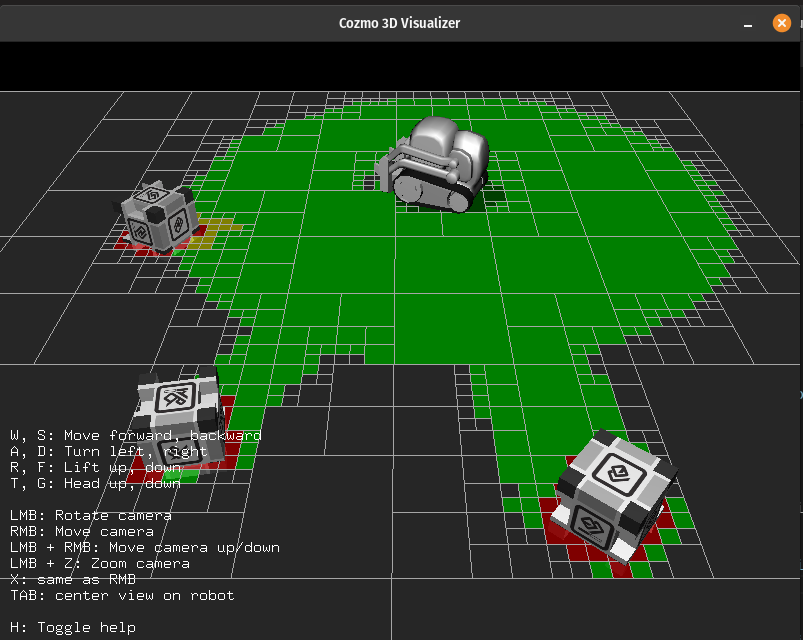


In order to move from the Exploration to the next step we need to make sure that all the 3 cubes have been found and we know the location of those cubes. To do that a while loop is implemented to check how many cubes have been found. If the robot can’t find all 3 cubes from its initial position it will move in a random direction which is achieved by 2 built in functions, the first one being the LookArroundInPlace which spins the robot for the 10 seconds mentioned above which will leave the robot facing a random angle after the timeout. The second function will either move the robot forward up to 10mm or back, combining the two random functions will result in a random behaviour from the robot. And in order to avoid collisions whilst moving the robot around, the distance is limited up to 10mm in each direction and at low speed.

In order to avoid adding the same cube in the list of found cubes, first we check if the cube doesn’t exist in the list by simply just looking if the Cube’s ID is already in the list. There are two lists, one list just checks track of the Cube’s ID and the other list stores the data about the Coordinates of each cube.



Due to the fact that the movements are random and there is no data from any kind of external sensors, sometimes it detects all three cubes very quickly and sometimes it takes quite a long time to detect all the targets. As an example the pictures below show the position of the cubes in relation to the robot according to cosmo’s beliefs. The only problem is that instead of turning right, due to randomness it moved a few times left in a row and found the last cube very slowly. This could potentially be improved by adding a counter of how many times it moves in the same direction in a row , and counteract it by either increasing the displacement at a time, or by moving in the opposite direction to counteract all the movements.

The coordinates from returned at the end of the exploration could be passes into the go\_to\_pose function to directly go to the target, or could be passed into the A\* algorithm to find the best way to each cube avoiding any potential obstacles.



**Navigation**

The implemented code shows Cozmo's movements, enabling it to navigate towards cubes, pick them up, and intelligently avoid obstacles, including walls marked with custom markers. The versatility of Cozmo's navigation behaviour is demonstrated in its adaptability to various maps. An A\* algorithm has been added to find the best possible route back to the original position of Cozmo so that it can drop the cube which it has picked up. This is to ensure that Cozmo can bring back the cube in a search and rescue mission as quickly and safely as possible.

**Navigation towards Cubes:**A computer screen shot of a program

Description automatically generated

The core navigation behaviour involves Cozmo actively seeking and approaching cubes within its environment. The cozmo\_drive\_to\_target function initiates this behaviour. Cozmo's head angle is set to a neutral position, and it employs a behaviour known as "LookAroundInPlace" to scan the surroundings for cubes. Upon detecting a cube, Cozmo moves towards it with precision, maintaining a safe distance. The code ensures that Cozmo halts 70mm away from the cube, optimising for efficient pickup.

**Cube Pickup and Interaction:**

Once Cozmo reaches the target cube, it executes the pickup sequence. The robot.go\_to\_object method is employed to drive Cozmo to the specified distance from the cube. After reaching the desired proximity, Cozmo executes the pickup action, lifting the cube. This demonstrates Cozmo's capability not only to navigate but also to interact with objects in its environment, a crucial aspect of search and rescue operations.

**Obstacle Avoidance - Custom Markers:**

A black background with white text

Description automatically generatedA computer screen shot of text

Description automatically generated

In search and rescue scenarios, the environment can be dynamic, and obstacles may vary. Cozmo is equipped to intelligently avoid obstacles marked with custom markers. The on\_marker\_appeared method is triggered when a custom marker is detected. The robot then utilises the avoid\_marker method, which assesses the distance to the marker and manoeuvres away from it. This behaviour ensures that Cozmo can navigate around obstacles, adapting to diverse and unpredictable terrains. The avoid marker method is used to avoid obstacles if it detects any marked walls, if it gets too close to the wall, the robot will back up a bit and drive either left or right. The reason why we make the Cozmo go randomly left or right is to factor in random maps. If Cozmo doesn’t know the map or has not explored enough of the map, going in random directions to avoid a wall would be the best way for Cozmo to navigate the maps.

**Cliff Avoidance:**

A screenshot of a computer

Description automatically generated

The avoid\_cliff method addresses the detection of a cliff. If a cliff is detected, Cozmo takes evasive action by moving laterally for a brief period, effectively avoiding the hazardous edge. This feature ensures Cozmo's safety by preventing accidental falls., demonstrating its ability to handle challenging terrains safely.

**A\* Algorithm to bring back the cube to original position of Cozmo:**

This code implements the A\* algorithm for Cozmo to find the best route back to its original location after picking up a cube.A screen shot of a computer program

Description automatically generated

The Node class represents a node in the A\* algorithm. Each node has a position attribute, which corresponds to a 3D coordinate, and a parent attribute, pointing to the previous node in the path. The g and h attributes are used for the cost calculations in the algorithm.

The heuristic function calculates the heuristic value for a node based on the Manhattan distance between the node's position and the goal position. This value helps estimate the cost from the current node to the goal.

The get\_neighbors function defines neighbouring positions for a given node within a certain range. It returns a list of neighbouring coordinates.

The a\_star function is the implementation of the A\* algorithm. It takes the robot, start position, and goal position as parameters and returns the optimal path from start to goal using the A\* algorithm. It utilises the Node class, the heuristic function, and the get\_neighbors function.A computer screen shot of a program code

Description automatically generated

The main function initialises the start and goal positions, calls the a\_star function to find the path, and then instructs Cozmo to follow the path using the obtained coordinates. It prints the path, each position Cozmo moves to, and a message when it reaches the goal.

Cozmo's navigation behaviours, as implemented in the provided code, demonstrate adaptability to various maps and environments. The robot can effectively locate and reach a cube on different surfaces and navigate through spaces marked with custom obstacles. This adaptability is crucial for Cozmo's utility in diverse settings, making it suitable for a search and rescue mission.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Testing Case | Expected Outcome | Pass or Fail 5 times | Comments |
| 1 | Cozmo Goes to cube position | Cozmo should be able to go to the position of the cube | P  P  P  P  P | Cozmo was able to always go to any cube position during testing. |
| 2 | Cozmo picks up the cube | Cozmo should be able to pick up the cube | P  P  P  P  P | Cozmo was always able to pick up the cube. |
| 4 | Cozmo Returns to initial position using A\* algorithm | Cozmo should utilise the A\* algorithm and return to its initial position with the cube using the shortest route | F  F P  F  P | Cozmo only managed to find the shortest path twice during testing and a lot of the time would not be able to. |
| 3 | Cozmo avoids walls | Cozmo should be able to avoid walls | P  P  F  F  p | Cozmo was able to avoid the walls most of the times however sometimes Cozmo would bump into a wall and be too close to recognise the custom markers |
| 4 | Cozmo avoids cliffs | Cozmo should be able to avoid cliffs | P  P  P  F  P | Cozmo would be able to avoid walls almost all the time however since Cozmo doesn’t have back sensors for cliff detection Cozmo would sometimes back out too much and fall. |

**Localisation**

Localisation is a unique problem which is needed when tracking the movement of the robot, finding global localisation of the robot depending on high-reliability landmarks and the robot is taken from the map and localised on a different pose (kidnapped robot problem). Localisation can be solved by Kalman Filters, Monte Carlo Localisation algorithms or Neural Network models.

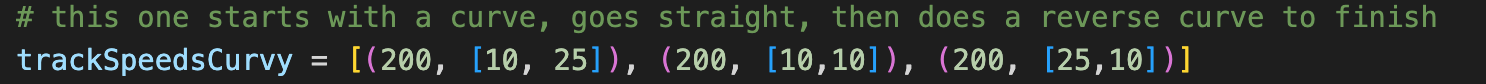
Kalman Filters are used to estimate the state of a robot, which includes its position and orientation (pose) in a given environment. They update estimated state recursively with new sensor data. The probability distributions of the robot's state and sensor measurements are assumed to be Gaussian (normal) distributions. Motion of the robot is assumed to be linear. This linearisation of the movement is achieved by calculating the next pose for a small interval.

Kalman Filters first predict the next state of the robot based on kinematic and previous position then it corrects the estimate of the pose of the robot using sensor measurements and reduces uncertainty. This filter has limitations because it assumes linearity and Gaussian noise but it is not the case in the real world. Vanilla Kalman Filter cannot solve the kidnapped robot problem but the Extended Kalman Filter might help but it also assumes the environment is static. They are straightforward mathematical calculations so implementation of Kalman Filters is simpler than other methods.

The other solution is the Monte Carlo method (The Particle Filter Solutions). They are more robust in handling non-linear and non-Gaussian scenarios. They can be used to estimate the robot's state even in cases of sudden and unpredictable changes like kidnapping. MCL uses randomly generated particles to represent the robot's belief about its pose. Each particle represents a potential robot pose in the environment.

Like the Kalman Filters, the Particle Filters first predict the robot of the pose by moving the particles according to the robot’s motion model using kinematic properties of the robot. After that, in the correction step, particles are assigned weights based on their estimated sensor measurements' likelihood compared to the actual sensor measurements received from the robot's sensors. The particles with higher weights are more likely to represent the true robot pose, while particles with lower weights are less likely. MCL uses a resampling process to draw a new set of particles from the current set of particles. This process helps to concentrate particles around the most likely pose estimates. It can handle uncertainties and accurately estimate the robot's pose in complex environments like the kidnapped robot problem. That’s why the Particle Filter Localisation is chosen for the implementation.

Before the implementation of the localisation algorithm, the noise of the Cozmo robot movement was analysed. At first the distance between the wheels of the robot is measured and it is 450 mm. The robot was tested with the Curvy path for 50 times. While using the given *motion\_logger.py* was tried to be used, some bugs were detected and fixed to use.



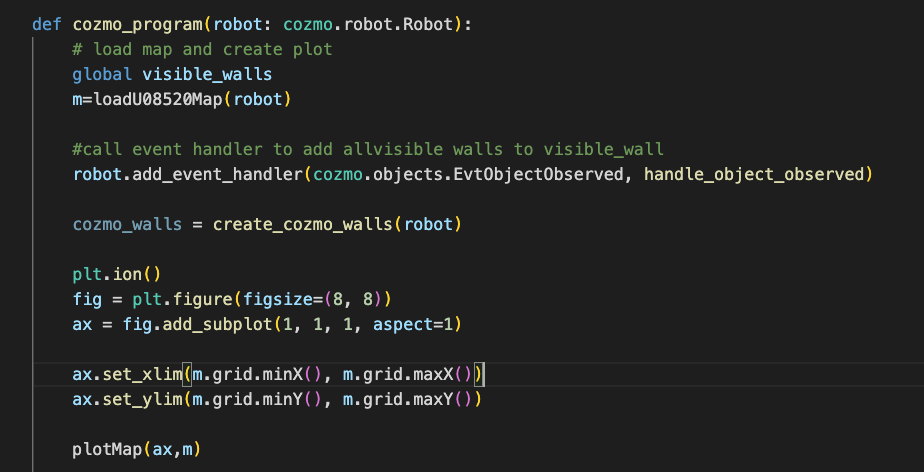


The x axis, y axis and angle differences between the first path and the other are gathered and odometry noises are calculated using Gaussian. The noise on the x axis is 0.01, the noise on the y axis is 0.01 and the noise on angle is 0.001 are selected. All these odometry noises are used in motion update while estimating the next position of the particles along with the movement of the robot.

**Particle Filter Implementation**

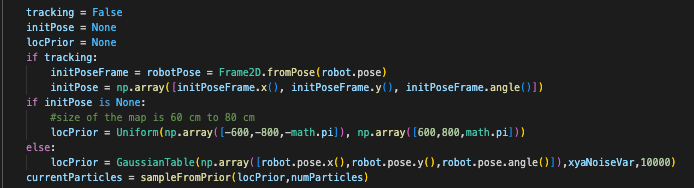
The particle filter localisation is tested on a given map with landmarks. To implement a particle filter localisation algorithm these steps are followed:

1. Load the map - loadU08520Map

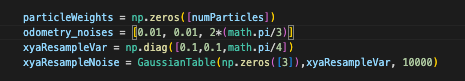


1. Decided the initial distribution of the particles.

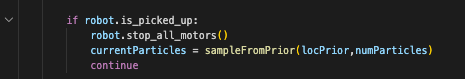
If tracking, initialise the location of the particles depending on the robot's pose. If the initial pose is null, uniformly put particles on the positions of the map.



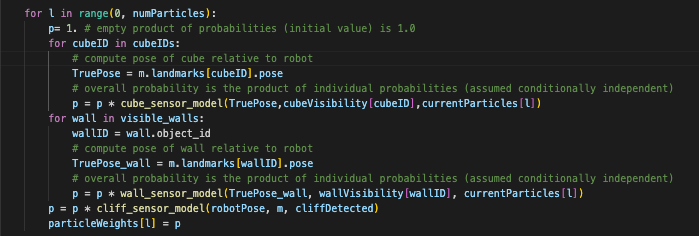
1. Initialise weights of the particles with 0s, odometry noises, resample variation and resample noises.



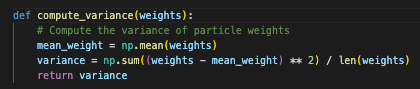
1. If the robot is kidnapped, use prior location samples and reset the current particles.



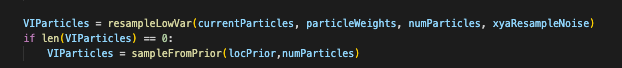
1. Use sensor models for cube, wall and cliff to calculate the likelihood of each particle which is its weight.



7. If the variance of the particle is smaller than 0.0001, it means that the particles converge on the robot's position.



8. Resample from current particles using sampling noise depends on particles’ weights.



9. Using the sensor models for cube, wall and cliff to calculate the weights of the resampled particles. Update the current particles with resampled particles. Get the particle with the maximum weight to estimate the position of the robot.



10. Plot the position of the robot and positions of the particles on the map plot. Save the robot position and particle positions in a log file to save data.



**Simultaneous Localization and Mapping (SLAM)-**

**What is SLAM and its issues**

Before implementing a SLAM algorithm into the Cozmo robot, it is essential to first understand what it is and the possible issues it brings. Simultaneous Localization and Mapping is a fundamental concept in robotics and autonomous systems, where a robot (like the Cozmo) simultaneously builds a map of an unknown environment while tracking its own location within that map. This dual challenge poses several critical problems, as outlined in key texts by Thrun (2002) and Siegwart, Nourbakhsh, and Scaramuzza (2011).

One significant problem in SLAM is loop closure and the accumulation of drifts. As a robot navigates and maps an environment, small errors in its movement and sensor measurements can accumulate over time, leading to a phenomenon known as drift. This drift can cause the robot's internal map to become increasingly inaccurate. To counter this, the robot must recognize previously visited locations—a process known as loop closure. Achieving accurate loop closure is essential for correcting these accumulated errors and aligning the map with the real environment. Thrun (2002) discusses the importance of probabilistic approaches in addressing this issue, where the robot uses statistical methods to estimate its position and revises the map accordingly.

Landmark association is another critical challenge in SLAM. As a robot navigates, it needs to identify and differentiate various landmarks to orient itself and construct the map. The challenge, as highlighted by Siegwart et al. (2011), lies in disambiguating one landmark from another, especially in environments where many landmarks may appear similar. Effective landmark association requires robust algorithms for feature detection and matching, allowing the robot to reliably identify landmarks and update its map consistently.

The requirement for real-time updates is a further complexity in SLAM. As a robot moves and senses its environment, it must rapidly update its position (pose) and integrate new sensor data into its map. This demands significant computational resources and efficient algorithms, as the process must be fast enough to keep pace with the robot's movements and the dynamic changes in its surroundings. Siegwart et al. (2011) emphasize the need for algorithms that can operate effectively in real-time, ensuring that the robot's map and location estimates are always up-to-date.

**Possible Solutions**

Several solutions have been developed to address the inherent challenges of constructing a map of an unknown environment while simultaneously tracking a robot's location within it. The most popular approaches to SLAM are generally derived from three main filtering techniques: the Kalman filter, the particle filter, and the extended information filter, each having its own strengths and specific application scenarios as discussed in Siegwart, Nourbakhsh, and Scaramuzza (2011).

The Kalman Filter approach in SLAM is heavily reliant on landmarks for accurate mapping and localization. In this method, the environment is represented by a set of landmarks, whose positions are estimated and updated frequently. The Extended Kalman Filter (EKF) is a standard adaptation of this technique used in SLAM. It linearizes about the current mean and covariance in a recursive way to update the robot's state and the map. However, a major concern in the Extended Kalman Filter approach is data association, which involves correctly matching sensor observations with the corresponding landmarks in the map. This requires the landmarks to be distinct and easily identifiable, as incorrect associations can lead to significant errors in the map and the robot's estimated position.

The Particle Filter approach, alternatively known as the Monte Carlo Localization method, is better suited for environments where updates can be less frequent and where the environment is more chaotic. In this method, a set of particles, each representing a possible state of the robot, is used to estimate the robot's trajectory and the map. The strength of the particle filter lies in its ability to handle non-linear, non-Gaussian problems effectively. However, a major concern in this approach is loop closure, particularly in symmetric environments where the robot has difficulty in recognizing if it has returned to a previously visited location.

The Graph-Based SLAM, as suggested by Siegwart et al. (2011), utilizes a graph structure where nodes represent robot poses and edges represent spatial relationships between these poses. This method is particularly effective in large-scale environments. Data association in graph-based SLAM is built-in, with the assumption that landmarks are more-or-less independent at large distances. This implies that the method works best with sparse landmarks. The graph-based approach efficiently handles the SLAM problem by optimizing the graph to find the configuration of nodes that best explains the spatial relationships.

**EKF SLAM and algorithms**

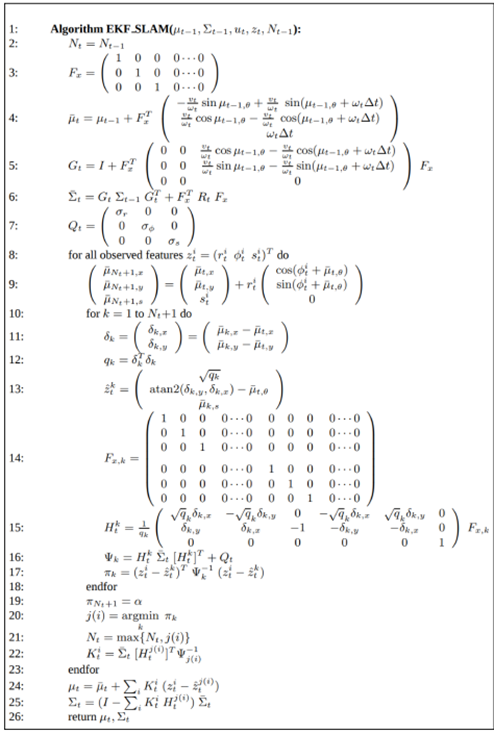
For the implemented probabilistic algorithm required, the chosen method was the EKF SLAM approach. The core idea of EKF SLAM is to amalgamate the robot's position and the map into a single state vector, along with a corresponding covariance matrix. This state vector and covariance matrix evolve over time as the robot moves and perceives its environment. The Extended Kalman Filter is employed to update these estimates iteratively based on the robot's sensor readings and motion model.

In EKF SLAM, data association plays a crucial role. New landmarks can be added at any step of the process. This is achieved by 'growing' the state vector and covariance matrix, incorporating the new landmark. Initially, the covariance associated with a new landmark is placed only on the diagonal of the matrix. This method allows the EKF SLAM to handle dynamic environments where new features may appear.

The use of saliency is another important aspect of EKF SLAM. Saliency involves determining the likelihood of observing a particular landmark given the current state and map. Siegwart et al. (2011) discuss employing a threshold (τ) to decide whether a landmark should be included in the map. This process helps in filtering out irrelevant or less significant features, thereby maintaining the efficiency of the map.

The Extended Kalman Filter in EKF SLAM effectively boils down to iteratively multiplying sensor model Gaussians over time. This iterative process involves updating the state estimate and covariance matrix based on the latest sensor readings, thereby refining the robot's understanding of its position and the map.

The algorithms to be used can be seen from Thrun, et al.:



The EKF SLAM algorithm integrates the state prediction, covariance prediction, Kalman Gain computation, state update, covariance update, and linearization into a cohesive process. Drawing from Thrun's (2002) work on probabilistic robotics, the following steps are an intrinsic part of the EKF SLAM algorithm:

· State Prediction:

This is where the algorithm predicts the next state based on the current state and motion commands. The state prediction formula can be found on line 4 of the provided image

· Covariance Prediction:

The algorithm predicts the covariance matrix for the next state, accounting for the motion of the robot and the uncertainty in that motion. This step is represented by line 6 in the image.

· Kalman Gain:

The Kalman Gain is calculated to integrate the measurement into the state estimate. It determines how much the predictions should be adjusted based on the new measurements. This calculation is shown on line 15 of the image.

· State Update:

After receiving measurements, the state estimate is updated to better reflect the new information. This updated estimate is more accurate than the prediction alone. The formula for updating the state is given on line 23 of the image.

· Covariance Update:

The covariance matrix is updated in tandem with the state update. This reflects the reduced uncertainty after incorporating the measurement. The updated covariance matrix is shown on line 24 in the image.

· Linearization:

Due to the non-linear nature of the robot's movement and sensor models, the EKF SLAM needs to linearize these models to apply the Kalman filter. This process is inherent in the calculation of the Jacobians (lines 5 and 14), which are used in the prediction and update steps respectively.

**Code**

The code done follows the steps above. In this code: predictState function represents the State Prediction. stateCovariance update after predictState is the Covariance Prediction. kalmanGain calculation is the Kalman Gain. robotState update with kalmanGain is the State Update. stateCovariance update after applying kalmanGain is the Covariance Update. stateJacobian and landmarkJacobian calculations represent Linearization.

sigmaPosition, sigmaOrientation, sigmaLandmarkPosition, sigmaLandmarkAngle =  
  
 def calculateVelocities(measuredPoses, timeStep):  
 if timeStep == 0:  
 velocityX = 0  
 velocityY = 0  
 angularVelocity = 0  
 else:  
 xt = measuredPoses[timeStep].x()  
 yt = measuredPoses[timeStep].y()  
 theta\_t = measuredPoses[timeStep].angle()  
 xt\_minus\_1 = measuredPoses[timeStep-1].x()  
 yt\_minus\_1 = measuredPoses[timeStep-1].y()  
 theta\_t\_minus\_1 = measuredPoses[timeStep-1].angle()  
 velocityX = xt - xt\_minus\_1  
 velocityY = yt - yt\_minus\_1  
 angularVelocity = theta\_t - theta\_t\_minus\_1  
 return (velocityX, velocityY, angularVelocity)  
  
 def predictState(currentState, measuredPoses, timeStep, transformationMatrix):  
 statePrediction = currentState  
 velocities = calculateVelocities(measuredPoses, timeStep)  
 motionModel = np.array(velocities)  
 transformedMotion = np.matmul(transformationMatrix.T, motionModel)  
 statePrediction += transformedMotion  
 return statePrediction  
  
 # Transformation matrix  
 transformationMatrix = np.eye(3)  
  
 # Initial robot state  
 robotState = np.array([0.0, 0.0, 0.0])  
 stateCovariance = np.eye(3) \* 100.0  
  
 # Initial noise  
 motionNoiseCovariance = np.diag([sigmaPosition, sigmaPosition, sigmaOrientation])  
 measurementNoiseCovariance = np.diag([sigmaLandmarkPosition, sigmaLandmarkPosition, sigmaLandmarkAngle])  
  
 numDataPoints = len(robotFrames)  
 robotPoseDict = dict(robotFrames)  
 landmarkPoseDict = dict(cubeFrames)  
 timeSteps = np.linspace(0, numDataPoints-1, numDataPoints)  
 landmarkObservations = np.zeros((numDataPoints, 3))  
 stateEstimates = np.zeros((numDataPoints, 3))  
 stateCovariances = np.zeros((numDataPoints, 3))  
  
 identityMatrix = np.identity(3)  
  
 currentTimeStep = 0  
 for timestamp, frame in robotFrames:  
 # State Prediction  
 predictedState = predictState(robotState, robotPoseDict, timestamp, transformationMatrix)  
 stateJacobian = identityMatrix # Linearization around current state is identity since state transition is linear  
 stateCovariance = stateJacobian @ stateCovariance @ stateJacobian.T + (transformationMatrix.T @ motionNoiseCovariance @ transformationMatrix)  
  
 # Observation available?  
 kalmanGain = np.zeros\_like(stateCovariance) # Initialize Kalman Gain  
 if timestamp in landmarkPoseDict:  
 # Landmark Initialization  
 if robotState.size == 3:  
 transformationMatrix = np.append(transformationMatrix, np.zeros((3, 3)), axis=1)  
 # Landmark estimate  
 initLandmarkX = robotState[0] + landmarkPoseDict[timestamp].x() - frame.x()  
 initLandmarkY = robotState[1] + landmarkPoseDict[timestamp].y() - frame.y()  
 initLandmarkTheta = robotState[2] + landmarkPoseDict[timestamp].angle() - frame.angle()  
 predictedState = np.append(predictedState, [initLandmarkX, initLandmarkY, initLandmarkTheta])  
 stateCovariance.resize((motionNoiseCovariance.shape[0]+3, motionNoiseCovariance.shape[1]+3))  
 stateCovariance[-1, -1] = sigmaOrientation  
 stateCovariance[-2, -2] = sigmaPosition  
 stateCovariance[-3, -3] = sigmaPosition  
  
 # State Update  
 landmarkMeasurement = np.array([landmarkPoseDict[timestamp].x(), landmarkPoseDict[timestamp].y(), landmarkPoseDict[timestamp].angle()])  
 predictedMeasurementDiff = predictedState[3:] - predictedState[:3]  
 landmarkTransformationMatrix = np.roll(transformationMatrix, 3, 1)  
 landmarkJacobian = np.append(-identityMatrix, identityMatrix, axis=1) @ landmarkTransformationMatrix  
 predictedMeasurementCovariance = landmarkJacobian @ stateCovariance @ landmarkJacobian.T + measurementNoiseCovariance  
 # Kalman Gain  
 kalmanGain = stateCovariance @ landmarkJacobian.T @ np.linalg.inv(predictedMeasurementCovariance)  
 robotState = predictedState + kalmanGain @ (landmarkMeasurement - predictedMeasurementDiff)  
 else:  
 robotState = predictedState  
  
 # Covariance Update  
 stateCovariance = (identityMatrix - kalmanGain @ landmarkJacobian) @ stateCovariance if timestamp in landmarkPoseDict else stateCovariance  
  
 landmarkObservations[currentTimeStep] = landmarkMeasurement if timestamp in landmarkPoseDict else np.zeros(3)  
 stateEstimates[currentTimeStep, :2] = robotState[:2]  
 stateCovariances[currentTimeStep, :2] = stateCovariance[0, 0], stateCovariance[1, 1]  
 currentTimeStep += 1

The code was made with the idea of removing redundant recalculations of the identity matrix by precomputing it outside of the loop, initialize the kalmanGain matrix with zeros to prepare it for cases where an observation is not available, has a condition to only update stateCovariance with the Kalman Gain and landmark Jacobian if an observation is available and ensure that landmarkObservations is updated with zeros when no observation is available to maintain the array's structure.

**Evaluation**

The Extended Kalman Filter (EKF) SLAM method, as used in the code, has strengths and weaknesses, which reflect on the method.

· Strengths of EKF SLAM and the Code:

EKF SLAM is designed for real-time applications, and the code reflects this by sequentially updating the state and covariance matrix as new measurements arrive, without the need for batch processing (Thrun, 2002).

The code leverages the EKF's ability to handle non-linear system models by using Jacobians for linearization. EKF SLAM is well-suited for the non-linear nature of robot motion and sensing, allowing it to perform well in many practical robotic applications (Siegwart et al., 2011).

Incremental Updates: The code benefits from the incremental update nature of the EKF, which is computationally more efficient than reprocessing all data upon each new measurement. This is particularly important in mobile robotics where computational resources are often limited (Thrun, 2002).

· Weaknesses of EKF SLAM and the Code:

The use of Jacobians for linearization introduces linearization errors, especially when dealing with highly non-linear functions or when the robot operates in environments that induce large measurement errors. These errors can lead to inaccuracies in the estimated state and map (Siegwart et al., 2011).

The code assumes a simplistic data association strategy, which could fail in environments with ambiguous or repetitive landmarks. EKF SLAM's performance heavily depends on correct data association, and any misassociations can lead to filter divergence (Thrun, 2002; Siegwart et al., 2011).

EKF SLAM, and by extension the code, can be sensitive to initialization and prone to divergence if the initial estimates are inaccurate or if the robot encounters unexpected observations that deviate significantly from the predicted model (Siegwart et al., 2011).