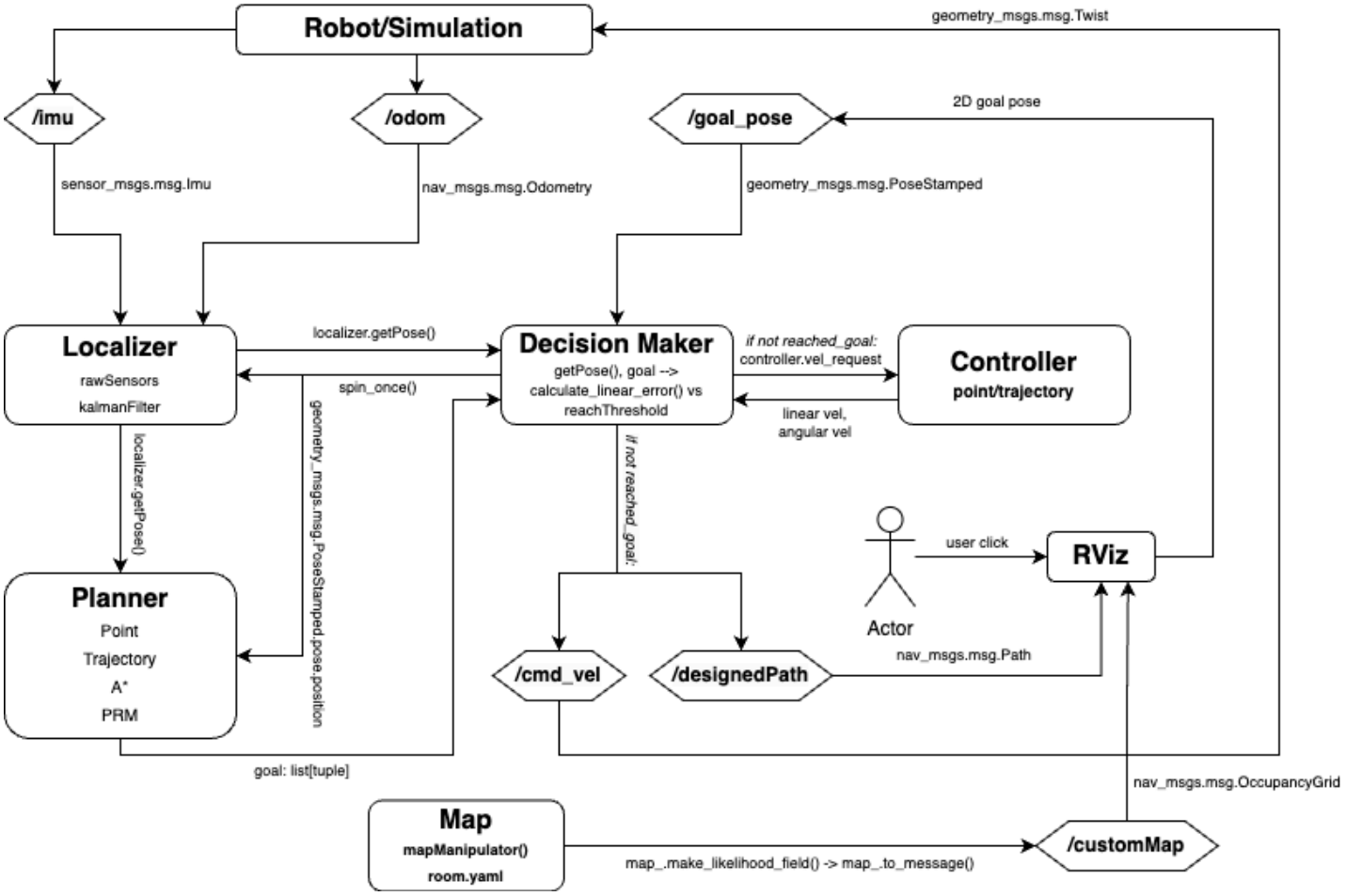
MTE544 Final Report

Xue, Kevin

20814292

Bonus Attempted?: **No**

Part 1 - Stack: the flowchart



The stack is designed for robotic navigation and path planning, incorporating multiple components to handle decision-making, localization, planning, and control. Below is a breakdown of how the stack operates:.

1. decisions.py:
    - The **decision_maker** class serves as the central decision-making hub. It decides the type of planner to use (e.g., PRM_PLANNER, ASTAR_PLANNER, etc.) and delegates tasks to the localizer, planner, and controller.
    - The module processes the start/current and goal locations, and based on a predefined goal or real-time user input (published from the ROS topic /goal_pose), it invokes the planner to calculate a path.
2. planner.py:
    - The **planner** class is responsible for generating the path based on the planner type (Point, Trajectory, A*, PRM). The Point and Trajectory planners were introduced in Lab 2, and the A* planner was introduced in Lab 4.
    - In the final exam, the PRM planner type was implemented as an alternative to the A* planner type, which leverages a likelihood field as the graph. With PRM, a graph is constructed by randomly sampling points, generating edges between them, and ensuring that no edges collide with obstacles. The planner then uses these nodes and edges to perform an A* search for computing the trajectory from the start to the goal.
    - By generating random sample points and ensuring connectivity, PRM provides a more scalable and time-efficient pathfinding approach, particularly in large or sparse environments, compared to grid-based methods using the likelihood field. However, based on the parameters available for tuning (N_SAMPLE, N_KNN, MAX_EDGE_LEN), it can result in no solution being found. This susceptibility is not present using the grid-based method.
3. localization.py:

- The **localization** class is responsible for determining the current state of the robot (which can include position and orientation, linear/angular velocity and acceleration). This is executed by calling **localizer.getPose()**.
- In Lab 2, the localization of the robot was done using raw sensors from the wheel encoders (**nav_msgs.msg.Odometry** published on the /odom topic) on the robot.
- In Lab 3, the Extended Kalman Filter (EKF) was introduced to improve upon the state estimation task by fusing measurements from the wheel encoders (**nav_msgs.msg.Odometry** published on the /odom topic) and from the IMU (**sensor_msgs.msg.imu** published on the /imu topic) with the previous state. The EKF relies on the assumption of the Markov property to perform the fusion of sensor data with the most recent state (as opposed to a longer history of states). This was thought to be an improvement rather than just relying on raw sensor data just from the wheel encoders, as the real life system is susceptible to phenomena such as sensor drift and wheel slippage, which results a compounding of state estimation error over time.

4. controller.py:
   - The **controller** class takes the planned goal in the form of a single point or a trajectory with a set of points and converts it into actionable commands for the robot's motors. It does this by publishing a geometry_msgs.msg.Twist() object to the through the /cmd_vel topic.

Part 2 - PRM implementation and A* implementation

Results

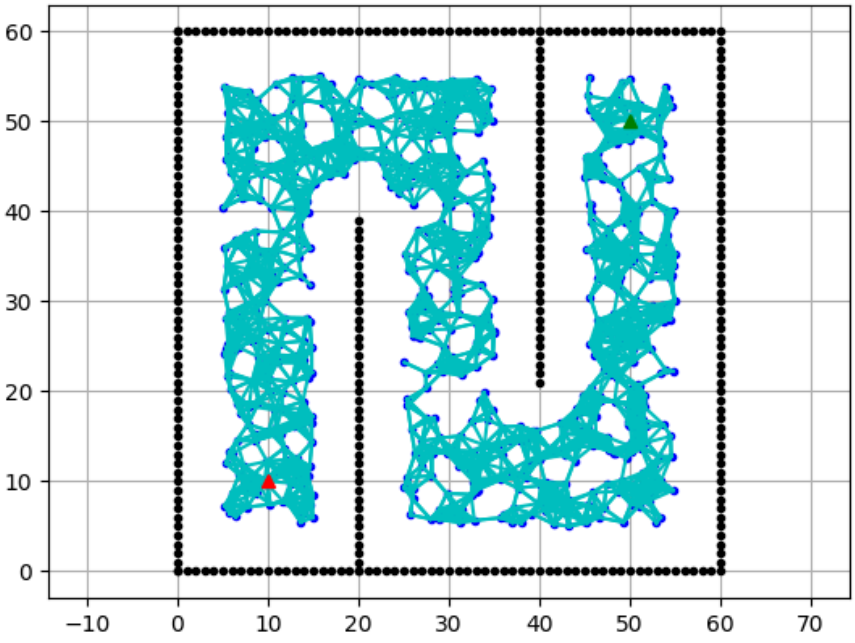


Figure 1: 1st run of: `python3 probabilistic_road_map.py`

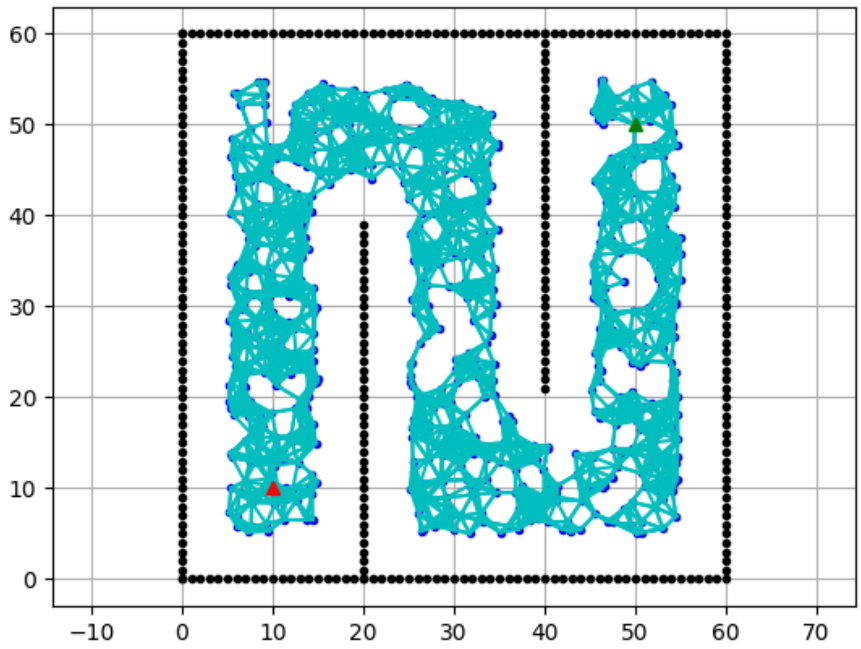


Figure 2: 2nd run of: `python3 probabilistic_road_map.py`

Analysis

The Probabilistic Roadmap operates as follows:

1. Sample Generation: **generate_sample_points**
    - This function handles creates random samples within the free space, avoiding obstacles and ensuring they are spaced at least the robot's radius away from any obstacles. It uses a KD-Tree to structure the cells which contain obstacles for efficient querying to check if a given point is valid (i.e., not too close to an obstacle).
2. Roadmap Creation: **generate_road_map**
    - Once the sample points are generated, this function creates the edges between these sample points. The edges are formed by connecting the sample points that are within a specified maximum edge length (**MAX_EDGE_LEN**), ensuring no collision with obstacles. The collision check is performed using the is_collision function, which evaluates if there's a clear path between two nodes.

To integrate the PRM with A* planning and the remainder of the stack for the robot, **PRM_PLANNER** was selected to be the planner type, the start and end pose were converted from cartesian coordinates to cell coordinates, and the **prm_graph** and **search_PRM** functions were used to generate the PRM graph and perform the A* search on that generate graph. This would yield a path/trajectory, which is then converted from cell coordinates back to cartesian coordinates, before return the result to the **decision_maker** class. From here, the logic is the exact same as the trajectory planner from Lab 2.

Part 4 - Discussion, Testing, and Integration of PRM + A*

Results

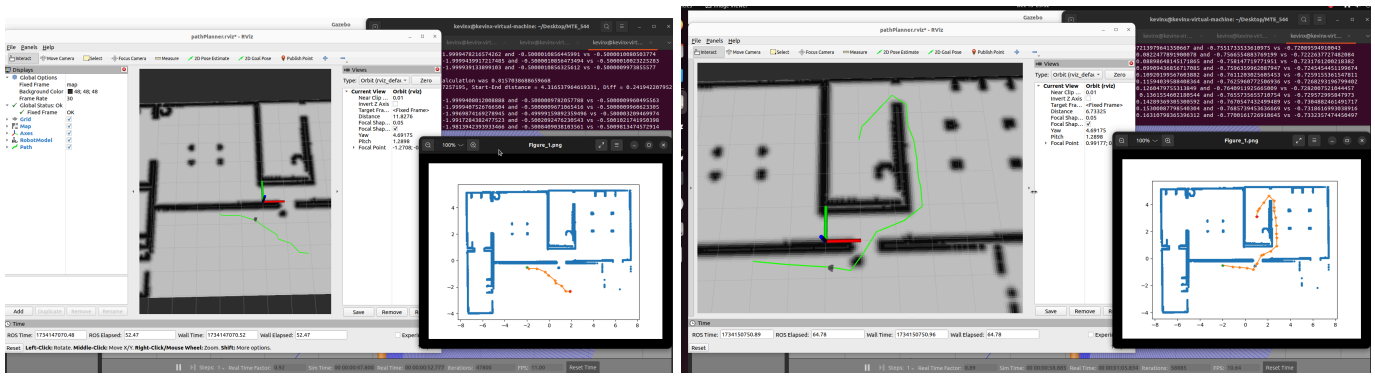


Figure 3: 1st (left) and 2nd (right) run with **PRM + A*** (2nd run is shown in the video)

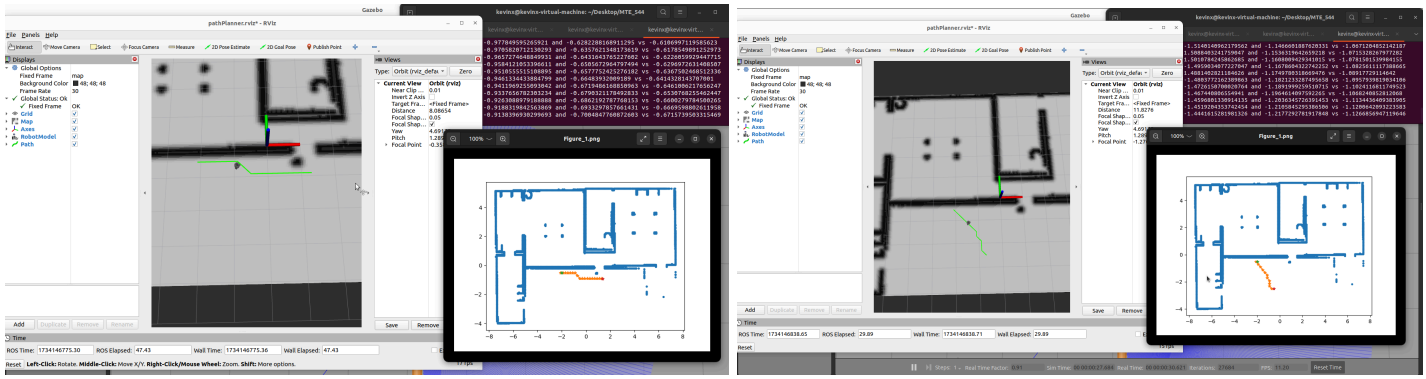


Figure 4: 1st (left) and 2nd run (right) with **A***

The key parameters for PRM are the number of samples (**N_SAMPLE**), the number of neighbors for each node (**N_KNN**), and the maximum edge length (**MAX_EDGE_LEN**). Similarly, A* requires tuning of **scale_factor** and **laser_sig**.

1. **N_SAMPLE**: I started with a large number of samples to ensure a connected graph, but found that increasing the number beyond 1000 didn't result in much improvement in path quality. A value of 1000 seemed to strike a balance between path accuracy and computational cost.
2. **N_KNN**: A larger number of neighbors provides a denser roadmap but increases the complexity of the graph and the search time. I found that 10 neighbors was sufficient to create a well-connected roadmap.
3. **MAX_EDGE_LEN**: The maximum edge length controls how far apart two nodes can be connected. I began with a value of 3 meters but reduced it to 2 meters to prevent edges from spanning large distances, ensuring smoother, more feasible paths.

For tuning the parameters, I conducted several trials and visualized the roadmap as well as the calculated path to ensure that the graph was well-connected and the path was optimal. Plots were generated to check the effectiveness of different values for each of the parameters, with 1000 samples, 10 neighbors, and a maximum edge length of 2 meters providing good connectivity in the PRM without excessive computation time.

4. The **laser_sig** parameter essentially controls the spread or smoothing of this probability field around obstacles. A high value of **laser_sig** results in a larger "uncertainty" or buffer zone around obstacles, leading to more cautious and conservative planning behavior. Conversely, a lower **laser_sig** allows the planner to be more aggressive and approach obstacles more closely.

5. The **scale_factor** determines the resolution of the occupancy grid map. A **scale_factor** is typically a ratio between the robot's world coordinates and the resolution of the grid map. A low **scale_factor** corresponds to a high map resolution, meaning the environment is divided into smaller cells, providing a more detailed representation of obstacles and free space. A high **scale_factor** results in a lower map resolution, where fewer, larger grid cells represent the environment.

A series of tests with different **scale_factor** and **laser_sig** values were performed, and the paths were visualized to check for any unnecessary detours.

Analysis

Performance of PRM vs. A* on the Occupancy Grid

1. Path Quality: The A* on the Occupancy Grid path had better overall smoothness, particularly for more complex goal poses.
2. Computation Time: While tuning either method, it was evident that PRM was consistently taking longer to compute compared to A* + Occupancy Grid. This is because with PRM, pathfinding is faster for large, complex environments. However, with A*, since there is no upfront graph-building computation, it can be faster for smaller, simpler maps.
3. Success Rate: A* on the Occupancy Grid had a 100% success rate of finding a path, while the PRM did not have a 100% success rate, depending on the parameter set at the time.

Overall Performance of the Entire Stack

The overall performance of the stack, including the PRM + A* planner, PID controller, and EKF, was not as satisfactory in navigating the robot within simulation environment as I had hoped. From all the Figures shown above, it is clear that with the current stack, even after I had attempted to tune the PID and EKF, still results in extremely inconsistent and undesirable behaviour when it comes to path following. After about 4-5 seconds of simulation, the robot usually starts to deviate heavily from the calculated path.

Appendix

Video Link: 2024-12-13 23-31-14.mkv

Part 2

```
# [Part 2] TODO The radius of the robot and the maximum edge lengths are given in [m], but the map is
given in cell positions.
      # Therefore, when using the map, the radius and edge length need to be adjusted for the
resolution of the cell positions
      # Hint: in the map utilities there is the resolution stored

      # Adjust the robot radius and max edge length based on the map resolution
      resolution = m_utilities.getResolution()
      robot_radius /= resolution
      max_edge_len /= resolution
```

```
# [Part 2] TODO Create list of sample points within the min and max obstacle coordinates
   # Use rng to create random samples.
   # NOTE: by using rng, the created random samples may not be integers.
   # When using the map, the samples should be indices of the cells of the costmap, therefore remember
to round them to integer values.
   # Hint: you may leverage on the query function of KDTree to find the nearest neighbors

   # Generate random points within the bounding box of the obstacles
   sample_x, sample_y = [], []

   while len(sample_x) <= N_SAMPLE:
       x_rand = round(rng.uniform(min(ox), max(ox))) # generate random x value
       y_rand = round(rng.uniform(min(oy), max(oy))) # generate random y value

       if obstacle_kd_tree.query([x_rand, y_rand])[0] > rr: # ensure that samples are spaced at least
rr away from any obstacles
           sample_x.append(x_rand)
           sample_y.append(y_rand)
```

```
# [Part 2] TODO Add also the start and goal to the samples so that they are connected to the roadmap
   sample_x.extend([sx, gx])
   sample_y.extend([sy, gy])

   return [sample_x, sample_y]
```

```
# [Part 2] TODO Check where there would be a collision with an obstacle between two nodes at sx,sy and
gx,gy, and wether the edge between the two nodes is greater than max_edge_len
   # Hint: you may leverage on the query function of KDTree

   # Compute the Euclidean distance
   distance = math.hypot(gx - sx, gy - sy)
   if distance > max_edge_len: # return collision if distance exceeds max_edge_len
       return True
```

```
    # Check for collisions along the path
    num_points = int(distance / rr)
    x_points = np.linspace(sx, gx, num_points)
    y_points = np.linspace(sy, gy, num_points)

    for x, y in zip(x_points, y_points):
        dist, _ = obstacle_kd_tree.query([x, y])
        if dist <= rr: # collision detected
            return True

    return False  # No collision
```

```
#[Part 2] TODO Generate roadmap for all sample points, i.e. create the edges between nodes (sample points)
    # Note: use the is_collision function to check for possible collisions (do not make an edge if there is collision)
    # Hint: you may ceate a KDTree object to help with the generation of the roadmap, but other methods also work

    for i, (sx, sy) in enumerate(zip(sample_x, sample_y)):
        neighbors = kd_tree.query([sx, sy], k=N_KNN + 1)[1] # get index of N_KNN closest neighbors
        edges = []
        for neighbor in neighbors[1:]:  # Skip self in the neighbors list
            gx, gy = sample_x[neighbor], sample_y[neighbor]
            if not is_collision(sx, sy, gx, gy, rr, obstacle_kd_tree, max_edge_len):
                edges.append(neighbor)
        road_map.append(edges)

    return road_map
```

Part 3

```
# [Part 3] TODO Use the PRM and search_PRM to generate the path
        # Hint: see the example of the ASTAR case below, there is no scaling factor for PRM
        if _type == PRM_PLANNER:
            # Generate the PRM graph
            start_time = time.time()
            sample_points, roadmap = prm_graph(
                startPose,
                endPose,
                self.obstaclesListCell,
                ROBOT_RADIUS,
                rng=np.random.default_rng(),
                m_utilities=self.m_utilities
            )

            path_ = search_PRM(sample_points, roadmap, startPose, endPose)
```

```
            end_time = time.time()
            print(f"The time took for PRM calculation was {end_time - start_time}")
```

```
# [Part 3] TODO Complete the this function so that it is adapted to search a graph provided as PRM instead of a grid maze
# Hint: look at the original A* search above and adapt to the PRM
def search_PRM(points, prm, start, end):
    """
        Returns a list of tuples as a path from the given start to the given end in the given maze
        :param prm: probabilistic roadmap
        :param start: starting position as cell positions (indexes of the costMap)
        :param end: goal position as cell positions (indexes of the costMap)
        :return: path as tuples from the given start to the given end
        :return:
    """
    # Create start and end node with initized values for g, h and f
    start_idx = points.index(tuple(start))
    start_node = Node(None, start_idx)
    start_node.g = start_node.h = start_node.f = 0

    end_idx = points.index(tuple(end))
    end_node = Node(None, end_idx)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize open and closed lists
    open_list = [start_node]
    closed_list = []

    # Loop until the end is found or no more nodes are available
    while open_list:
        # Get the node with the lowest f value from the open list
        current_node = min(open_list, key=lambda node: node.f)
        open_list.remove(current_node)
        closed_list.append(current_node)

        # Check if we have reached the goal
        if current_node == end_node:
            path = []
            while current_node:
                path.append(points[current_node.position])
                current_node = current_node.parent
            return path[::-1]  # Reverse the path to start-to-end order

        # Iterate over neighbors of the current node
        for neighbor_idx in prm[current_node.position]:
            # Check if the neighbor is already in the closed list
            if any(node.position == neighbor_idx for node in closed_list):
```

```
            continue

        # Calculate g (cost to reach this neighbor), h (heuristic), and f values
        g = current_node.g + np.linalg.norm(
            np.array(points[current_node.position]) - np.array(points[neighbor_idx])
        )
        h = np.linalg.norm(np.array(points[neighbor_idx]) - np.array(points[end_idx]))
        f = g + h

        # Check if the neighbor is already in the open list
        existing_node = next((node for node in open_list if node.position == neighbor_idx), None)
        if existing_node:
            # If the new path to the neighbor is better, update it
            if g < existing_node.g:
                existing_node.g = g
                existing_node.h = h
                existing_node.f = f
                existing_node.parent = current_node
        else:
            # Otherwise, create a new node and add it to the open list
            neighbor_node = Node(current_node, neighbor_idx)
            neighbor_node.g = g
            neighbor_node.h = h
            neighbor_node.f = f
            open_list.append(neighbor_node)

    # If no path is found, return an empty list
    return []
```

Part 4

```
# [Part 4] TODO Use the EKF localization instead of rawSensors
        self.localizer=localization(kalmanFilter)
```

```
# [Part 4] TODO PID gains if needed
        self.controller=trajectoryController(klp=0.6, klv=0.2, kli=0.6, kap=1.2, kav=0.2, kai=0.6)
```