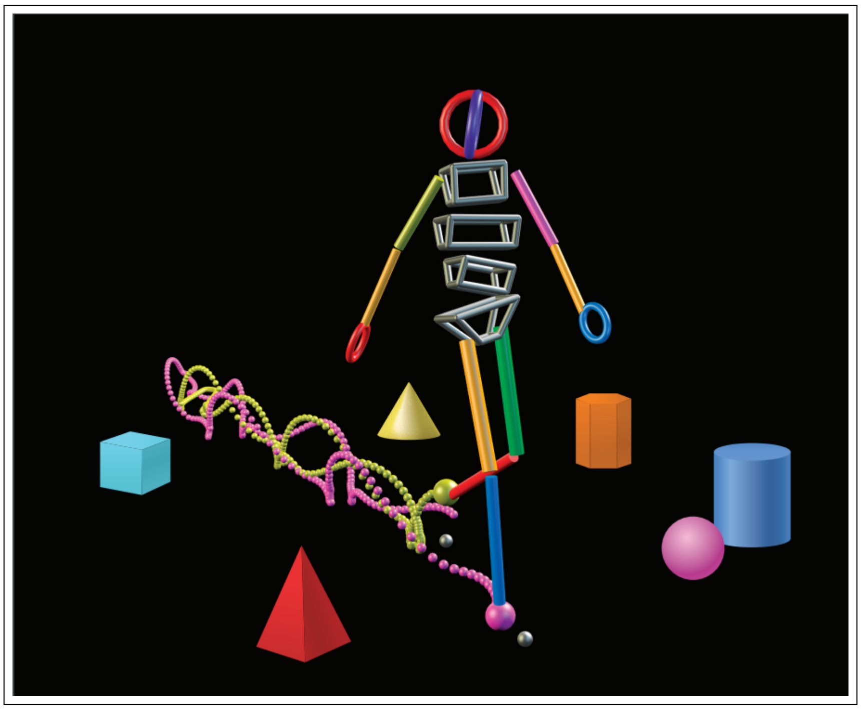
Algorithmic Robot Motion Planning - 236901

Homework #3: Manipulation & Inspection Planning



Name	ID	Email
Niv Ostroff	212732101	nivostroff@campus.technion.ac.il
Yotam Granov	211870241	yotam.g@campus.technion.ac.il

Faculty of Computer Science
Technion - Israel Institute of Technology
Winter 2022/23

(2) Robot Manipulator: Motion & Inspection Planning

In the previous assignment, you were asked to implement the RRT algorithm for a point robot. Here, you will experiment with the challenges when working with a manipulator robot, and will be required to complete both motion-planning and inspection-planning tasks.

Part 2.1 - Code Overview

The starter code is written in Python and depends on numpy, matplotlib, imageio and shapely. We advise that you work with virtual environments for python. If any of the packages are missing in your python environment, please use the relevant command:

```
    pip install numpy
    pip install matplotlib
    pip install imageio
    pip install Shapely
```

You are provided with the following files:

- **run.py** - Contains the main function. Note the command-line arguments that you can provide.
- **Robot.py** - Robot management class, containing all functions related to the robot, including information on links and end-effector.
- **MapEnvironment.py** - Environment management class, containing all functions related to the environment itself.
- **map_mp.json** - An environment (map) file for a motion-planning task. Contains the map's dimensions, (workspace) obstacles, and start and goal locations (in the C-space !).
- **map_ip.json** - An environment (map) file for an inspection-planning task. Contains the map's dimensions, (workspace) obstacles, (workspace) inspection points, and start location (in the C-space !).
- **RRTMotionPlanner.py** - RRT planner for motion planning. Logic to be filled by you.
- **RRTInspectionPlanner.py** - RRT planner for inspection planning. Logic to be filled by you.
- **RRTTree.py** - Management class for the RRT tree for both motion and inspection planning.

The following are examples of how to run the code for motion planning/inspection planning tasks:

```
    python run.py -map map_mp.json -task mp -ext_mode E2 -goal_prob 0.05
    python run.py -map map_ip.json -task ip -ext_mode E2 -coverage 0.5
```

Part 2.2 - Robot Modelling

In this exercise, we will work with a manipulator robot. The robot has four degrees of freedom with four links of fixed size (defined in `Robot.py`). You will start by implementing a few features that are necessary to operate a robotic manipulator. In your writeup, **detail your approach** for implementing each of the following functions in `Robot.py`:

1. `compute_distance` - Your task is to compute the Euclidean distance between the two given configurations, and return the result.

2. `compute_forward_kinematics` - Given a configuration (where angles are in radians and ranged in $[-\pi, \pi]$), your task is to compute the position of each link, including the end-effector (and excluding the origin), and return it. Use the notation in the figure to understand the geometric contribution of each angle to the location of the link. Make sure that the output of this function is a `numpy` array of the shape $(4, 2)$, where the row i represents the i-th link, and contains the x-axis and y-axis coordinates w.r.t. the environment's origin ($[[x_1, y_1], \cdots, [x_4, y_4]]$). **Clarification:** The last row should represent the end-effector, so make sure you receive the start and goal locations that are in the figure.

3. `validate_robot` - For a given set of all robot's links locations (including the origin), your task is to validate that there are no self-collisions of the robot with itself. Return `True` for no self-collisions (**Notice!** you are not required to check for collisions with obstacles or with the environment's boundaries! This is already implemented in the `MapEnvironment.py`).

Solution:

1. For the `compute_distance` function, we first found the difference (element-wise) of the two given configuration vectors using the `numpy.subtract` function from the `numpy` library, and then took the Euclidean norm of the resulting difference vector with the `numpy.linalg.norm` function. The full code implementation looks as follows:

```python
    def compute_distance(self, prev_config, next_config):
        diff_vec = np.subtract(next_config, prev_config)
        return np.linalg.norm(diff_vec,2)
```

2. For the `compute_forward_kinematics` function, we sought to calculate the Cartesian position of the end-effector in the workspace reference frame by iteratively finding the positions of each preceding link, starting at the origin, given a configuration $[\theta_1, \theta_2, \theta_3, \theta_4]$ and a robot arm with link lengths $[L_1, L_2, L_3, L_4]$. For the first link, which is fixed to the origin at one side, it's clear that the joint angle given in the configuration is identical to the angle in the workspace frame, and so the Cartesian position of the second joint (i.e. the other end of the first link) would just be:

$$(x_1, y_1) = L_1 \cdot (cos(\theta_1), sin(\theta_1))$$

For the next link, we need to consider that the configuration angle of the second joint is given relative to the current orientation of the first link (not relative to the workspace frame), and so we used the provided method `compute_link_angle` of the `Robot` class to calculate the angle of the second link relative to the workspace frame. This function just adds the given angle (i.e. the joint configuration) and the orientation angle of the preceding link, and wraps the result to the range $[-\pi, \pi]$ radians. Thus, the angle of the second joint relative to the workspace frame is

$$\theta_2' = \theta_1 + \theta_2 + 2\pi k_2,\ k_2 = \{z \in \{-1, 0, 1\} \mid \theta_1 + \theta_2 + 2\pi z \in [-\pi, \pi]\}$$

Thus, the Cartesian position of the third joint (i.e. the other end of the second link) would just be:

$$(x_2, y_2) = L_2 \cdot (cos(\theta_2'), sin(\theta_2')) + (x_1, y_1)$$

We added the position of the second joint here, since the second link begins at that point. Continuing on in this manner, we see that the Cartesian position of the i-th joint relative to the workspace frame is:

$$\begin{gathered}\forall i \in [1, 4] : (x_i, y_i) = L_i \cdot (cos(\theta_i'), sin(\theta_i')) + (x_{i-1}, y_{i-1}) \\ = L_i \cdot (cos(\theta_i + \theta_{i-1} + 2\pi k_i), sin(\theta_i + \theta_{i-1} + 2\pi k_i)) + (x_{i-1}, y_{i-1}) \\ \text{Initial Conditions: } (x_0, y_0) = (0, 0),\ \theta_0 = 0\end{gathered}$$

In our code, we recorded the angle of the previous link θ_{i-1} as a variable `prev_angle` (initialized to zero), the Cartesian position of the base of a link (i.e. the location of the known joint) (x_{i-1}, y_{i-1}) as a two-element list called `link_base` (initialized to $[0, 0]$ to represent the origin), and the angle of the current link θ'_i relative to the workspace frame as the variable `new_angle`. The original configuration angle θ_i for each link (relative to the preceding link) is obtained from the corresponding element in the input list `given_config`.

We iterated over each link in the robot to calculate the position of the unknown end, and added the result to an initially empty list called `pos_vec`, which is returned by the function (as a `numpy` array) once the loop is complete. The *i*-th element of this list contains the position of the ends of each link (not including the origin), with the final element giving the Cartesian position of the end-effector relative to the workspace frame. The full code implementation looks as follows:

```
def compute_forward_kinematics(self, given_config):
    link_base = [0,0]
    pos_vec = []
    prev_angle = 0
    for l in range(len(given_config)):
        new_angle = self.compute_link_angle(prev_angle,given_config[l])
        x = self.links[l]*np.cos(new_angle) + link_base[0]
        y = self.links[l]*np.sin(new_angle) + link_base[1]
        link_base = [x,y]
        pos_vec.append(link_base)
        prev_angle = new_angle
    return np.array(pos_vec)
```

3. For the `validate_robot` function, we found the simplest way to check if a configuration would contain self-collisions was to build a `LineString` object (from the `shapely` library) from the given link positions, and then used the `is_simple` attribute of the `LineString` class to check if the robot's geometry is simple, i.e. it contains no self-collisions anywhere (except maybe the boundary points, in this case the origin and the end-effector). The `is_simple` attribute evaluates to *True* if the object is indeed simple. The full code implementation looks as follows:

```
def validate_robot(self, robot_positions):
    arm = LineString(robot_positions)
    return arm.is_simple
```

If we wanted to implement this function by hand, all we would need to do is iterate over every pair of links and check if they intersect (since they are line segments, there is a geometric condition for this that can be checked relatively efficiently).

2.3 - Motion Planning

In the previous assignment, you implemented the Rapidly-Exploring Random Tree (RRT) for a point robot in a 2D world. Here, you will start by implementing the same algorithm for the robotic manipulator. In the file `RRTMotionPlanner.py`, implement the following functions:

1. `extend` - With the same instructions as before - $E1$ for extending all the way until the sampled point and $E2$ for extending to the sampled point only by a step-size η).

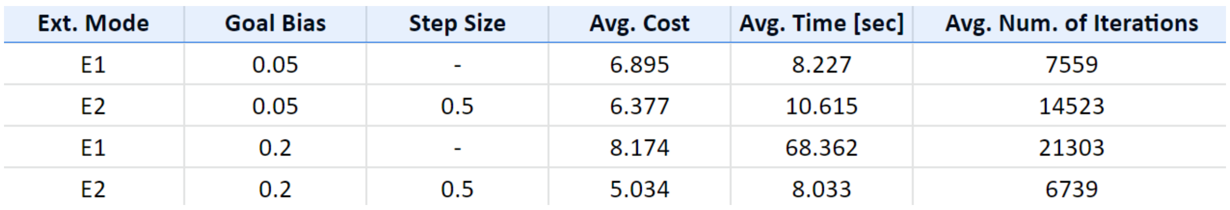
2. `compute_cost` - Compute the cost of the given path, as the sum of the distances of all steps.

3. `plan` - The body of your planning algorithm with goal biasing. Make sure that the output is a `numpy` array of the calculated path in the configuration space (should be a shape of $(N, 4)$ for N configurations, including start and goal configurations).

Report the performance of your algorithm (cost of the path and execution time) for goal biasing of 5% and 20% averaged over 10 executions for each, and attach the visualizations created by `visualize_plan`.

Solution:

The code for the functions we were required to implement here is shown (in a cleaned-down version) in the appendix at the end of this document.



Ext. Mode	Goal Bias	Step Size	Avg. Cost	Avg. Time [sec]	Avg. Num. of Iterations
E1	0.05	-	6.895	8.227	7559
E2	0.05	0.5	6.377	10.615	14523
E1	0.2	-	8.174	68.362	21303
E2	0.2	0.5	5.034	8.033	6739

Figure 1: Table containing the statistical results for the motion planning experiments

In Figure 1, we see that on average the E2 extension mode computes a lower cost path, as expected - the step size is smaller and incremental. On the other hand, we expected that the average computation time would be smaller for E1, as the tree is built outwards faster. However, when the goal bias is 0.2, it is clear that the average computation time in the experiments is higher for the E1 extension mode. When retrieving statistics for the E1 mode, we found there was an anomaly of a single trial which took over 300 seconds, drastically increasing our average computation time.

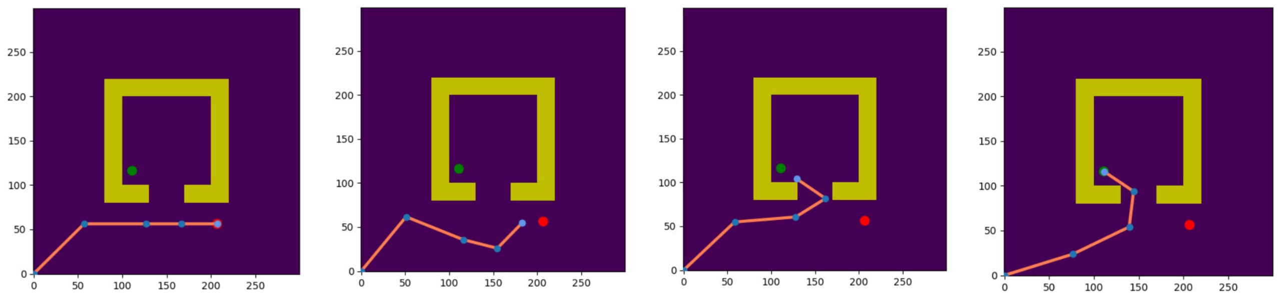


Figure 2: Snapshots from a GIF showing a motion planning run

2.4 - Inspection Planning

In inspection planning, we are require to inspect a set of points of interest (POI) and not reach a predefined goal. In this exercise you are required to adapt your RRT implementation to handle the inspection task (this is slightly easier than the IRIS algorithm taught in class). A vertex v_i in the RRT tree, will now also include the subset of POI that were seen so far when following the path from the root of the RRT tree to the configuration associated with the vertex and set $I_i = \{p_1, \cdots, p_i\}$ to be this subset of POI. Namely, if in the (vanilla) motion-planning problem we defined a vertex as $v_i := [\theta_{i,1}, \theta_{i,2}, \theta_{i,3}, \theta_{i,4}]$, it will now be $v_i := [\theta_{i,1}, \theta_{i,2}, \theta_{i,3}, \theta_{i,4}, I_i]$. Moreover, if the edge (v_{i-1}, v_i) exists, then the set of POI inspected must be $I_i = I_{i-1} \cup S(v_i)$, where $S(v_i)$ is the set of inspection points that the robot can see at step vertex v_i.

Next, we define the coverage of a vertex v_i as $\frac{|I_i|}{|I_{max}|}$, where I_{max} is the set of all POI. Namely, a vertex with coverage of 1.0 corresponds to a path along which all POI have been inspected. To this end, the stopping condition of your planner will be reaching not a goal vertex but a vertex with a desired coverage (use the argument `--coverage` as shown in Sec. 2.1) and return the path to it. Implement the following in `RRTInspectionPlanning.py`:

1. `extend` and `compute_cost` - These can stay the same as in the motion-planning task.
2. `compute_union_of_points` (in `MapEnvironment.py`) - Compute the union of two sets of inspection points.
3. `plan` - The body of your planning algorithm. Make sure that the output is a numpy array of the calculated path in the configuration space (should be a shape of $(N, 4)$ for N configurations, including start and end configurations).

Report your performance (cost of the path and execution time) for coverage of 0.5 and 0.75, averaged over 10 executions for each, and attach the visualizations created by `visualize_plan`.

Discuss the changes you were required to do in order to compute inspection planning. Notice that for higher coverage the search may take a few minutes, so think about how you can improve your search!

Hint: In motion planning we biased the sampling towards a goal vertex. If there is no goal, how can we bias the sampling to improve coverage?

Solution:

For the inspection planning task, we recognized that the lack of concrete goal points meant that we'd need to create a whole new scheme for the biasing portion of the algorithm. The algorithm that we produced to this end is outlined below in Algorithm 1.

Algorithm 1 MODIFIED RRT FOR INSPECTION PLANNING (RRT-IP)

```
function RRT-IP($v_{start}, coverage, prob = 0.05, \alpha = 10^{-3}, \eta = 0.5$)
    $p_0 \leftarrow prob,\ v_{best} \leftarrow v_{start},\ N \leftarrow 0,\ rewire\_successful \leftarrow False$
    $Tree.init(v_{start})$
    $all\_seen\_pts \leftarrow \{p : v_{start}$ for $p$ in $v_{start}.inspected\_points\}$
    $max\_cov \leftarrow Compute\_Coverage(v_{best}.inspected\_points)$
    while $max\_cov < coverage$ do
        $N \leftarrow N + 1$
        $p \leftarrow Unif[0, 1]$
        if $p \geq prob$ then                                                    ▷ exploration
            $C_{new} \leftarrow Random\_Sample\_Config()$
            $prob \leftarrow Min(p_0 + 1 - e^{-\alpha \cdot N}, 1 - p_0)$        ▷ prob. of exploration decays over time
        else                                                                     ▷ exploitation
            if we go a certain num. of iters. without improving the max. coverage then
                $rewire\_successful \leftarrow$ REWIRE($Tree, all\_seen\_pts, v_{best}$)
            if $rewire\_successful == False$ then                                ▷ rewire failed or not attempted
                $C_{new} \leftarrow$ IMPROVED_SAMPLE_CONFIG($v_{best}$)
        if $rewire\_successful == False$ and $Valid\_Config(C_{new})$ then       ▷ no rewire + check config. validity
            $v_{near} \leftarrow Nearest\_Neighbor(Tree, C_{new})$
            if ext. mode E2 is enabled then
                $C_{new} \leftarrow Extend(C_{new}, v_{near}.config, \eta)$
            if $Collision\_Free(v_{near}.config, C_{new})$ then
                $v_{new} = Tree.add\_vertex(C_{new})$
                $Tree.add\_edge(v_{near}, v_{new})$
                for $p$ in $v_{new}.inspected\_points$ do
                    if $p$ not in $all\_seen\_pts.keys()$ then                   ▷ if the new sample sees an unseen point
                        $all\_seen\_pts[p] \leftarrow v_{new}$                   ▷ then record this in the $all\_seen\_pts$ dictionary
        $v_{best} \leftarrow Get\_Best\_Vertex(Tree)$                            ▷ best = highest coverage (not lowest cost)
        $max\_cov \leftarrow Compute\_Coverage(v_{best}.inspected\_points)$
    return the path from $v_{start}$ to $v_{best}$ in $Tree$
```

In this algorithm, we will build a tree where each vertex v_i records the configuration it corresponds to $[\theta_{i,1}, \theta_{i,2}, \theta_{i,3}, \theta_{i,4}]$, the inspected points seen on the path to this vertex from the root I_i, and the cost to reach this vertex from the root (i.e. the sum of the distances between every pair of consecutive configurations along the path). Given that the coverage of a vertex v_i is defined to be $Cov_i = \frac{|I_i|}{|I_{max}|}$, we will also define the maximum coverage of the tree $\mathcal{T}$ to be

$$Cov_{max} = \max_{v_i \in \mathcal{T}} \frac{|I_i|}{|I_{max}|}$$

Thus, our algorithm will build a tree starting from the root vertex (which corresponds to the given initial configuration of the robot arm) until its maximum coverage is better than the requested coverage value. Once this threshold is exceeded, the algorithm will return the path from the root vertex C_{start} to the vertex which has the maximal coverage.

In contrast to the classic motion planning problem, the lack of concrete goal points here led us to inventing the following scheme for building the tree. We begin with some provided probability value (herein the "bias", denoted as *prob*) from the user (the initial bias value is recorded as p_0), with the default being 5%, and we randomly sample a value from the uniform distribution (in [0,1]). If our sampled value is greater than the bias, then we will conduct an "exploration"-like scheme, where we will randomly sample a new configuration $[\theta_1, \theta_2, \theta_3, \theta_4]$, where each value θ_j is sampled from the range $[\pi, \pi]$. We then update the bias value using a decay scheme, where the new bias will be determined as follows:

$$prob = min\{p_0 + 1 - e^{-\alpha \cdot N}, 1 - p_0\}$$

where α is a decay parameter (set to 10^{-3} as default, empirically this value exhibited the best performance in terms of minimal runtime) and N is the number of iterations conducted thus far. When the number of iterations $N > 3000$ (at this point, the maximal coverage usually exceeds 0.35), we see that the bias will reach its maximum value $(1 - p_0)$ - this maximum value is less than 1 in order to allow for some exploration even in the late stages of a run. Then, if this sampled configuration is valid, we will find the nearest neighbor (i.e. the vertex in the tree whose configuration is closest to the sampled configuration) to the sampled configuration and check if a valid edge can be constructed between them. If the edge can indeed be constructed, then we add a new vertex to the tree corresponding to this configuration and we add the edge between this new vertex and the neighbor vertex to the tree.

Lastly, we will record all inspection points seen by our tree (and the first vertex to see them) using a dictionary called *all_seen_pts*. We initialize it to include all the points seen by the start vertex (in our case there are no such points, and so the dictionary is initially empty), and each time we add a new vertex to the tree we will check if any of the inspected points seen along the path to this vertex have been seen before elsewhere in the tree. For newly-seen points, we add them as a key in the dictionary whose value is the index of the new vertex (this value will remain constant for the remainder of the run).

Now, back to the biasing scheme, if our sampled value is instead lower than the current bias value, then we will conduct an "exploitation"-like scheme, where we will repeatedly sample new configurations until we either generate a configuration which sees points unseen by our current best vertex or until we've unsuccessfully generated 150 such configurations (this way we don't spend too long looking for such points, in case they are too difficult too find). This process can be seen in Algorithm 2.

Once a new configuration C_{new} is obtained, we will again check that it is valid, identify the nearest neighbor, and check if a valid edge can be constructed between them. We proceed to add the edge and vertex to the tree as well as update the *all_seen_pts* dictionary exactly as before in the exploration scheme. This method allows us to bias our sampling towards configurations that could potentially increase the coverage of our current best path - this is the major difference from goal biasing in the regular motion planning task. Empirically, this method has shown effective for reaching (on average) a maximal coverage of at least 0.35 by the 1000-th iteration of the algorithm.

Algorithm 2 Improved Configuration Sampling Scheme

```
function Improved_Sample_Config($v_{best}$)
    $max\_cov \leftarrow Compute\_Coverage(v_{best}.inspected\_points)$
    $new\_cov \leftarrow -1$
    $count \leftarrow 0$
    while $count < 150$ and $new\_cov < max\_cov$ do
        $count \leftarrow count + 1$
        $C_{new} \leftarrow Random\_Sample\_Config()$
        $new\_pts \leftarrow Compute\_Union\_of\_Pts(Get\_Inspected\_Points(C_{new}), v_{best}.inspected\_points)$
        $new\_cov \leftarrow Compute\_Coverage(new\_pts)$
    return $C_{new}$
```

Once the tree reaches a maximal coverage of above 0.35, though, we noticed that the algorithm described until now begins to struggle with quickly finding new configurations that improve the tree's maximal coverage. This phenomenon is caused by the intrinsic nature of the RRT algorithm: new vertices are always added to the nearest paths to them in the tree. Let's say we have one set of inspection points on the left side of the workspace, and another set on the right side, and let's say that the tree has one path nearest to each set. Thus, if we want to improve the coverage of the path on the right (which has seen the points on the right but not on the left), we'll need to sample new configurations near enough to the inspection points on the left to see them but also near enough to the right path such that it will be the nearer path to these samples (rather than the left path). The likelihood of randomly sampling such configurations is relatively low, and so we will instead just continue to grow each of the paths without significant improvements to the coverages of either one.

This issue causes the algorithm to have quite large runtimes for coverages greater than about 0.35, and so we'll need to come up with some way to push it past this point and allow for quicker runtimes. To do so, we will implement an idea that resembles the rewiring method from the RRT* algorithm: if an inspected point is not seen along the current best path in the tree, but is known to be seen by some other configuration elsewhere in the tree, then we will try to "rewire" the tree such that the current best vertex will now be the parent of that vertex corresponding to that other configuration. An overview of this approach is shown below in Algorithm 3.

Algorithm 3 Rewiring Scheme

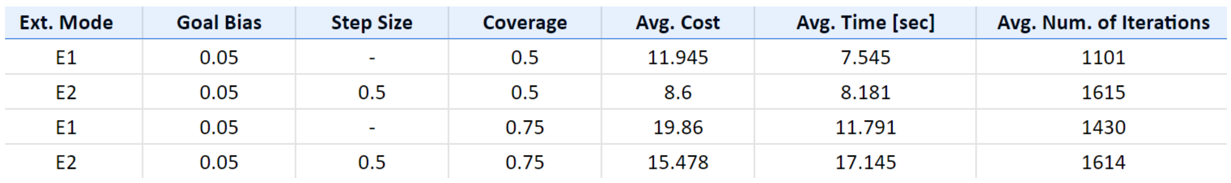
```
function Rewire($Tree, all\_seen\_pts, v_{best}$)
    $missing\_pts \leftarrow []$
    for $p$ in $all\_seen\_pts.keys()$ do
        if $p$ not in $v_{best}.inspected\_points$ then
            $v_p \leftarrow all\_seen\_pts[p]$
            $cost \leftarrow Compute\_Distance(v_p.config, v_{best}.config)$
            $missing\_pts.insert(v_p)$                        ▷ queue in order of increasing cost
    if $len(missing\_pts) == 0$ then
        return $False$
    $v_{new} \leftarrow None$
    $rewire\_successful \leftarrow False$
    while $v_{new} == None$ and $len(missing\_pts) > 0$ do
        $v_{curr} \leftarrow missing\_pts.pop()$
        if $Collision\_Free(v_{best}.config, v_{curr}.config)$ then
            $Tree.edges[v_{curr}.idx] \leftarrow v_{best}.idx$          ▷ this is the rewiring
            update $v_{curr}$ and all of its descendants
            $v_{new} \leftarrow v_{curr}$
            $rewire\_successful \leftarrow True$
    return $rewire\_successful$
```

We see that this algorithm begins by identifying all points seen somewhere in the tree but not yet seen along the path leading to the best vertex, and using the *all_seen_pts* dictionary to record the vertex corresponding to each such missing point. These vertices are then stored in a queue called *missing_pts*, which is sorted according to the distance between the vertex (which sees a missing point) and the best vertex - the vertex with the lowest such distance will be first in the queue. We do not yet check whether there is a valid edge between these vertices and the best vertex - we will postpone this step to reduce our computational overhead (i.e. we'll do a form of lazy collision-checking).

Once we have our *missing_pts* queue (and assuming it is not empty), we will iterate through the queue and check whether each vertex in the queue can be connected to the best vertex in the tree with a collision-less edge - here we do the actual collision-checking, and in many cases we will get lucky and only need to do the check for the first few vertices in the queue (as opposed to doing so for all vertices in the queue - this is thanks to the fact that the first vertices are closer to the best vertex than the later ones, and thus a collision-free edge is more likely to exist given the relatively simple geometry of our workspace). Once the first vertex in the queue with a collision-free edge to the best vertex is identified, we will reassign its parent vertex to now be the best vertex (i.e. we essentially delete the edge between this vertex and its parent, and create a new edge between this edge and the best vertex), and recalculate its properties (the cumulative cost of and the union of all inspection points seen on the new path to this vertex).

The coverage of this vertex will then be *at least* the maximal coverage of the tree (since adding new edges to the best vertex can't decrease the path's coverage). We then update the same properties (cost, inspected points) for all of the descendants of this vertex using a recursive function (called `propagate` in our code), and for each such descendant vertex we will compare its coverage to the maximal coverage of the tree thus far - if it is greater than or equal to the max. coverage, we will set the best vertex to now be this descendant vertex. Doing so, we will ensure that the best vertex in the tree after rewiring is always a leaf node - this prevents possible cycles in the tree, which we initially dealt with and saw to be problematic.

Thus, we see that each successful rewiring operation will result in a tree with an increased maximal coverage, and this method has proved greatly effective at causing the algorithm to terminate much more quickly for coverages above 0.35 (on average, it can terminate in less than 20 seconds for coverages as high as 0.75). Thus, during each run of the entire algorithm, we will choose to attempt a rewiring every time we enter "exploitation" mode once the maximal tree coverage is above 0.35, and if the rewiring does not succeed (i.e. none of the vertices that have seen missing points are reachable from the current best vertex) then we will attempt the improved configuration sampling scheme described previously (Algorithm 2). If the rewiring does succeed, then we just record the new maximal coverage of the tree and proceed on to the next iteration (if the desired coverage is not yet reached). When the maximal coverage is below 0.35 during a run, we will still allow our planner to attempt rewires during "exploitation" mode, but given the nascent nature of the tree in such cases it will not be effective to attempt a rewiring every single time. Instead, we will only attempt rewiring at maximum coverages below 0.35 if we have gone 1000 iterations without improving the maximum coverage value of the tree. This way, we give our planner sufficient opportunity to explore and build the tree quickly (even during exploitation), while giving it a boost when the results are stagnant for too long.



Ext. Mode	Goal Bias	Step Size	Coverage	Avg. Cost	Avg. Time [sec]	Avg. Num. of Iterations
E1	0.05	-	0.5	11.945	7.545	1101
E2	0.05	0.5	0.5	8.6	8.181	1615
E1	0.05	-	0.75	19.86	11.791	1430
E2	0.05	0.5	0.75	15.478	17.145	1614

Figure 3: Table containing the statistical results for the inspection planning experiments

The algorithm described thus far corresponds to the E1 extension mode. In the case of the E2 extension mode, the only change will be to use the extension feature that we've seen before in RRT to redefine the configurations sampled during the exploration and non-rewiring exploitation modes. We set the step size for these extensions to be $\eta = 0.5$ (we empirically observed this to be a favorable value), and we would expect that this extension mode will cause the tree to grow more slowly (increasing the total runtime needed) while returning better paths (decreasing the total cost). A statistical comparison of the algorithm under the E1 and E2 extension modes - for requested coverages of 0.5 and 0.75 - over 10 trials each can be seen in the table in Figure 3.

As expected, the planner returned lower-cost paths (on average) for the E2 mode compared to the E1 mode for each requested coverage value, as a result of the ”finer” nature of the trees generated under the E2 mode. The tradeoff for this was that the E2 mode required more runtime (on average) than the E1 mode to reach a valid plan, and this is due to the fact that the tree under the E2 mode tends to grow more slowly (at least, in the initial stages when the maximal coverage of the tree is less than 0.35 - beyond this point both modes will attempt to rewire every iteration).

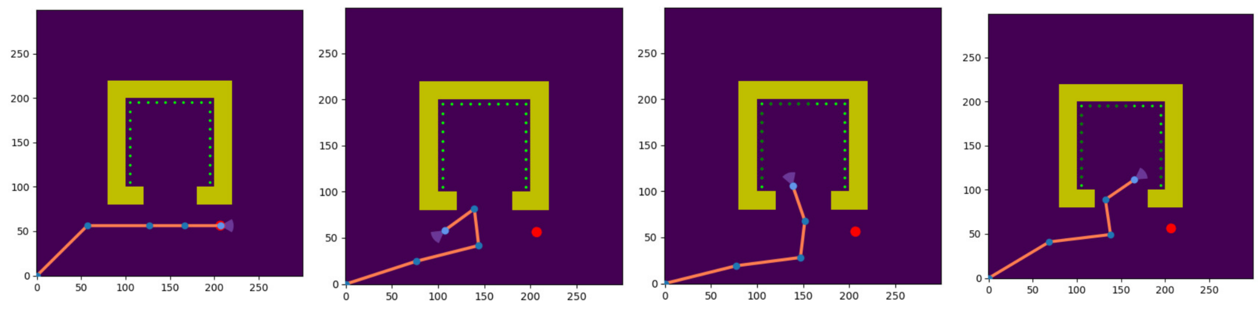


Figure 4: Snapshots from a GIF showing an inspection planning run

Code Appendix

2.3 - Motion Planning Code

The constructor of the RRTMotionPlanner class:

```python
class RRTMotionPlanner(object):
    def __init__(self, planning_env, ext_mode, goal_prob):
        # set environment and search tree
        self.planning_env = planning_env
        self.tree = RRTTree(self.planning_env)

        # set search params
        self.ext_mode = ext_mode
        self.goal_prob = goal_prob
        self.step_size = planning_env.step_size # step size for extensions (custom)
```

1. extend:

```python
    def extend(self, near_config, rand_config):
        goal = False
        goal_config = self.planning_env.goal
        if np.allclose(rand_config, goal_config):
            goal = True

        vec = np.subtract(rand_config, near_config)
        vec_mag = np.linalg.norm(vec,2)
        unit_vec = vec / vec_mag

        # Projection of the step size in the direction of the neighbor
        new_vec = self.step_size * unit_vec
        new_config = near_config + new_vec # New sample point

        # Check if this overshoots the goal, if yes return the goal
        goal_added = False
        if goal and vec_mag < self.step_size:
            new_config = goal_config
            goal_added = True # Tells the motion planner to terminate

        return new_config, goal_added
```

2. compute_cost:

```python
    def compute_cost(self, plan):
        return self.tree.get_vertex_for_config(plan[-1]).cost
```

3. plan:

```python
    def plan(self):
        # Initialize the tree
        env = self.planning_env
        self.tree.add_vertex(env.start)

        plan = []; goal_added = False
        while not goal_added:
            goal = False

            # Sampling step
            p = np.random.uniform() # goal biasing
            if p < self.goal_prob:
                config = env.goal
                goal = True
            else:
                config = np.random.uniform(low=-np.pi,high=np.pi,size=(4,))

            # Verify that the sample is in free space
            if not env.config_validity_checker(config):
                continue

            # Get nearest vertex to the sample
            nearest_vert = self.tree.get_nearest_config(config)
            nearest_vert_idx = nearest_vert[0]

            # Partial extensions, if enabled
            if self.ext_mode == 'E2':
                config, goal_added = self.extend(nearest_vert[1], config)
                if not env.config_validity_checker(config):
                    continue

            # Check obstacle-collision for potential edge
            if env.edge_validity_checker(config, nearest_vert[1]):
                config_idx = self.tree.add_vertex(config, nearest_vert)
                cost = env.robot.compute_distance(config, nearest_vert[1])
                self.tree.add_edge(nearest_vert_idx, config_idx, cost)
                if goal and self.ext_mode == 'E1':
                    goal_added = True
            else:
                goal_added = False

        # Record the plan
        plan.append(config)
        child_idx = config_idx
        parent_config = nearest_vert[1]
        while self.tree.edges[child_idx]:
            plan.append(parent_config)
            child_idx = self.tree.get_idx_for_config(parent_config)
            parent_idx = self.tree.edges[child_idx]
            parent_config = self.tree.vertices[parent_idx].config
        plan.append(parent_config)
        plan = plan[::-1]

        return np.array(plan)
```

2.4 - Inspection Planning Code

The constructor of the RRTInspectionPlanner class:

```
class RRTInspectionPlanner(object):
    def __init__(self, planning_env, ext_mode, goal_prob, coverage):
        # set environment and search tree
        self.planning_env = planning_env
        self.tree = RRTTree(self.planning_env, task="ip")

        # set search params
        self.ext_mode=ext_mode; self.goal_prob=goal_prob; self.coverage=coverage

        # set custom parameters
        self.step_size = planning_env.step_size # for extensions
        self.decay = 10**-3 # for bias parameter decay

        # record best vertex info
        self.best_idx = None; self.best_pts = None; self.best_config = None
```

1. extend:

```
def extend(self, near_config, rand_config):
    vec = np.subtract(rand_config, near_config)
    vec_mag = np.linalg.norm(vec,2)
    unit_vec = vec / vec_mag

    new_vec = self.step_size * unit_vec
    new_config = near_config + new_vec

    if self.planning_env.config_validity_checker(new_config):
        return new_config
    else:
        return None
```

2. compute_cost:

```
def compute_cost(self, plan):
    return self.tree.get_vertex_for_config(plan[-1]).cost
```

3. compute_union_of_points:

```
def compute_union_of_points(self, points1, points2):
    # check if either list is empty (can't concatenate empty and non-empty list)
    if len(points1) == 0:
        return points2
    elif len(points2) == 0:
        return points1
    # return unique points in the concatenation of the two sets (i.e. the union)
    return np.unique(np.concatenate([points1, points2]), axis=0)
```

4. plan:

```
def plan(self):
    start_time = time.time(); env = self.planning_env
    plan = [] # initialize an empty plan
    p_0 = self.goal_prob # original bias value

    start_pts = list(env.get_inspected_points(env.start))
    start_idx = self.tree.add_vertex(env.start, inspected_points=start_pts)
    # we want to keep track of all points seen in the tree so far
    self.points_inspected = {tuple(p):start_idx for p in start_pts}

    num_iter = 0; reset_counter = 0; thresh = 1000
    while self.tree.max_coverage < self.coverage:
        rewire_successful = False; num_iter += 1
        current_coverage = self.tree.max_coverage
        self.update_best() # identify best vertex in the tree

        # bias towards configurations with better coverage opportunities
        p = np.random.uniform()
        if p < self.goal_prob: ## Exploitation
            # if no max. coverage improvements after too long, attempt rewire
            if self.tree.max_coverage > 0.35:
                thresh = 0
            if reset_counter > thresh:
                reset_counter = 0
                rewire_successful = self.rewire(stats_mode=stats_mode)

            # sample configs. until point not seen by best path is seen
            new_cov = -1; count = 0
            if not rewire_successful:
```

```python
                    while (count < 150) and (new_cov < self.tree.max_coverage):
                        config = np.random.uniform(low=-np.pi, high=np.pi, size=(4,))
                        count += 1
                        # ensure that the sampled configuration is valid
                        if not env.config_validity_checker(config): continue
                        pts = env.get_inspected_points(config)
                        new_pts = env.compute_union_of_points(pts, self.best_pts)
                        new_cov = env.compute_coverage(inspected_points=new_pts)

            else: ## Exploration
                config = np.random.uniform(low=-np.pi, high=np.pi, size=(4,))
                # add a decay that will decrease exploration over time
                self.goal_prob = min(p_0+1-np.exp(-self.decay*num_iter), 1-p_0)
                if not env.config_validity_checker(config): continue

            if not rewire_successful: # no rewiring was successfully done
                nearest_vert = self.tree.get_nearest_config(config)
                nearest_vert_idx = nearest_vert[0]

                if self.ext_mode == 'E2': # Partial extensions, if enabled
                    config = self.extend(nearest_vert[1], config)
                    if config is None: continue

                if env.edge_validity_checker(config, nearest_vert[1]):
                    pts_prev = self.tree.vertices[nearest_vert_idx].inspected_points
                    pts_new = env.get_inspected_points(config)
                    # take union of inspected pts. of parent and child vertex
                    pts_newer = env.compute_union_of_points(pts_prev, pts_new)
                    v_new_idx=self.tree.add_vertex(config,inspected_points=pts_newer)

                    # check if new vertex's inspected pts. were already seen in tree
                    pts_to_check = [key for key in self.points_inspected]
                    union = env.compute_union_of_points(pts_to_check, pts_new)
                    if len(union) > len(self.points_inspected):
                        for p in union:
                            if tuple(p) not in pts_to_check:
                                self.points_inspected[tuple(p)] = v_new_idx
                    # calculate cost of new vertex and add the edge to the tree
                    cost = env.robot.compute_distance(config, nearest_vert[1])
                    self.tree.add_edge(nearest_vert_idx, v_new_idx, cost)

            self.update_best()
            if current_coverage < self.tree.max_coverage:
                reset_counter = 0
            else: # checking for stagnations in performance
                reset_counter += 1

        plan.append(self.best_config)
        child_idx = self.best_idx
        parent_idx = self.tree.edges[child_idx]
        parent_config = self.tree.vertices[parent_idx].config

        while self.tree.edges[child_idx]:
            plan.append(parent_config)
            child_idx = self.tree.get_idx_for_config(parent_config)
            parent_idx = self.tree.edges[child_idx] # new parent
            parent_config = self.tree.vertices[parent_idx].config
        plan.append(parent_config)
        plan = plan[::-1]

        return np.array(plan)
```

Auxiliary functions for the `plan` method:

```
    def update_best(self):
        # when a new max. coverage vertex is found, we should update all of the
        # attributes related to the best path
        self.best_idx = self.tree.max_coverage_id
        best_vert = self.tree.vertices[self.best_idx]
        self.best_pts = [tuple(b) for b in best_vert.inspected_points]
        self.best_config = best_vert.config
        return

    def rewire(self):
        ## Step 1: Check which points have been seen in other paths but not in the
        # current best path (if there are any)
        missing_pts = []
        for p in self.points_inspected:
            if p not in self.best_pts:
                missing_pts.append(p)
        if len(missing_pts) == 0:
            return False

        ## Step 2: For each missing point, record the vertex that sees it in a new
        # list and sort this list according to the minimum distance between each
        # vertex and the best vertex in the tree
        v_curr = None; dists = []; env = self.planning_env
        for m in missing_pts:
            v_p_idx = self.points_inspected[m]
            v_p = self.tree.vertices[v_p_idx]
            dist = env.robot.compute_distance(v_p.config, self.best_config)
            dists.append({'Vertex':v_p_idx, 'Dist':dist})
        dists.sort(key=lambda x:x.get('Dist'))
        v_curr_idx = dists[0]['Vertex']
        v_curr = self.tree.vertices[v_curr_idx]

        ## Step 3: If a valid path was found to a missing point's config, then rewire
        # the tree such that the current best path leads to that config (i.e. the
        # vertex corresponding to that config will be updated)
        if v_curr is not None:
            i = 0
            while not env.edge_validity_checker(v_curr.config, self.best_config):
                i += 1
                if i > len(dists)-1:
                    return False
                # get index of vertex in the tree that corresponds to that config.
                v_curr_idx = dists[i]['Vertex']
                v_curr = self.tree.vertices[v_curr_idx]

            # cost = env.robot.compute_distance(v_curr.config, self.best_config)
            # update the cost of the path to the vertex
            cost = dists[i]['Dist']
            old_cost = v_curr.cost # get the original cost of the rewired vertex
            v_curr.cost = self.tree.vertices[self.best_idx].cost + cost

            # check pts seen by rewired vert, take their union with best vert's pts
            pts_new = env.get_inspected_points(v_curr.config)
            pts_newer = env.compute_union_of_points(self.best_pts, pts_new)
            v_curr.inspected_points = pts_newer # update inspected pts of vert
            self.tree.edges[v_curr_idx] = self.best_idx # update edge leading to vert

            # update the max. coverage (this should always increase it)
            self.tree.max_coverage = env.compute_coverage(inspected_points=pts_newer)
            self.tree.max_coverage_id = v_curr_idx

            # update costs and inspected pts of all descendents of the rewired vertex
            self.propagate(v_curr_idx, old_cost)
            self.update_best()
            return True # rewire successful
        return False

    def child_update(self, child, parent, cost_0):
        # record the configuration of the child node
        v_child = self.tree.vertices[child]
        config = v_child.config
        cost_1 = v_child.cost # original child cost

        # update the cost of the child node
        v_parent = self.tree.vertices[parent]
        v_child.cost = (cost_1 - cost_0) + v_parent.cost

        # record the points seen directly by this configuration
        env = self.planning_env
        orig_pts = env.get_inspected_points(config)

        # update the inspected points seen along path to the child
        parent_pts = v_parent.inspected_points
        v_child.inspected_points = env.compute_union_of_points(orig_pts, parent_pts)

        # check if coverage improves / stays same (ensures best vertex is leaf node)
        new_coverage=env.compute_coverage(inspected_points=v_child.inspected_points)
        if new_coverage >= self.tree.max_coverage:
            self.tree.max_coverage = new_coverage
            self.tree.max_coverage_id = child
        return cost_1

    def propagate(self, parent, cost_0):
        # propagates the costs + inspected pts of the descendants of rewired vertex
        if parent not in self.tree.edges.values():
            return
        children = []
        for k,v in self.tree.edges.items():
            if v == parent:
                children.append(k)
        for c in children:
            cost_1 = self.child_update(c, parent, cost_0)
            self.propagate(c, cost_1)
        return
```