Robotic Science and Systems - Exercise 2

Kevin Robb

C0

Given a 2D workspace with two square robots that move on fixed tracks, we can create an equivalent configuration space by parametrizing distance along each track as a configuration parameter. While the workspace has axes $x \in \{ 1, .., 12 \}$ and $y \in \{ 1, .., 8 \}$, each track has a length of 13 units. We can represent the location of Robot A along track A with a value $q_A \in \{ 0, .., 13 \}$ and the position of Robot B along track B as $q_B \in \{ 0, .., 13 \}$. We treat the leftmost position as being $q_i = 0$.

Under this definition, we can visualize our configuration space, and mark allowable and disallowable configurations (based on points where the robots would be in collision. The configuration represented in the figure can be written as $\begin{bmatrix} q_A & q_B \end{bmatrix}^T = \begin{bmatrix} 5.5 & 3.5 \end{bmatrix}^T$. A discretized sketch of the configuration space follows.

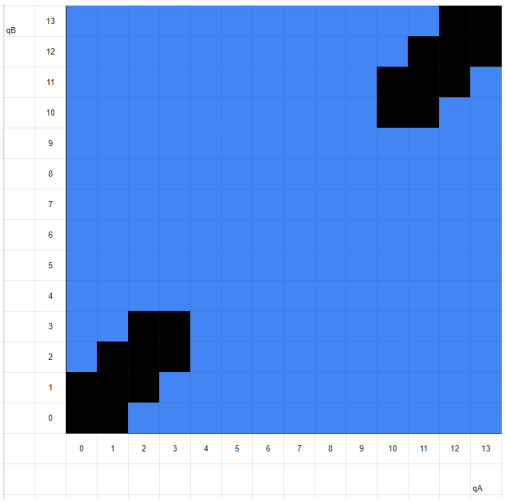
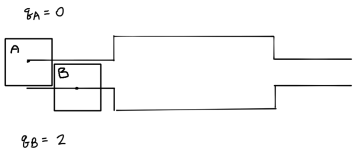
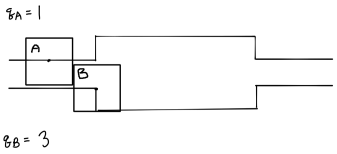


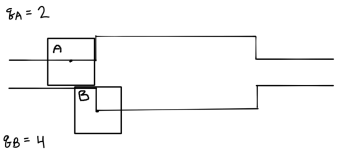
Figure 1: Discretized sketch of the C0 configuration space defined above. Black cells are configurations in collision, while blue cells are free. We can see the only regions in collision are those in which both squares would be near the same side of the workspace. We can also see this is symmetric along both diagonals corresponding to the horizontal and vertical symmetry of the workspace.



(a) A and B can smoothly move along this segment together, preserving a distance of 2 units between them.



(b) A must wait for B to move fully from $q_B = 3$ to $q_B = 4$ before increasing q_A higher than 1.



(c) Now that B has moved to $q_B \geq 4$, A is free to move anywhere (only considering left half of workspace).

Figure 2: Visual explanation of configuration-space obstacles. These can be mirrored vertically and horizontally to explain all occluded regions in Figure 1.

C1

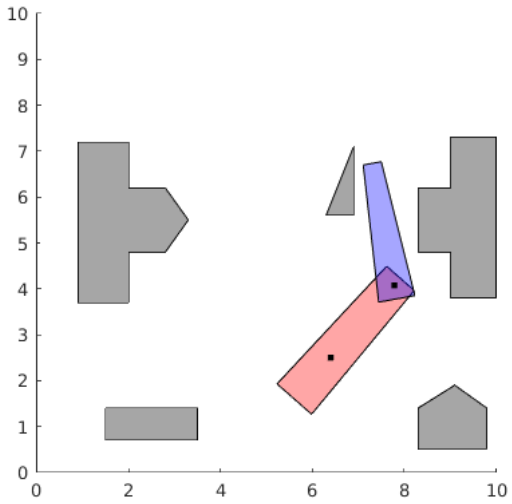
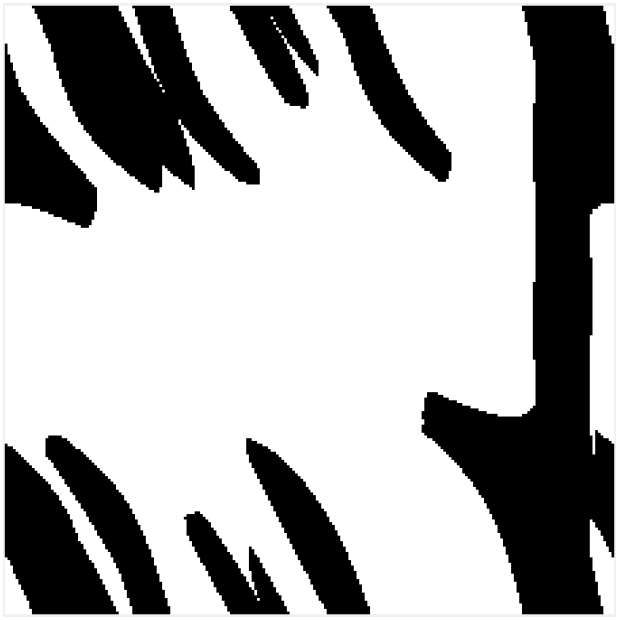


Figure 3: The arm in its starting configuration.

C2

(a) Configuration space with the default resolution `num_samples = 100`.



(b) Configuration space with the resolution increased in `ex2_cspace` to `num_samples = 225`. This is needed for C7.

C3

The distance transform figure looks different from the assignment pdf because I accounted for paths being able to wrap from top $\leftrightarrow$ bottom and left $\leftrightarrow$ right due to the toroidal nature of this configuration space.

(a) Distance transform with the default resolution `num_samples = 100`.

(b) Distance transform with the resolution increased in `ex2_cspace` to `num_samples = 225`.

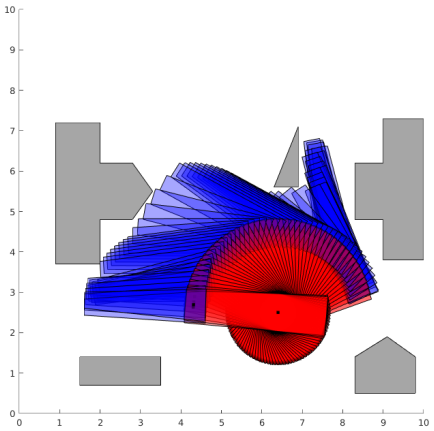
C4

When deciding on a path to follow, my implementation breaks ties by prioritizing movement in both axes as $-1 \to 0 \to 1$, so moving down and left takes precedence.

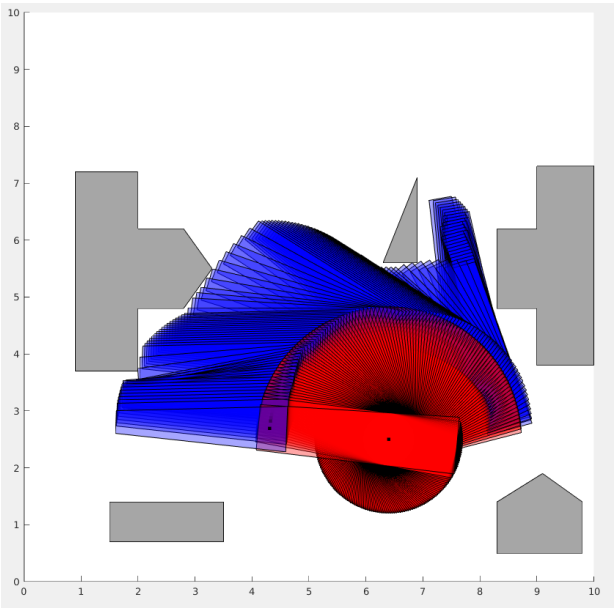
(a) Greedy path with the default resolution `num_samples = 100`.

(b) Greedy path with the resolution increased in `ex2_cspace` to `num_samples = 225`.

C5



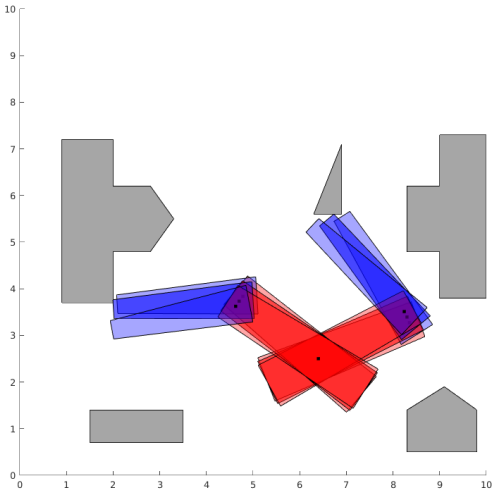
(a) Arm motion with the default resolution `num_samples = 225`.



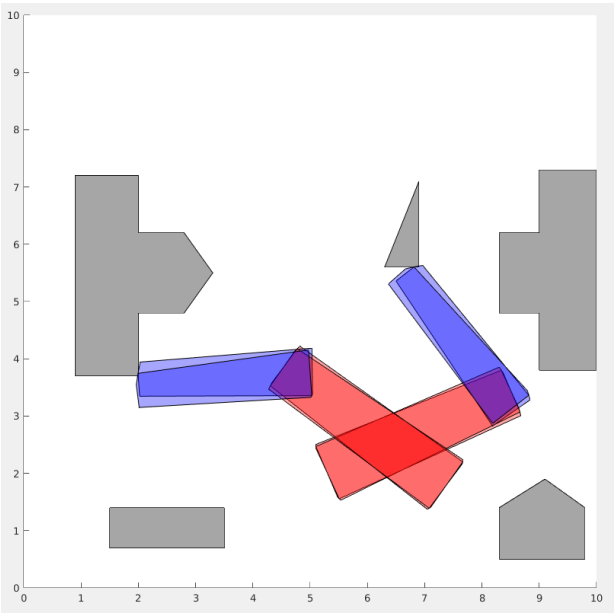
(b) Arm motion with the resolution increased in `ex2_cspace` to `num_samples = 225`.

C6

In my implementation, there are 4 collisions, happening in two groups of two.



(a) Collisions with the default resolution `num_samples = 100`.



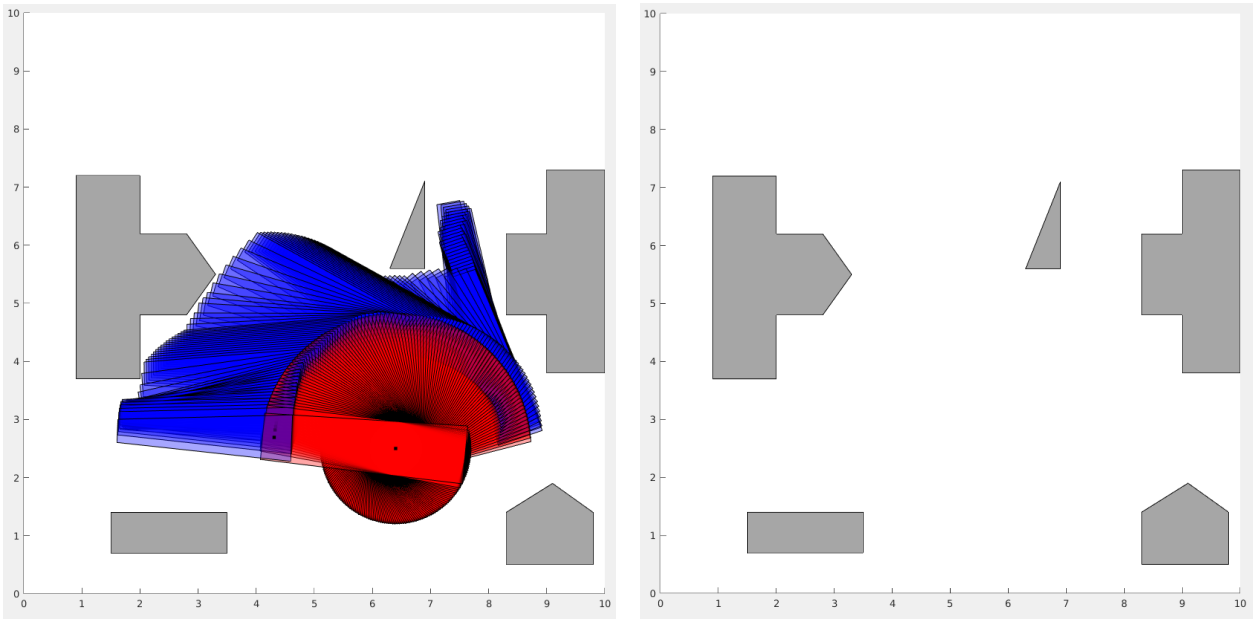
(b) Collisions with the resolution increased in `ex2_cspace` to `num_samples = 225`.

C7

There is so little leeway around the starting position in this configuration space that padding obstacles even by a single grid cell makes this region entirely occluded, meaning a path to or from this point is impossible. Since my path planning does not allow movement to an occupied cell (even if the current cell is occluded) it is not possible to create a path to or from this point.

Figure 9: No path exists between the specified points when any padding is introduced, with the default resolution of 100. A single red dot at the problematic configuration, as well as some at the SW and NE corners, are visible on this cspace plot.

By manually increasing the resolution from 100 to 225 when creating the cspace in C2, there will be enough granularity to pad obstacles and find a path with no collisions.



(a) Path after padding the high-res cspace.

(b) No collisions in this new path.

M0

The check_collision function first computes the position of the ends of each link using the robot's transformations. It then creates an array of ticks between 0 and 1 based on the desired resolution. Then, using the `repmat` function, the position vectors are tiled horizontally to end with one large matrix `points` whose columns are a series of interpolated vectors starting with $\begin{bmatrix}0 & 0 & 0\end{bmatrix}^T$, going through the point at the joint between the two links, and ending with the point at the end of the second link.

Using this, the function then checks all spheres for collision with the points. It does this by computing the distance from each point along the arm to each sphere, and marks a collision if this distance is shorter than the combined radii of the arm and the sphere.

Potential issues with this implementation are as follows. By relying on a specified resolution, and checking planar sections of the arm's cylindrical links, rather than truly checking the 3D volumes for intersections, the possibility is left open that there may indeed be a slight collision which occurs between these discretized planes. This becomes less and less likely as the resolution tends higher and higher. There is also the issue of runtime as a trade-off with resolution; the `check_edge` function interpolates between configurations, so with a high enough resolution to mitigate missed collisions between points or configurations, the number of computations happening will scale as $O(n^2)$, where n is the resolution.

The check_edge function similarly creates an interpolated array of vectors in configuration space between two specified configurations. It then calls the check_collision function on each configuration in the array. The result is that a straight-line path in configuration space is checked for collisions with the arm and all obstacles along the path.

M1

My code successfully generates the set of samples in bounds with the following single line:

```
qs = rand(num_samples, 4) .* (q_max - q_min) + q_min;
```

This ensures all configurations will be doable by the system, although it does not check for collisions. Because of this, in M2 when collision-checking becomes necessary, I had to break apart this expression to generate only one sample at a time, such that it can be vetted for collision before being added as a sample.

M2

My implementation follows the method description in the problem statement, producing the following typical adjacency matrix. We can see in the figure that it is symmetric across the NW-SE diagonal, as we expect.

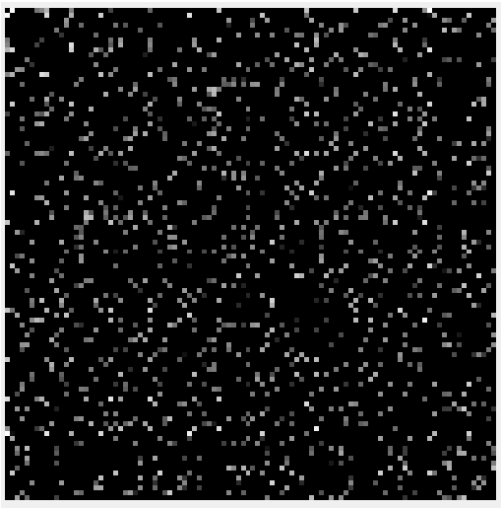


Figure 11: An example adjacency matrix.

M3

When the adjacency matrix and list of samples have already been generated, M3 runs very quickly, finding a path with 5-10 intermediate configurations to the goal position. It does this by creating a graph (whose nodes are indices in `samples`) from the adjacency matrix, and using the provided `shortestpath` function for graphs in MATLAB.

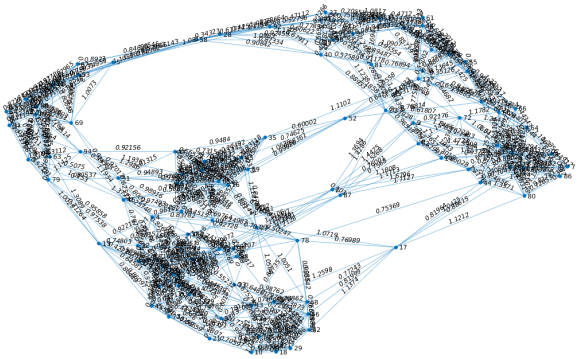


Figure 12: An example PRM graph produced during M3. The physical location of nodes in cspace has no bearing on this graph visualization.

M4

My RRT implementation is able to find a path which is usually around 70 to 100 nodes long. The arm sometimes finds a path to the left of the ball, and sometimes to the right. It runs in under a second, far faster than PRM if we include the adjacency matrix generation time.

The hyperparameters I chose are

- `step_size` = 0.05

- `freq_sample_goal` = 0.5
- `max_nodes` = 800
- `tolerance` = 0.05

If a node is within `tolerance` of the goal, we simply add the goal node and call the path done. If `max_nodes` nodes are added to the tree without the goal being found, my code will exit and report a failure to find a path.

M5

By removing stretches of unnecessary intermediary nodes, I'm able to reduce the path from M4 down to 5-9 configurations. This also runs extremely quickly, and produces a path like what we achieved with PRM.

M6

M6 is able to run my RRT and smoothing with no problems, and produces a valid path of around the same length as with the less challenging obstacle configuration. I'm very pleased with the effectiveness of my implementation.

All code is in the associated .zip file.