Algorithmic Robotics
COMP/ELEC/MECH 450 or COMP/ELEC/MECH 550
Project 5: Special Topics in Motion Planning

The documentation for OMPL can be found at http://ompl.kavrakilab.org.

In this assignment, you are free to select a project from one of several below or propose your own interesting motion planning problem. Those who propose a project must receive prior approval from the instructor. This project may be completed in pairs.

Deliverables

First, a statement of what project you are working is due **Saturday November 4th at 11:55pm** on Canvas.

An initial progress report is due **Thursday November 9th at 1pm** on Canvas. This report should be short, no longer than one page in PDF format. At a minimum, the report should state who your partner is, which project you are working on, and what progress you have made thus far. Those who propose their own project or deviate significantly from a proposed project must receive approval from the instructor prior to submitting their progress report. Those completing the project in groups need to provide just one progress report.

Final submissions are due Wednesday November 22nd at 1pm. Submissions are on Canvas and they consist of two things. First, to submit your project, clean your build space with `make clean`, zip up the project directory into a file named `Project2_<your NetID>_<partner's NetID>.zip`. Your code must compile within a modern Linux environment. If your code compiled on the virtual machine, then it will be fine. If you deem it necessary, also include a README with details on compiling and executing your code.

Second, submit a written report that summarizes your experiences and findings from this project. The report should be no longer than 10 pages, in PDF format, and contain (at least) the following information:

- A problem statement and a short motivation for why solving this problem is useful.
- The details of your approach and an explanation of how/why this approach solves your problem.
- A description of the experiments you conducted. Be precise about any assumptions you make on the robot or its environment.
- A quantitative and qualitative analysis of your approach using the experiments you conduct.
- Rate the difficulty of each exercise on a scale of 1–10 (1 being trivial, 10 being impossible). Give an estimate of how many hours you spent on each exercise, and detail what was the hardest part of the assignment.

Please submit the PDF write-up separate from the source code archive. Those completing the project in groups need to provide just one submission for code and report. In addition to the code and report, there will also be a short ($\sim$5 minute) in-class presentation on your chosen topic. Details of the presentation will be announced closer to the presentation dates.

If your code or your report is missing by the deadline above, your project is late. The latest date for submission is Friday at 5pm before the last week of classes. Your presentation is during the last week of classes.

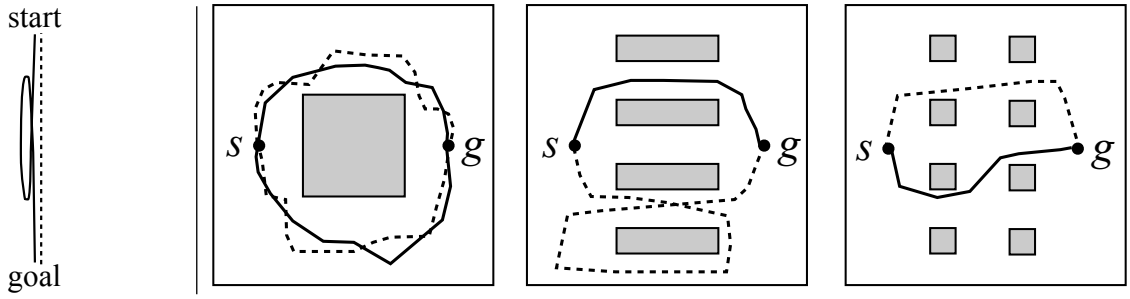
Project Topics

The topics proposed below are a broad spectrum of interesting and useful problems within robot motion planning. Each topic comes with its own set of questions and deliverables that should be addressed in the code and written report. For topics marked **450-only**, you may only choose the topic if both you and your partner are enrolled in the undergraduate versions of the class. The **graduate** section of the topics is required if either you or your partner is enrolled in the graduate versions of the course, and optional if you are both enrolled in the undergraduate versions. No extra credit will be rewarded for undergraduate students completing the graduate requirement. Although it may not be stated explicitly, *visualization is essential* for virtually every project, report, and presentation to establish correctness of the implementation.

5.1 Path Clustering (450-Only)

Sampling-based motion planning algorithms can produce many different paths for a given query. However, many of these paths can be virtually identical to one another. Whether this is the case is often difficult to determine, particularly if the configuration space has more than 3 dimensions since it is difficult to visualize such a space. In this project you will implement a distance metric for *paths* using a given distance metric for *configurations*. The path distance metric is then used to cluster a collection of paths that begin and end at the same configurations.

Path Distance:
The distance from a path A to a path B can be defined by integrating the distance to the closest point on B for a point moving along A. This formulation, however, is problematic since two very different paths can still have distance 0 (see Figure 1(a) for an example).



(a) Path distance should be significantly greater than 0 between the solid and dotted curve.

(b) Increasingly complex environments which result in increasingly many path clusters.

Figure 1: Path distance and path clustering.

A more sophisticated approach first constructs a monotonic mapping between curve-length parametrizations of A and B and then integrates the distance between corresponding points on A and B. Under the assumption that paths A and B are densely and uniformly sampled so that there are m states along A and n states along B, we can compute the distance using *dynamic programming*. Let $D_{i,j}$ be the distance between state $i \in A$ and state $j \in B$ using the standard distance metric for configurations. By taking advantage of the optimal substructure of this computation, we can compute the distance between the two paths by finding a minimum-distance mapping between the states in A and B. Let $C_{i,j}$ denote the minimum distance mapping up to state $i \in A$ and state $j \in B$. Then $C_{m,n}$ is the total distance between two paths, where $C_{1,1} = D_{1,1}$ and

$$C_{i,j} = \min[C_{i-1,j}, C_{i,j-1}] + D_{i,j}.$$

Path Clustering:
Once you have implemented a distance metric for paths, you can begin clustering paths that are *close* given your metric. You may choose any clustering algorithm: single-linkage (nearest neighbor) clustering, k-means, or more sophisticated techniques. Single-linkage and k-means are simple, heuristic approaches that attempt to group paths together into a set of clusters. Given the clustering output, one can visually assess the cluster quality using a *similarity* matrix. The similarity matrix is computed by sorting the data based on cluster id and then visualizing the elements of D (after normalization). An example of a good clustering and similarity matrix is shown in Figure 2.

If two paths are considered close according to the distance metric, but cannot be continuously deformed from one to another without passing through an obstacle, then these paths are topologically different. Ideally,

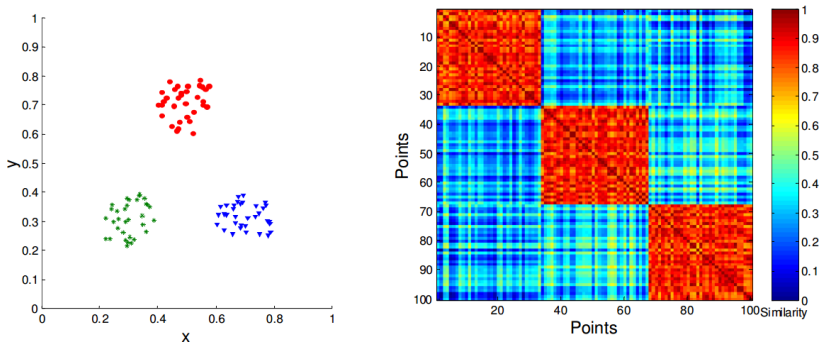


Figure 2: (Left) An example data set and grouping into three clusters. (Right) The similarity matrix for the clustering. Red squares along the diagonal indicate a good clustering.

we would like paths in different homotopy classes to end up in different clusters. This is a hard problem in general, but the following heuristic can potentially improve the clustering: if the straight-line path between $a_i \in A$ to $b_j \in B$ is not valid, then $D_{i,j}$ should change to a large value to discourage clustering paths in different homotopic classes. If all straight line paths between every a_i and corresponding b_j are valid, the original distance is returned.

Deliverables:
Implement the path distance function, with and without the homotopy check, and a clustering algorithm as described above. The implementation should work for any arbitrary, user-defined distance metric. Verify that the distance function reports a significant different between the two paths in Figure 1a. Evaluate your implementation in a variety of environments, like those in Figure 1b, to verify whether your clusters make sense. You may assume a point robot translating in $\mathbb{R}^2$.

How well does the path distance function and clustering algorithm classify your data? Does the homotopy heuristic improve the clustering? How did you select the number of clusters? Be sure to cluster both small and large numbers of paths, and to visualize the similarity matrix of your clustering when presenting your conclusions.

Protips:
Note that even in the simplest case with just one obstacle, there exist more than two clusters of paths (e.g., there is a cluster of paths that all go around the obstacle clockwise 5 times before reaching the goal). Generally speaking, there are infinitely many homotopy classes, thus infinitely many "ideal" clusterings.

To find many different paths, you must run a planner many times and extract a solution path from each run. It may be advantageous to generate paths from different planners for diversity. Moreover, you may also want to disable path simplification for some or all of the paths that you generate.

Bonus (5 pt):
Clustering algorithms can produce wildly different results on the same input. As a bonus experiment, compare and contrast two clustering algorithms on paths using the path distance function described above. You can choose any pair of clustering algorithms: single linkage (nearest neighbor) clustering, k-means, etc. Present a qualitative review of the clustering algorithms you choose. What are the strengths and weaknesses of the algorithms? Compare the clusterings computed by the algorithms.

5.2 Path optimization

The paths produced by sampling-based motion planning algorithms are often neither smooth nor short. They also don't necessarily guarantee a large clearance with respect to obstacles. However, these are often desirable properties for a path. Short paths take less time, smooth paths might be easier to follow by a robot, and with high-clearance paths small errors or changes in the environment do not cause a collision. A common approach to producing higher quality path is to post-process the output of the planners. Smooth, short, or high-clearance paths often exist in a 'neighborhood' of solutions produced by sampling-based motion planning algorithms. With local optimization techniques, one can often obtain high-quality paths with only a small additional post-processing cost.

The *smoothness* of a path can be measured, e.g., by the integral of curvature-squared along the path. For paths that consist of line segments, a discrete approximation of curvature can be defined as follows:

$$\kappa(i) = \frac{2\arccos(\frac{v_i}{\|v_i\|} \cdot \frac{v_{i+1}}{\|v_{i+1}\|})}{\|v_i\| + \|v_{i+1}\|},$$

where v_i is the vector from configuration $i-1$ to configuration i along the path [1]. The total 'curviness' (or cost) of the path is thus $\sum_i \kappa^2(i)$. You may also want to include a penalty for the number of configurations in your path, if you insert extra configurations as you optimize total curvature: cost $= \lambda n + \sum_i \kappa^2(i)$, where λ is a small constant and n is the number of configurations. To see why this might be a good idea, consider how the total curvature changes if extra interpolated configurations are inserted into a path without changing the shape of the path.

The *clearance* of a path can be defined as the sum (integral) of clearances of states along the path. The clearance of a state is simply the distance to the closest obstacle. This may not be easy to implement yourself in the general case, so you should use the state validity checkers implemented in OMPL.app that wrap around the FCL and PQP collision checking libraries. They both provide a function to compute clearance. The cost of a path can then be defined as the negated clearance of a path. See http://www.staff.science.uu.nl/~gerae101/motion_planning/clearance2.html for more information on clearance optimization.

One common technique to *shorten* paths is by repeatedly trying to connect random pairs of points along the path directly with the local planner. This is implemented in OMPL in the `ompl::geometric::PathSimplifier` class. Note that this technique does not necessarily smooth the path, although this tends to be a side-effect of path shortening. Other optimization techniques include:

- Iteratively perturbing points along the path to optimize a path segment. These perturbations can be random (in which case one needs to check that the perturbation will lead to an improvement) or by following a gradient of the optimization criterion as a function of the states along the path. Often extra states are inserted in the path, to give the algorithm more flexibility in the optimization. In OMPL extra states can added to a path by calling the `interpolate()` method of a solution path.
- Path hybridization: (1) compute multiple paths for a motion planning query, (2) compute "cross-over points," configurations on different paths that are close and allow you to jump from one path to the next, (3) compute a new, better path from the available path segments between crossover points. This is already implemented for path shortening in `ompl::geometric::PathHybridization`.

[1] See this paper for detail: T. Langer, A.G. Belyaev, and H.-P. Seidel, *Asymptotic analysis of discrete normals and curvatures of polylines,* in Proceedings of the 21st Spring Conference on Computer Graphics, pp. 229–232, 2005. DOI: 10.1145/1090122.1090160.

Deliverables:
For this assignment you will implement the path perturbation technique. Your implementation should work with arbitrary path cost functions. You will implement two path cost functions: path smoothness and path clearance, as defined above. Demonstrate that the path optimization improves the path quality.

Graduate:
Modify path hybridization to optimize for clearance.

Bonus (5 pt):
Modify path hybridization to optimize for smoothness. Path hybridization for smoothness is very non-trivial, since the curvature as defined above depends on triplets of states along a path.

5.3 Dynamic manipulation

NOTE: This project is popular with mechanical engineers, but is very complicated if you do not understand the math. Please only choose this project if you are confident with your ability to understand the referenced work.

Robotic manipulators consist of *links* connected by *joints*. They are often arranged in a serial chain to form an arm. Attached to the arm is a hand or a specialized tool. Clearly, manipulators are essential if we want robots to do useful work. However, planning for manipulators can be very challenging. The state of a manipulator can be described by a vector θ of joint angles (assuming there are only revolute joints). Usually the joint angles cannot be controlled directly. Instead, the motors apply torques to the joints to change their acceleration. The state vector then becomes $(\theta, \dot{\theta})$ (i.e., a vector that contains both joint positions and velocities). In this assignment you will develop an OMPL model for general planar manipulators with n revolute joints. The equations of motion are given in the first reference below. Although it is important that you understand this paper, *you do not need to derive any equations yourself.* Specifically, the kinematics and dynamics of a planar manipulator are given. The dynamics equations are of the form:

$$\tau = M(\theta)\ddot{\theta} + C(\theta, \dot{\theta}) + V(\theta),$$

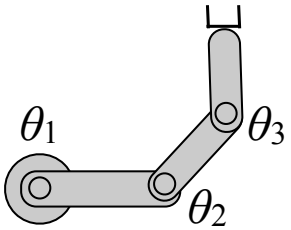


Figure 3: Three-link planar manipulator.

where M is the inertia matrix, C is the vector of Coriolis and centrifugal forces, and V is the vector of gravity forces. You need to rewrite them to obtain the following systems of ordinary differential equations:

$$\frac{d}{dt}\begin{pmatrix}\theta\\ \dot{\theta}\end{pmatrix} = \begin{pmatrix}\dot{\theta}\\ M^{-1}(\theta)\big(\tau - C(\theta,\dot{\theta}) - V(\theta)\big)\end{pmatrix}$$

and implement this as a "propagate" function [2]. For simplicity, you do not need to deal with friction or collisions. Given the "propagate" function, a planner needs to find a series of controls $\tau_1, \ldots, \tau_n$ that, when applied for $t_1, \ldots, t_n$ seconds, respectively, will bring a robot from an initial pose and velocity to a final pose and velocity.

When you test your code, it is helpful to check the total energy $E(\theta, \dot{\theta})$ when you integrate the equations above with $\tau = 0$ and the initial values for $(\theta, \dot{\theta})$ chosen arbitrarily (for a straight horizontal configuration you might have some intuition what motion would "look right"). In that case, the system is closed and the energy should remain constant (up to numerical precision). The energy is equal to $E(\theta, \dot{\theta}) = \frac{1}{2}\dot{\theta}^T M(\theta)\dot{\theta} + g\sum_{i=1}^{n} m_i h_i$, where $g = 9.81$ is the gravitational constant, m_i is the mass of link i, and h_i is the height of the center of mass of link i.

In the references below, the variables (x_i, y_i) do *not* refer to the position of joint i in the workspace, but the position of the end effector relative to the position of joint i. This seemingly odd parametrization is useful, because it greatly simplifies the dynamics equations. Using the notation of the references, the endpoint of link i is simply $(\sum_{i=1}^{n} l_i \cos\phi_i, \sum_{i=1}^{n} l_i \sin\phi_i)$. For visualizing the output of your program, you can print these positions and plot them with any plotting program you are familiar with.

References:

[1] L. Žlajpah, Dynamic simulation of n-R planar manipulators. In *EUROSIM '95 Simulation Congress*, Vienna, pp. 699–704, 1995. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.1622&rep=rep1&type=pdf. *This is a concise description of the dynamics.*

[2] You should not write your own numerical matrix inversion routine. Instead, use a linear solver (a function that finds x s.t. $Ax = b$) from a standard library, such as LAPACK, the C++ Eigen library, or Python's numpy module.

[2] L. Žlajpah, Simulation of n-R planar manipulators, *Simulation Practice and Theory*, 6(3):305–321, 1998. http://dx.doi.org/10.1016/S0928-4869(97)00040-2. *This paper has a few typos, but it has more details on how to compute* $C(\theta, \dot{\theta})$.

Deliverables:
You will implement a general state space for dynamic planar manipulators with n revolute joints. You need to produce solutions for manipulators with 3 joints between two states with arbitrary joint positions and velocities. It should be possible to set the number of joints, the link lengths, and limits on joint velocities and torques. If you model each joint position with a SO2StateSpace (which is recommended), you do not have to worry about joint *position* limits (the velocity and torque limits are still important). If the limits are too small, it may not be possible to solve a given motion planning problem, while large limits might force a planner to search parts of the configuration space that are not all that useful.

The mass matrix M (called H in the papers above) is defined in equations 8–11 of ref. 1 above. The Coriolis and centrifugal term C (called h in the papers above) is defined by the fourth equation on p. 310 of ref. 2, equation 11 and the two equations right above it (all in ref. 2). Finally, the gravity term V (called g in the papers above) is defined in equation 15 of ref. 1. There is a typo in ref. 2 when computing the Coriolis and centrifugal term C:

$$\frac{\partial^2 y_{ci}}{\partial q_j \partial q_k} = \begin{cases} -y_r + y_i - l_{ci} \sin(\varphi_i), & j \leq i \text{ and } k \leq i \\ 0, & j > i \text{ or } k > i \end{cases}$$

This is correct in reference 1.

There is another typo when computing the gravity term. Equation 15 in ref. 1 and Equation 13 in ref. 2 differ on the initial subscript in the sum. The correct subscript is $k = i + 1$.

Graduate:
Implement collision checking for planar manipulators. You can model a manipulator as a collection of connected line segments.

Bonus (5 pt):
Compare the solutions you get by using a sampling-based planner with those obtained with a pseudo-inverse type controller as described in the references for a variety of queries.

5.4 Centralized Multi-robot Planning

Planning motions for multiple robots that operate in the same environment is a challenging problem. One method for solving this problem is to compute a plan for all of the robots simultaneously by treating all of the robots as a single composite system with many degrees of freedom. This is known as centralized multi-robot planning. A naive approach to solving this problem constructs a PRM for each robot individually, and then plans a path using a typical graph search in the composite PRM (the product of each PRM). Unfortunately, composite PRM becomes prohibitively expensive to store, let alone search. If there are k robots, each with a PRM of n nodes, the composite PRM has n^k vertices!

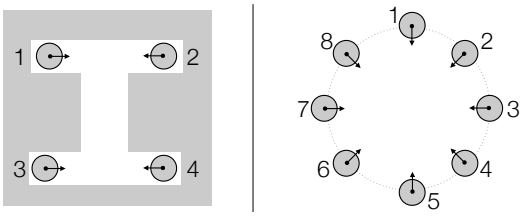


Figure 4: Two challenging multi-robot problems. *Left:* robots 1 and 2 need to swap positions with robot 4 and 3, respectively. *Right:* robot $i, i = 1,\ldots,4$, needs to swap positions with robot $i+4$.

Searching the exponentially large composite roadmap for a valid multi-robot path is a significant computational challenge. Recent work to solve this problem suggests *implicitly* searching the composite roadmap using a discrete version of the RRT algorithm (dRRT). At its core, dRRT grows a tree over the (implicit) composite roadmap, rooted at the start state, with the objective of connecting the start state to the goal state.

The key steps of dRRT are as follows:

1. Sample a (composite) configuration q_{rand} uniformly at random.
2. Find the state q_{near} in the dRRT nearest to the random sample.
3. Using an *expansion oracle*, find the state q_{new} in the composite roadmap that is connected to q_{near} in the closest direction of q_{rand}. (Figure 5).
4. Add the path from q_{near} to q_{new} to the tree if it is collision free.
5. Repeat 1-4 until until the goal is successfully added to the tree.

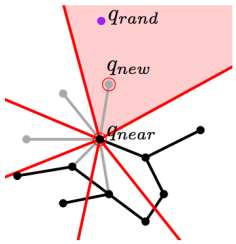


Figure 5: The dRRT expansion step. *From Solovey et al.; see below.*

Deliverables:
Implement the dRRT algorithm for multi-robot motion planning. For n robots, you will first need to generate n PRMs, one for each robot. If your robots are all the same, one PRM will suffice for every robot. Then use your dRRT algorithm to implicitly search the product of the n PRMs. The challenges here are the implementation of the expansion oracle (step 3) and the local planner to check for robot-robot collisions (step 4). The reference below gives details on one possible expansion oracle (section 3.1) and local planner (section 4.2). Evaluate your planner in scenarios with *at least* four robots, like those in Figure 4. For simplicity you may assume that the robots are planar rigid bodies. You may need to construct additional scenarios to evaluate different properties of the dRRT algorithm.

Address the following in your report: Is your dRRT implementation always able to find a solution? How does the algorithm scale as the number of robots increases? Elaborate on the relationship between the quality of the single robot PRMs and the time required to run dRRT. What do the final solution paths look like?

Reference:

K. Solovey, O. Salzman, and D. Halperin, Finding a needle in an exponential haystack: Discrete RRT for exploration of implicit roadmaps in multi-robot motion planning. In *Algorithmic Foundations of Robotics*, 2014. http://robot.cmpe.boun.edu.tr/wafr2014/papers/paper_20.pdf.

5.5 Decentralized Multi-robot Coordination

Planning motions for multiple robots simultaneously is a difficult computational challenge. Algorithms that provide hard guarantees for this problem usually require a central representation of the problem, but the size and the dimension of the search space limits these approaches to no more than a handful of robots. Decentralization or distributed coordination of multiple robots, on the other hand, is an effective (heuristic) approach that performs well in practical instances by limiting the scope of information reasoned over at any given time. There exist many effective methods in the literature to practically coordinate the motions of hundreds or even thousands of moving agents.

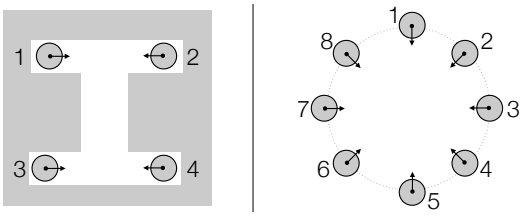


Figure 6: Two challenging multi-robot problems. *Left:* robots 1 and 2 need to swap positions with robot 4 and 3, respectively. *Right:* robot $i, i = 1, \ldots, 4$, needs to swap positions with robot $i+4$.

One popular decentralized approach is to assign a priority to each robot and compute motion plans starting with the highest priority. The robot with the highest priority ignores the position of all other robots. Motion plans for subsequent (lower priority) robots must respect the paths for the previously planned (higher priority) robots. In short, the higher priority robots become moving obstacles that lower priority robots must avoid. An example of such an approach is detailed in *Berg et al. 2005*, cited below.

A more recent approach reasons over the relative velocities between robots, employing the concept of a *reciprocal velocity obstacle* to simultaneously select velocities for each robot that both avoid collision with other robots and allow each robot to progress toward its goal. This technique is used not only in robotics but also in animation and even video games (e.g., Warhammer 40k: Space Marine, Crysis 2). Briefly, a reciprocal velocity obstacle RVO^B_A defines the space of velocities of robot A that will cause A to collide with another robot B at some point in the future, given the current position and velocity of A and B. Using this concept, a real-time coordination algorithm can be developed that simulates the robots for some short time period, then recalculates the velocities for each robot by finding a velocity that lies outside the union of each robot's RVO.

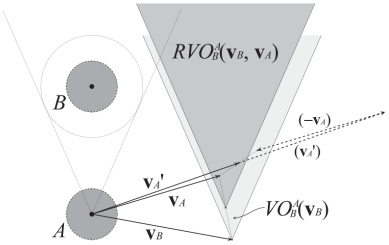


Figure 7: The reciprocal velocity obstacle formed by robot B's current position and velocity relative to robot A. *From Berg et al. 2008; see below.*

Deliverables:
Implement one of the methods above for decentralized multi-robot coordination and evaluate the method in multiple scenarios with varying numbers of robots. **Graduate students must implement reciprocal velocity obstacles**. You may assume that the robots are disks that translate in the plane.

Address the following: Is your method always successful in finding valid paths? Expound upon the kinds of environments and/or robot configurations that may be easy or difficult for your approach? How well does your method scale as you increase the number of robots? Qualitatively, how do the solution paths look? Depending on the method you chose, answer the following questions.

Prioritized method: Compare and contrast the paths for higher priority robots and lower priority robots. How sensitive is your method to the assignment of priorities?

RVO method: Discuss the challenges faced by the *RVO* approach as the number of robots increases. Is there always a velocity that lies outside of the *RVO*? How practical do you think this algorithm is?

Protips:
For the prioritized method, the key challenge is the implementation of the prioritized state validity checker. Not only must this function check that the robot being planned does not collide with obstacles, but also for collisions with the higher priority robots that have been previously planned. The notion of time must be incorporated into the configuration space (and possibly the planner as well) to ensure collision-free coordination. A modified version of RRT, EST, or similar planner to generate the single-robot paths is sufficient.

The implementation of the *RVO* method requires reasoning over the velocities of your robots. You can greatly reduce the complexity of the implementation by planning geometrically and bounding the velocity by limiting the distance a robot can travel during any one simulation step. Moreover, the *RVO* implementation requires reasoning over the robot geometry; choose simple geometries (circles) and ensure your implementation is correct before considering more complex geometries.

Visualization, particularly animation, is key in debugging these methods. Matlab, Matplotlib, and related packages support generating videos from individual frames.

References:

J. van den Berg and M.H. Overmars, Prioritized Motion Planning for Multiple Robots. In *IEEE/RSJ Int'l Conference on Intelligent Robots and Systems*, pp. 2217-2222, 2005. http://arl.cs.utah.edu/pubs/IROS2005-2.pdf.

J. van den Berg, M. Lin, and D. Manocha, Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation. In *IEEE Int'l Conference on Robotics and Automation*, pp. 1928-1935, 2008. https://www.cs.unc.edu/~geom/RVO/icra2008.pdf.

5.6 Planning Under Uncertainty (450-Only)

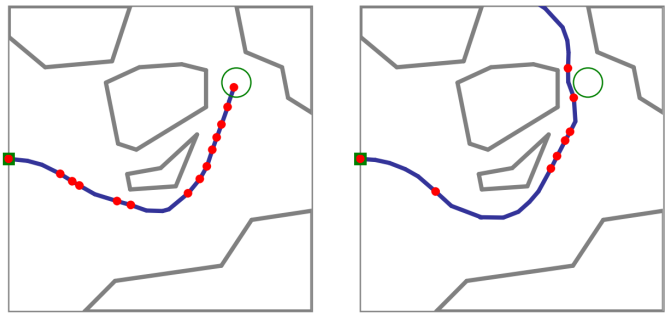


Figure 8: Even under an optimal policy, a system with actuation uncertainty can fail to reach the goal (green circle) consistently. *From Alterovitz et al.; see below.*

For physical robots, actuation is often imperfect: motors cannot instantaneously achieve a desired torque, the execution time to apply a set of inputs cannot be hit exactly, or wheels may unpredictably slip on loose terrain just to name a few examples. We can model the actuation error for many situations using a conditional probability distribution, where the probability of reaching a state q' depends on the initial state q and the input u, written succinctly as $P(q'|q,u)$. Because we cannot anticipate the exact state of the robot *a priori*, algorithms that reason over uncertainty compute a *policy* instead of a path or trajectory since the robot is unlikely to follow a single path exactly. A policy is a mapping of every state of the system to an action. Formally, a policy π is a function $\pi : Q \to U$, where Q is the configuration space and U is the control space. When we know the conditional probability distribution, an optimal policy to navigate the robot can be computed efficiently by solving a Markov decision process.

Markov decision process (MDP):
When the transitional probability distribution depends only on the current state, the robot is said to have the Markov property. For Markov systems, an optimal policy is obtained by solving a Markov decision process (MDP). An MDP is a tuple (Q,U,P,R), where:

- Q is a set of states
- U is a set of controls applied at each state
- $P : Q \times U \times Q \to [0,1]$ is the probability of reaching $q' \in Q$ via initial state $q \in Q$ and action $u \in U$.
- $R : Q \to \mathbb{R}$ is the reward for reaching a state $q \in Q$.

An optimal policy over an MDP maximizes the total *expected reward* the system receives when executing a policy. To compute an optimal policy, we utilize a classical method from dynamic programming to estimate the expected total reward for each state: Bellman's equation. Let $V_0(q)$ be an arbitrary initial estimate for the maximum expected value for q, say 0. We iteratively update our estimate for each state until we converge to a fixed point in our estimate for all states. This fixed-point algorithm is known as *value iteration* and can be succinctly written as

$$V_{t+1}(q) = R(q) + \max_{u \in U} \sum_{q'} P(q'|q,u) V_t(q'), \tag{1}$$

We stop value iteration when $V_{t+1}(q) = V_t(q)$ for all $q \in Q$. Upon convergence, the optimal policy for each state q is the action u that maximizes the sum in the equation above.

Sampling-based MDP:
Efficient methods to solve an MDP assume that the state and control spaces are discrete. Unfortunately, a robot's configuration and control spaces are usually continuous, preventing us from computing the optimal policy exactly. Nevertheless, we can leverage sampling-based methods to build an MDP that approximates the configuration and control space. The *stochastic motion roadmap* (SMR) is one method to approximate a continuous MDP. Building an SMR is similar to building a PRM; there is a learning phase and a query phase. During the learning phase, states are sampled uniformly at random and the probabilistic transition between states are discovered. During the query phase, an optimal policy over the roadmap is constructed to bring the system to a goal state with maximum probability.

Learning phase: During the learning phase, an MDP approximation of the evolution of a robot with uncertain actuation is constructed. First, n valid states are sampled uniformly at random. Second, the transition probabilities between states are discovered. SMR uses a small set of predefined controls for the robot. The transition probabilities can be estimated by simulating the system m times for each state-action pair. Note that m has to be large enough to be an accurate approximation of the transition probabilities. For simplicity, we assume that the uncertainty is represented with a Gaussian distribution. Once a resulting state is generated from the state-action pair, you can match it with the n sampled states through finding its nearest neighbor, or if a sampled state can be found within a radius. It is also useful to add an *obstacle* state to the SMR to indicate transitions with a non-zero probability of colliding with an obstacle.

Query phase: The SMR that is constructed during the learning phase is used to extract an optimal policy for the robot. For a given goal state (or region), an optimal policy is computed over the SMR that maximizes the probability of success. To maximize the probability of success, simply use the value iteration algorithm from equation 1 with the following reward function:

$$R(q) = \begin{cases} 0 & \text{if } q \text{ is not a goal state,} \\ 1 & \text{if } q \text{ is a goal state.} \end{cases}$$

Note: it is assumed that the robot stops when it reaches a goal or obstacle state. In other words, there are no controls for the goal and obstacle states and $V_t(q) = R(q)$ is constant for these states.

Deliverables:
Implement the SMR algorithm for motion planning under action uncertainty with a 2D steerable needle. Modeling a steerable needle is similar to a Dubins car; see section III of the SMR paper (cited below) for details on the exact configuration space and control set. Construct some interesting 2D environments and visualize the execution of the robot under your optimal policy. Simulated execution of the steerable needle is identical to the simulation performed when constructing the transition probabilities. Simulate your policy many times. Does the system always follow the same path? Does it always reach the goal? How sensitive is probability of success to the standard deviation of the Gaussian noise distribution?

Reference:
R. Alterovitz, T. Siméon, and K. Goldberg, The Stochastic Motion Roadmap: A Sampling Framework for Planning with Markov Motion Uncertainty. In *Robotics: Science and Systems*, pp. 233–241, 2007. http://goldberg.berkeley.edu/pubs/rss-Alterovitz2007_RSS.pdf.

Bonus (5 pt):
The value $V(q)$ computed by value iteration for each state q is the estimated probability of reaching the goal state under the optimal policy of the SMR starting at q. As a bonus experiment, we can check how accurate this probability is. Keeping the environment, start, and goal fixed, generate 20 (or more) SMR structures with n sampled states and compute the probability of reaching the goal from the start state using value iteration. Then simulate the system many times (1000 or more) under the same optimal policies. How does the difference between the expected and actual probability of success change as n increases? Evaluate starting with a small n (around 1000), increasing until some sufficiently large n; a logarithmic scale for n may be necessary to show a statistical trend.