## Algorithmic Robotics
## COMP/ELEC/MECH 450/550

# Project 5: Multi-robot Planning and Coordination

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

In this assignment, you are free to select a project from the six below. This project must be completed in pairs and ***only by graduate students***. For **PhD students only**, you are allowed to propose your own project. But you must receive approval from the instructor prior to submitting your project statement on **October 18th**.

### Deliverables

First, a statement of what project you are working is due **Monday October 18th at 1pm** on Canvas.

An initial progress report is due **Thursday November 4th 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. Only one report needs to be submitted for each pair.

**Final submissions are due Tuesday November 23th at 1pm. Submissions are on Canvas and consist of two things.** First, to submit your project, clean your build space, zip up the project directory into a file named `Project5_<your NetID>_<partner's NetID>.zip`. You are allowed to use any language (C++, Python, etc.), but your code must compile and run within a modern Linux environment. If your code compiled on the virtual machine, then it will be fine. Also include a README with details on compiling and/or 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. In addition to the code and report, there will also be a short ($\sim$15 minute) in-class presentation on your chosen topic. Details of the presentation will be announced closer to the presentation dates.

**Note:** Each team need only provide one submission. When making the final submission, each student in the team must send a private email to the TAs describing in detail their own contribution to the project.

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 (November 26th). Your presentation is during the last week of classes.

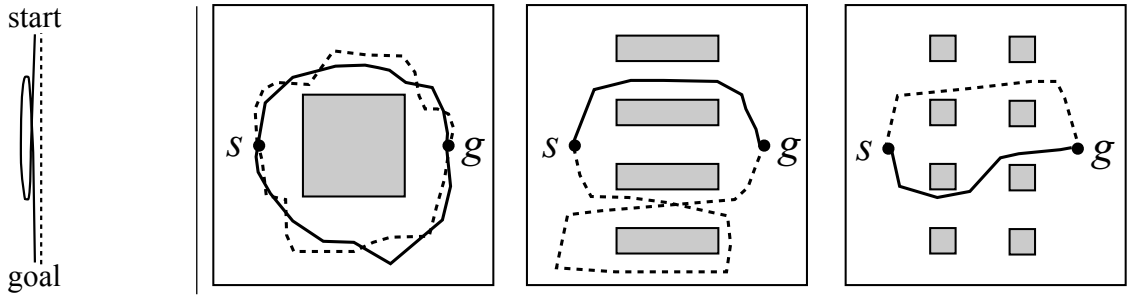
## Project Topics

The topics proposed below are a broad spectrum of interesting and useful problems for algorithmic robotics. Each topic comes with its own set of questions and deliverables that should be addressed in the code and written report. Please be aware that some of the topics are recommended for those having required knowledge. Although it may not be stated explicitly, *visualization is essential* for both projects, report, and presentation to establish correctness of the implementation.

## 5.1 Path Clustering

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,

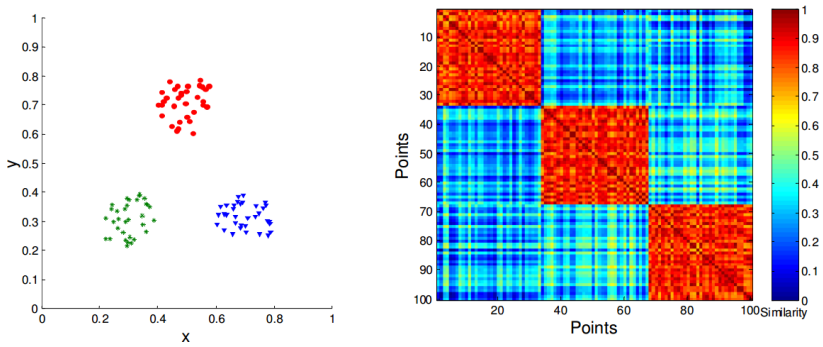



**Figure 2:** (Left) An example data set and grouping into three clusters. (Right) The similarity matrix for the clustering. Red squares along the diagonal indicate a good clustering.

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 $\mathbb{R}^2$.

**Part 1:** 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.

**Part 2:** Clustering algorithms can produce wildly different results on the same input. 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.

**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.

## 5.2 Planning Under Uncertainty

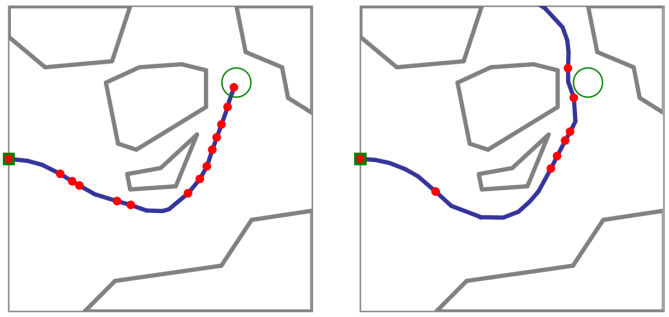


**Figure 3:** Even under an optimal policy, a system with actuation uncertainty can fail to reach the goal (green circle) consistently. *From Alterovitz et al.; see below.*

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 $\pi$ 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:**

**Part 1:** 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?

**Part 2:** 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 $q$. 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.

**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.

## 5.3 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 $\lambda$ 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:**
**Part 1:** 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.

**Part 2:** 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.

## 5.4 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 $\theta$ 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:

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

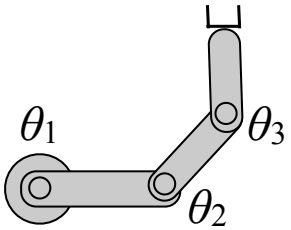



**Figure 4:** Three-link planar manipulator.

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:**
**Part 1:** 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 $k = i + 1$.

**Part 2:** 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.

## 5.5 Elastic Kinematic Chains

**NOTE:** This project is recommended for students that are interested in computational biology.

Let an elastic kinematic chain be defined as a planar kinematic chain with $n$ revolute joints, where each joint has an angular spring. The spring energy of the chain is $\sum_i u_i^2$, where $u_i$ corresponds to joint angle $i$. Suppose the base link is fixed at $(0,0)$ with orientation 0. Now imagine that the end effector is slowly moved around to different positions and orientations so that the arm is always at a stable equilibrium configuration. How many parameters would you need to describe *all* stable configurations for $n$ joints? The surprising answer is 3 for any $n$! There is a relatively straightforward algorithm to determine which configurations are stable (although understanding its correctness is far from trivial).

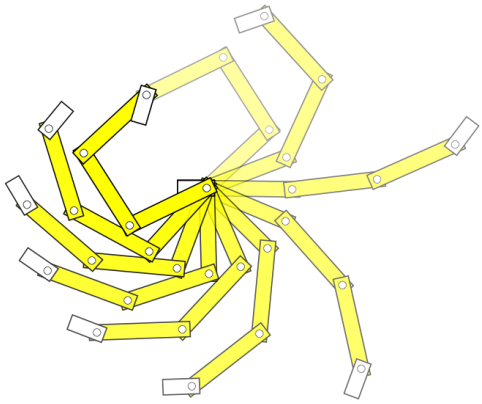


**Figure 5:** A sequence of stable configurations for a chain with 4 joints. *From McCarthy & Bretl; see below.*

There is one small typo in the algorithm in the reference below. The line that reads

$$P_{n-4} = (A^\dagger B)^T (I - NK)^T M (I - NK)(A^\dagger B)$$

should be changed to

$$P_{n-4} = Q_{n-4} + (A^\dagger B)^T (I - NK)^T M (I - NK)(A^\dagger B).$$

As you can tell from this one line, you will want to use a linear algebra package such as Eigen (a C++ header-only library) or Numpy (a Python library with many Matlab-like functions) for your implementation.

**Reference:**

Z. McCarthy and T. Bretl, Mechanics and Manipulation of Planar Elastic Kinematic Chains, in *IEEE Intl. Conf. on Robotics and Automation,* 2012. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6224693

**Deliverables:** Implement the `isStable` function from the reference above and use it as a StateValidityChecker to plan paths between stable configurations of an elastic kinematic chain. Add an option to check whether the joint angles (denoted by $u_i$ in the paper) are within $-\pi$ and $\pi$. Then use any OMPL planner to find a path of stable configurations between any pair of stable configurations. Next, write an algorithm that using this function finds a pair of configurations that requires a long path (in terms of number of states on the path). It is assumed that you will use path simplification on each path produced by the planner so that a long path is not simply the result of an unlucky run of the planner. How do the $[-\pi,\pi]$ joint limits change the results?

Picking reasonable bounds for the 3-parameter space of stable configurations is not entirely obvious. For a 10-joint arm you could use the following bounds on the configuration space: $[-15,15] \times [-15,15] \times [-\pi,\pi]$. They may have to be adjusted for a different number of joints. Implementing the bonus below can help you visualize when the free space becomes almost fractal-like (in which case, you would need to specify a very small value for setStateValidityCheckingResolution). Consider the symmetries that should exist in your c-space obstacles. If your visualization does not match the expected symmetries, there may be an error in your computations.

Write code that visualizes the free space of the three-parameter configuration space. This can be done by creating plots of 2D slices of the configuration space, as was done in the reference above. You don't actually

have to compute the boundaries of the configuration space obstacles. It is sufficient to create a dense 3D grid of the configuration space, where for each grid point you compute the state validity. You can then plot each slice of the grid by drawing invalid configurations as black pixels and free configurations as white pixels. If you are using the Python numpy module or Matlab, you can use the `spy` function.

## 5.6 Task Planning

Task planning focuses on planning the sequence of actions required to achieve a task. To solve a robotics problem using task planning, it is important to consider how to model the problem as a task planning problem, and use the appropriate technique for solving it.

Consider the following ”Sokoban on Ice” domain. A robot working in a grid world is tasked with reaching a target cell. Each cell may be empty, contain a static obstacle, or contain a movable box. The robot can move in any of the four cardinal directions. However, the floor is icy, so the robot will slip along the direction of motion until a static obstacle or the workspace boundary is hit (You may choose to model the boundary as static obstacles all-around). If the robot hits a movable box during its motion, the box gets pushed with the robot until the box hits another object (obstacle, another movable box, or workspace boundary).

An example domain is shown in on the right. The red block represent the robot, the green cell represent the goal, blue blocks represent movable boxes, and black blocks are obstacles. In this case, the robot could first move right, than down, then left, then up, to push the movable box upward. This will leave the movable box to the immediate right of the goal, and the robot right under the box. Then, the robot could move left, up, then right, to reach the green goal cell.

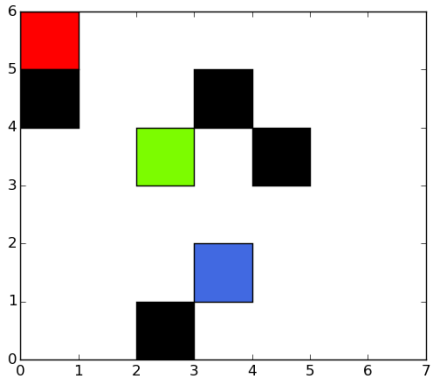



**Figure 6:** Scene 1

In this project, you will model the ”Sokoban on Ice” domain, and solve it by implementing a task planner. Make sure you are very clear on what are the state variables, what are the actions, and what are the preconditions and effects of the actions. Then, implement SAT-plan [1] to solve problems in your domain. Visualize your output to confirm the correctness of your implementation.

To implement SAT-plan, you may use the Z3 smt solver for SAT-solving https://github.com/Z3Prover/z3

**Protips:**
There is a trade-off between speed of generating the SAT instance and solving it. Using a naive encoding (e.g. using a proposition for each robot, each box, and each obstacle for each cell) could make it easy to encode, but make the SAT solver do more work. Using a smarter encoding (optimizing to reduce the number of variables, combining actions, removing propositions for obstacle cells) could make the generation of the SAT formula more difficult, but reduce the size of the SAT formula.

One trick to make encoding more efficient is to consider the boundary of the map as a rectangular frame of obstacles.

**References:**
[1] Kautz, Henry A., and Bart Selman. ”Planning as Satisfiability.” ECAI. Vol. 92. 1992. http://zones1v01.ing.unibo.it/Courses/AI/applicationsAI2009-2010/articoli/Planning_as_Satisfiability.pdf
[2] LaValle, Steven M. Planning algorithms. Cambridge university press, 2006. http://planning.cs.uiuc.edu/
*Chapter 2 has a very good explanation of SAT-plan.*

**Deliverables:**
First implement a planner that can solve the ”Sokoban on Ice” problem with no movable boxes. This reduces to the ”Ice Path” problem from Pokemon. Use your planner to solve the ice_path problem in figure 7. Then, expand your implementation to handle movable boxes. Use your planner to solve the problem in figure 6 and figure 8 provided in the accompanied zip file. Visualize your outputs to confirm that your solution is correct. Does your algorithm have any guarantees regarding the optimality of the solutions found? Investigate how the runtime of the algorithm is affected by the number of grid cells, as well as the depth of plan.

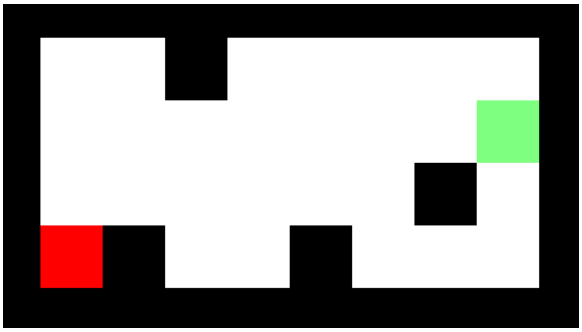


**Figure 7:** Ice Path

**Bonus (5 pt):**
Implement a search-based algorithm to solve your domain by performing BFS or A* (if you can construct a good heuristic) on the possible actions in your domain. Compare the runtime and quality of solution between using the search-based algorithm and SAT-plan.

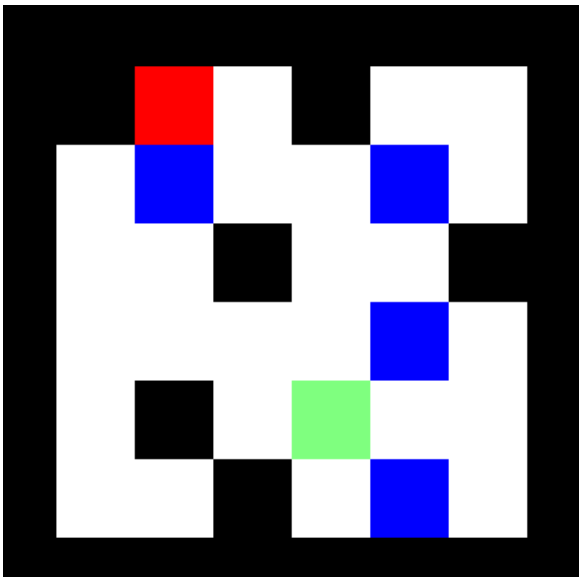


**Figure 8:** Scene 2