Motion Planning RBE 550

Final Project: Special Topics in Motion Planning

In this assignment, you are free to select a project from one of several below or propose your own interesting motion planning problem. Each project has a maximum number of people that could work on it. If you would like to have more people working on that project you would have to increase the scope of the project.

Deliverables

There will be 3 Deadline to keep in mind:

1. Declare which Project you are doing and the group, No Deliverable **Due: March 28**.
2. Interim Presentation, and Report. **Due: April 15**
3. Final Presentation, Report, and code. **Due: May 7**

1.1 Interim Presentation and Report

1. **Presentation:** It should be 4 min presentation +1 min Questions. An updated version of the init presentation focusing on your progress/results so far. It should include at minimum:
   (a) Team Members, and chosen topic.
   (b) Problem Description (In your own words, one slide).
   (c) Plan of Action, How will you solve the problem?.
   (d) What Tools/Platforms you will use.
   (e) Evaluation how will you evaluate your results.
   (f) Timetable and task delegation.
   (g) Initial results you might have.

   Not submitting the presentation will result in a 10% penalty.
2. **Report:** Should be compiled with the IEEE conference format and have the following components:
   - Introduction (Research Question & Application Impacts)
   - Related Work
   - Proposed Methods
   - Platform & Evaluation
   - Limitations of the project
   - Break-down contributions of team members

1.2 Final Presentation, Report and code

1. **Presentation:** The final presentation with all your results.
2. **Report:** Final Report including the information mentioned in the previous section
3. **Code:** A zip file containing your source code, with a README.md file explaining how to install the code, and how to run it to recreate figures and results from your report.

Final 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. Although it may not be stated explicitly, *visualization is essential* for virtually every project, report, and presentation to establish correctness of the implementation.

You are highly encouraged to use the tools that were presented in the class to complete these topics.

Additionally (after prior communication with the instructor) you are encouraged to:

1. Extend one the existing projects
2. Propose your own project

2.1 Planning Under Uncertainty (Max 4 Students)

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.

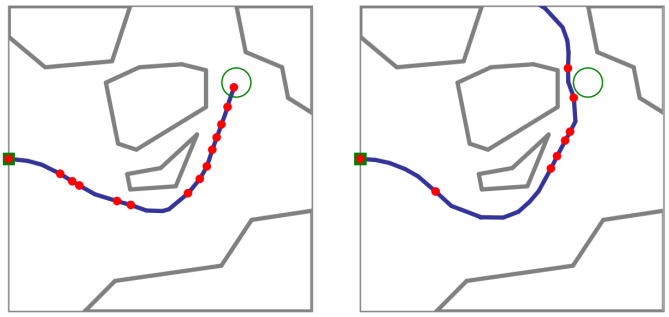

Figure 1: Even under an optimal policy, a system with actuation uncertainty can fail to reach the goal (green circle) consistently. *From Alterovitz et al.; see below.*

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.

POMDP:

Besides action uncertainty there can also be state uncertainty in the robot or the world robot. The partially observable Markov decision process (POMDP) is a mathematically principled framework for modeling decision-making problems in the nondeterministic and partially observable scenarios mentioned above. POMDPs stochastically quantify the nondeterministic effects of actions and errors in sensors and perception. They estimate the robot's state as probability distribution functions over states, called beliefs, and compute the best actions to perform with respect to these beliefs, rather than with respect to single states. The computed action strategies automatically balance information gathering and goal attainment. This concept is powerful: It is general and could enable robust operation even when the robot operates near an environmental boundary or near the limit of the robot's capability.

Deliverables:
Implement the SMR algorithm for motion planning under action and state 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. Note that simple modeling the problem as an MDP as in the SMR paper will not account for state uncertainty. You can use the example from Figure 2 of the second reference for an example on how to model state uncertainty and an appropriate observation model.

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?

Bonus (20 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.

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.

Kurniawati, Hanna. "Partially observable markov decision processes and robotics." Annual Review of Control, Robotics, and Autonomous Systems 5.1 (2022): 253-277. https://www.annualreviews.org/docserver/fulltext/control/5/1/annurev-control-042920-092451.pdf?expires=1729871823&id=id&accname=guest&checksum=1A42DEE399E4C0DE73E37C38DDB91234

2.2 Dynamic manipulation (Max 4 students)

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:

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

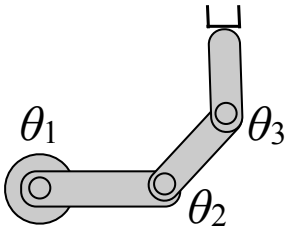

Figure 2: Three-link planar manipulator.

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 [1]. 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.

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

[1] 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.

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 $k = i + 1$.

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

Bonus (20 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.

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] 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* $C(\theta, \dot{\theta})$.

2.3 Centralized Multi-robot Planning (Max 3 Students, No bonus)

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!

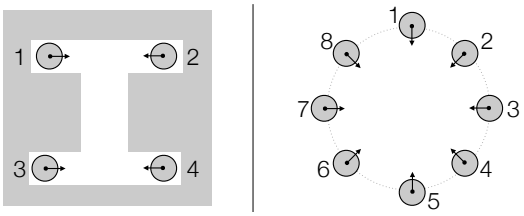

Figure 3: Two challenging multi-robot problems. *Left:* robots 1 and 2 need to swap positions with robot 4 and 3, respectively. *Right:* robot $i, i = 1,\ldots,4$, needs to swap positions with robot $i+4$.

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 4).
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.

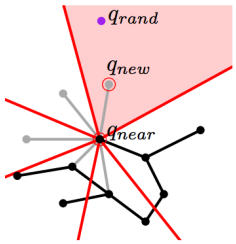

Figure 4: The dRRT expansion step. *From Solovey et al.; see below.*

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 3. 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.

2.4 Decentralized Multi-robot Coordination (Max 4 Students)

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.

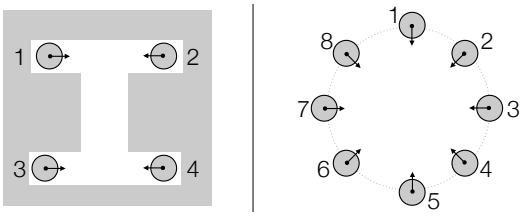

Figure 5: Two challenging multi-robot problems. *Left:* robots 1 and 2 need to swap positions with robot 4 and 3, respectively. *Right:* robot $i, i = 1,\ldots,4$, needs to swap positions with robot $i+4$.

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.

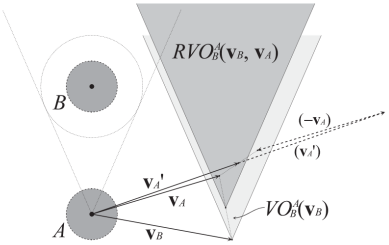

Figure 6: The reciprocal velocity obstacle formed by robot B's current position and velocity relative to robot A. *From Berg et al. 2008; see below.*

Deliverables:
Implement both of the methods above for decentralized multi-robot coordination and compare them in multiple scenarios with varying numbers of robots. 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.

Bonus (20 pt):

RVO vs Prioritezed method: Compare and contrast the performance of these two methods in terms of success rate and computational. Make a comprehensive analysis of the advantages and disadvantages of each one.

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.

2.5 Task Planning(Max 3 students)

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.

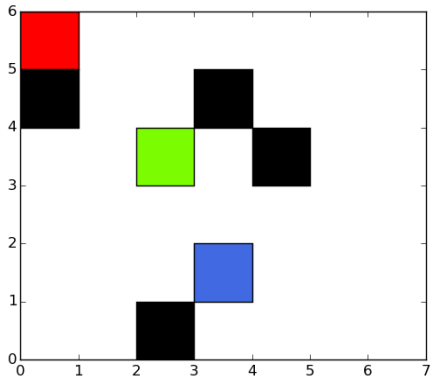

Figure 7: Scene 1

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 8. Then, expand your implementation to handle movable boxes. Use your planner to solve the problem in figure 7 and figure 9 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.

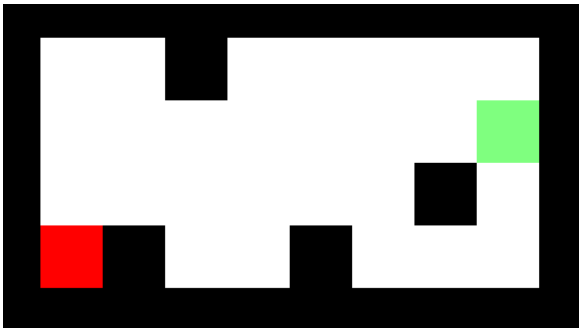

Figure 8: Ice Path

Bonus (20 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.

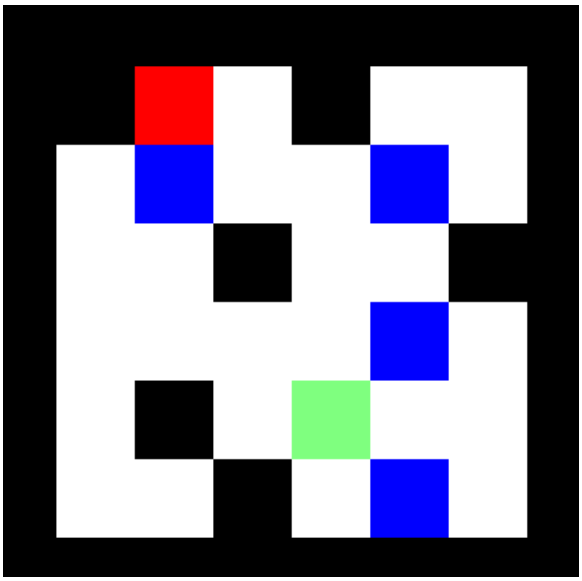

Figure 9: Scene 2

Research Project Topics

These topics are relevant for some of the existing research in the ELPIS Lab. If you are interested in one of these please schedule a meeting with the instructor to discuss the details of the projects. These projects will be using some of the internal lab tools that are currently not publicly available, and will also be more closely supervised from the supervisor.

2.6 Planning Under Uncertainty with Chance Constrained RRT (Max 3 Students)

Motion planning under uncertainty is a fundamental challenge in robotics. Real-world robots rarely have perfect information; there are uncertainties in robot motion, sensing, and obstacle locations. How can a robot plan a path that is probably safe despite these uncertainties? Chance-constrained motion planning offers a compelling approach: instead of requiring absolute certainty of collision-free paths (often impossible in uncertain environments), we allow a small probabilistic risk of collision that can be bounded. In this project, students will tackle the problem of planning robot motions under uncertainty by implementing the Chance-Constrained Rapidly-Exploring Random Tree (CC-RRT)

. CC-RRT is a variant of the popular sampling-based RRT planner that incorporates chance constraints to ensure that the probability of collision along the planned path stays below a specified threshold. The appeal of CC-RRT lies in its ability to balance safety and feasibility: it accepts a tiny risk of collision (e.g. allowing up to 5% probability) in exchange for finding solutions in cases where a strictly safe path might not exist. This makes CC-RRT well-suited for real-time planning in complex, uncertain environments, providing higher success rates than overly conservative methods that never find a path at all. 10.

The core goal of this project is to reimplement the CC-RRT algorithm (as described by Luders et al. 2010) in a minimal 2D simulation environment. Students will build a simple custom 2D world to simulate a robot navigating among obstacles. We will leverage the Open Motion Planning Library (OMPL) to handle basic sampling-based planning components — for example, OMPL can provide the standard RRT planner as a starting point. On top of this, students will add the necessary extensions for handling uncertainty and chance constraints:

- **State Representation**: Instead of planning with deterministic states, CC-RRT plans in the space of state distributions. In practice, this means associating a probability distribution (e.g. a Gaussian with mean and covariance) to the robot's state at each node of the tree. As the tree expands, uncertainty is propagated along each edge (e.g. using a simple motion model with process noise to update the covariance).
- **Chance Constraints**: Every new node (state distribution) generated by the RRT will be subjected to a probabilistic feasibility check. The algorithm will compute the probability that the robot in that state would collide with any obstacle or violate constraints. If this collision probability exceeds a chosen threshold (the chance constraint), the node is discarded (treated as unsafe). Otherwise, the node is retained in the tree and can be used to continue growing future branches. Geometrically this can be though as inflating the obstacles by an uncertainty margin or checking if the "uncertainty ellipse" around the robot intersects any obstacles (as illustrated in the figure above).
- **Tree Expansion**: The planner will continue extending the RRT tree in the usual way (sampling random points, steering from nearest neighbors, etc., which OMPL can assist with), but it only keeps those nodes and edges that meet the chance-constrained safety criterion. Over time, the tree will hopefully find a path that reaches the goal region with high probability of safety. If a branch of the tree ventures too close to an obstacle (such that the risk of collision is too high), that branch is pruned and the RRT will explore alternative routes.
- **2D Simulation Environment**: Students will implement a lightweight 2D simulator to test the planner. This involves defining some obstacles (e.g. polygons or circles in the plane) and a robot model (which can be as simple as a point mass or circle with a given motion uncertainty). The simulator will not use Genesis or other heavy robotics platforms; instead, part of the learning experience is building

the environment from scratch or with simple libraries (for example, using matplotlib or Pygame for visualization). This keeps the focus on the algorithmic aspects of CC-RRT.

Bonus (20 pt):
A bonus part will be available after discussion with the instructor.

References:
Luders, Brandon, Mangal Kothari, and Jonathan How. "Chance constrained RRT for probabilistic robustness to environmental uncertainty." AIAA guidance, navigation, and control conference. 2010. https://core.ac.uk/download/pdf/4430576.pdf.

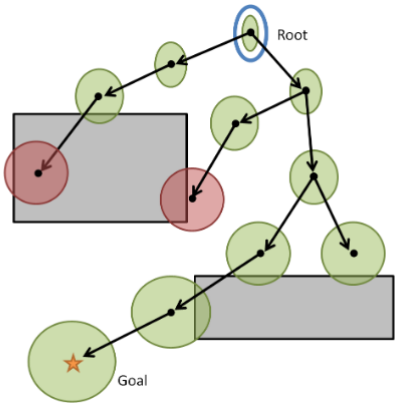

Figure 10: Chance constrained RRT in practice

MotionBenchMaker in Genesis

Evaluating motion planning algorithms for complex robot manipulators can be challenging and time-consuming. Researchers have found that having a diverse set of benchmark problems is critical for comparing planners and training new planning methods. **MotionBenchMaker** is one example of a tool created to generate such manipulation planning datasets in a standardized way. Inspired by this, in this project students will implement a simplified version of MotionBenchMaker using the *Genesis* simulator[2] as the underlying environment. The final goal is to create a small library of motion planning benchmark scenarios (e.g., pick-and-place tasks) and organize them into a reusable dataset for motion planning research.

Using Genesis allows us to leverage its high-performance physics simulation and built-in OMPL motion planning integration. This means that for each benchmark scenario, students can directly test sampling-based planners like RRT* or PRM through Genesis's API. Figure 11 illustrates different motion planning problems from MotionBenchMaker. By the end of the project, we will have a collection of saved scenarios and queries that can be easily shared or extended for further experimentation.

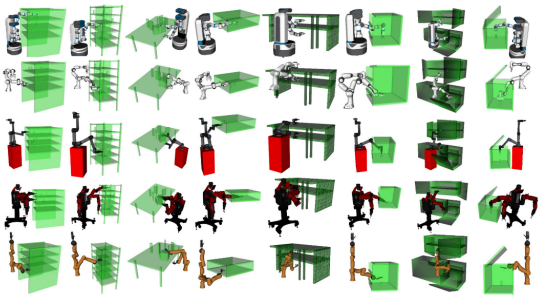

Figure 11: Examples of manipulation benchmark tasks, such as retrieving an object from a container (left) and reaching into a shelf (right), similar to scenarios from

Implementation

Students will develop the following components step-by-step:

1. **Port MotionBenchMaker Scenarios to Genesis** Write a wrapper to convert available MotionBenchMaker motion planning problems to the genesis simulator.
2. **Benchmark Task Design:** Define a new set of manipulation tasks in Genesis. For example, create a *pick-and-place* scenario where a robot arm must move an object from one location to another, and a *rearrangement* scenario where multiple objects need to be moved into new configurations. Each task will involve setting up the environment with a robot (e.g., a Panda arm) and objects (e.g., blocks, containers) with initial and goal poses.
3. **Scenario Data Saving:** For each task, implement a method to save the important details of the scenario (the scene layout, object configurations, robot start state, and the planning query such as the goal configuration or end-effector target) to a file. This could be a simple structured format (for example, JSON or YAML) that records all information needed to recreate the scene and the motion planning problem. The aim is to build a small dataset of these scenarios that others can reuse or extend.
4. **Planner Integration and Testing:** Utilize Genesis's OMPL integration to run motion planning algorithms on each saved scenario. Genesis uses the OMPL library for planning, accessible via a simple API call (e.g., planning a path to a target joint configuration). Students should set up one or more planners (for instance, RRT*, PRM, or RRT-Connect) and attempt to solve each benchmark problem. The code should record whether a solution was found and relevant metrics like planning time or path length for each attempt.

[2]X. Zhou *et al.*, "Genesis: A Universal and Generative Physics Engine for Robotics and Beyond," 2024.

During implementation, students can directly use Genesis's functions to plan motions rather than writing planners from scratch. This project emphasizes scenario creation and data organization, using the existing planners in Genesis to validate that the scenarios pose interesting challenges for motion planning.

Bonus (20 pt)

For extra credit, you can discuss with the instructor,

References:

Chamzas, Constantinos, et al. "Motionbenchmaker: A tool to generate and benchmark motion planning datasets." IEEE Robotics and Automation Letters 7.2 (2021): 882-889. https://www.kavrakilab.org/publications/chamzas2022-motion-bench-maker.pdf.