An Introduction to Robotic Motion Planning, Sampling-Based Algorithms, and the Refactoring Process of the Adaptive Neighborhood Connector Method in the Open-Source Parasol Planning Library

Ana Elissa Cabrera
DREAM Fellowship Research Student
Master's in Computer Science Class of 2024
University of Illinois at Urbana-Champaign, aec4@illinois.edu

Nancy N. Amato
DREAM Fellowship Research Advisor
Head of the Computer Science Department
Co-Director of the Parasol Laboratory
University of Illinois at Urbana-Champaign, namato@illinois.edu

***Abstract* - My project for the Distributed REsearch Apprenticeships for Master's (DREAM) Assistantship was to research motion planning and robotics, experiment with probabilistic motion planning algorithms, and to help refactor the outdated Adaptive Neighborhood Connector (ANC) algorithm code for Probabilistic Roadmaps (PRM) in the Parasol Planning Library, an open-source codebase developed by the Parasol Laboratory at the University of Illinois at Urbana-Champaign. This paper introduces motion planning applied to robotics and its history, previous works in the field, research and experiments on key motion planning algorithms, and my open-source code refactoring process of the Adaptive Neighborhood Connector Method.**

Introduction

Motion planning is a branch of robotics that, *given* an environment with a moveable robot, obstacles, and start and goal configurations, *computes* a valid collision-free path that takes the robot from a start to a goal position. For the PPL crash course, I completed five modules where I read revolutionary research papers on motion planning and robotics; coded motion planning programs in C++ and XML; experimented with varying environments, planners, robots, algorithms, and other parameters; improved my debugging strategies; wrote reports on my experiment results; and completed quizzes.

Module 0: Access Accounts and Webpages

A. Set Up

I requested access to the Parasol virtual machines (VM), laboratory facility, and added myself to the Parasol website. My work was done on the Black Widowers and Nancy Drew VMs, which run on MATE Linux. I configured the FastX3 toolkit to render the remote laboratory Linux servers on my laptop. My laptop model is Lenovo ThinkPad X1 Carbon Generation 9, runs on Ubuntu Linux 22.04, has 16.0 gibibyte memory, and a 11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz × 8 processor.

B. Readings

The first required reading was *How to Read a Paper* (Keshav, 2007). According to Keshav, you do not have to read an entire research paper as the first instinct. Instead, tackle the paper in three "passes". In the first pass, only read the abstract, titles and subtitles of each section and conclusion. In the second pass, only look at visual data such as graphs, diagrams, images, and captions. This foreshadows experimental results and can indicate if the data is realistic and reliable. The third pass consists of reading the entire paper, but is not required if it is not relevant to your research. I applied Keshav's three-pass reading algorithm to the robotics papers I found while researching, since not all of them were on motion planning. This made my technical readings significantly simpler and more efficient.

The second required reading I completed was *A Journey of Robots, Molecules, Digital Actors, and Other Artifacts (Latombe, 1999).* Latombe stated the basic **motion planning problem** is to find a collision-free path for a robot among rigid static obstacles, and is a purely geometric problem that looks simple but is computationally hard. In 1966, the first mobile robot system with motion planning capabilities, Shakey, was described by Nils J. Nilsson in the time of his tenure at the then Stanford Research Institution, presently known as SRI International. Nilsson introduced the visibility graph method, combined with the A* algorithm, to find the shortest collision-free path for Shakey, represented by a point amid polygonal obstacles (Nilsson, 1984). This robotic dot representation technique is popular in motion planning to this day, and will be explained in the Module 1.

In the late-1970s, the ideas of shrinking a robot to a point for collision avoidance (Udupa, 1977) and using a complete path planner for robots moving in translation among obstacles (Lozano-Oerez and Wesley, 1979) was proposed. The latter work led to the concept of the configuration space. In the mid-1980s, planners barely computed collision-free paths for planar objects

moving in two-dimensional workspaces. Common challenges included limited tools and technology to represent complex geometric models, slow and computationally expensive distance computations from collision checks, and the often unpredictable relation between the number of samples and the probability of not finding a path with randomized planners.

By the late-1990s, in nearly two decades, engineers significantly advanced motion planning research to better understand more complex problems where narrow passages exist (Amato et al, 1998), robots have kinematic and dynamic constraints, robots move in higher-dimensional spaces, optimized trajectory computations are required, and multiple robots are coordinated. Today, motion planning is applied to a plethora of fields such as industrial robotics for manufacturing, virtual prototyping, graphic animation, medical surgery, and computational biology. In addition to robotics, Parasol develops optimized algorithms for motion planning applications from fields such as computational biology, neuroscience, physics, and geophysics; computer-aided design (CAD); and virtual reality.

Module 1: Motion Planning and Configuration Space

A. Readings

The required reading was four sections of "Chapter 3: Configuration Space" found in *Principles of Robot Motion: Theory, Algorithms, and Implementations* by Choset et al (2005). This chapter argues that motion plans require us to give a specification of the location of every point of the robot to ensure that no point on the system collides with an obstacle. This chapter introduced concepts such as the configuration of a robot, degrees of freedom (DOF), configuration space (C-Space), configuration space obstacles (C-Obst) and free configuration space (C-Free).

The **configuration** of a robot system is a complete specification of the position of every point in that system. The **degrees of freedom** of a robot is the dimension of the configuration space. DOFs indicate the minimum number of parameters needed to specify the configuration. The **C-space** of a robot system is the space of all possible configurations of the system. The **C-Free** is the set of feasible configurations at which the robot does not intersect any obstacles. The **C-Obst** is the set of non-feasible configurations at which the robot intersects an obstacle in the workspace. The ∂**C-Obst** is is the set of points in C-space which have neighbors in both C-Obst and C-Free. In other words, this set of placements are where the robot contacts the obstacle but can still pass through it. This term can be traced back to set theory.

The most insightful part of my reading was the visualization of a robot in a two-dimensional ambient space — referred to as the workspace — into the C-space. Figure 1 and Figure 2 are found in the Principles of Robot Motion reading. In Figure 1: (a) a circular mobile robot approaches the workspace polygon obstacle and (b) slides around the obstacle to find the constraints this obstacle places on its configuration. By keeping track of the curve traced out by the center reference point, one can construct the C-Obst. In Figure 1(c), the motion planning for the circular robot in the workspace seen in Figure 1(a) was transformed into motion planning for a point robot in the C-space.

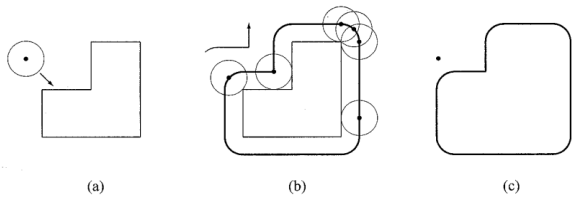


Figure 1. Transformation of Workspace into C-space for a Circular Mobile Robot
Taken from *Principles of Robot Motion* (Choset et al, 2005)

In Figure 2, three (point, circular, larger circular) mobile robots of different radii in the same environment each try to find a path from one configuration to the second one. The workplace representation of the robot is transformed into C-space representation by "growing" the polygon obstacle's walls outward and the walls inward. It might seem like the robots in Figure 1 (c) shrunk, but in reality, the obstacles "grew" when accounting for the points occupied by the robot body's volume, thus making it easier to see that it has no solution. Although these two figures are simplifications of motion planning problems, it was much easier to think about points moving around a space than bodies with volumes.

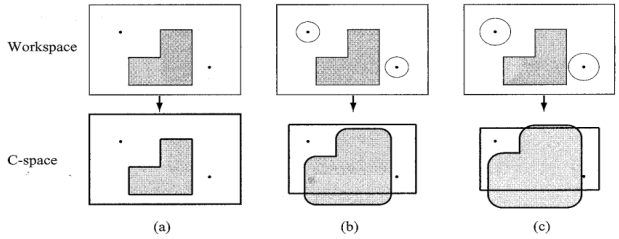


Figure 2. Obstacles Growing to Transform Workspace of Mobile Robots into C-Space
Taken from *Principles of Robot Motion* (Choset et al, 2005)

Some terminology in my highly-technical readings were worded with advanced verbiage and led to more questions. My mentor and other graduate students at Parasol thoroughly taught me key robotic vocabulary and now, I can define them in my own simple words. A **robot** is a rigid or articulated object that exists within the environment it is moving in. A robot's **environment** can be composed of free space, obstacles and less commonly, the border that encloses the area such as walls. A robot has a **holonomic constraint** when the state is only dependent on its *current* position, such as cleaning iRobots. A robot has a **nonholonomic constraint** when the state is only dependent on its *previous* position and times, such as cars. Nonholonomic robots may have **kinematic constraints** on the movement of its mechanical

components, such as its position, velocity, or acceleration. Below, I created a tree diagram to simplify how to decide if a robot is holonomic or non-holonomic.

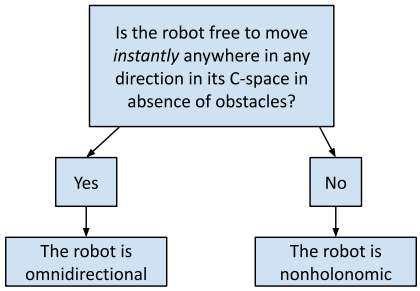


Figure 3. A Tree that Decides if a Robot has Nonholonomic Constraints

B. Experiments

I. PPL Checkout, Compilation and Execution

1. I checked out the PPL motion planning code, which means I made my own local copy of the Parasol GitLab repository.

2. I checked out the libraries that make up PPL utilities. A few main directories and files of PPL are Behaviors, ConfigurationSpace, Examples, MPLibrary, MPProblems, Test, Traits, Utilities, Workspace, Makefile, and main.cpp. The core directory is MPLibrary, which contains all our libraries used to solve motion planning problems, and includes open-source code for Connectors, DistanceMetrics, LocalPlanners, MapEvaluators, Metrics, MPStrategies, MPTools, Samplers, NeighborhoodFinders, ValidityCheckers, and more.

3. I checked out and built the vizmo visual tool on my Linux machine. A doctorate-candidate student, Irving Solis, taught me how to create an alias on the terminal in order to open the software in less than a second by typing “vizmo” anywhere in my local PPL directory instead of manually entering multiple directories to reach the software location.

4. I learned how to compile and run code in vizmo via crucial command such as “make clean/reallyclean/reallyreallyclean” and “make pmpl”. I also ran a CfgExamples.xml file with MP examples and observed its components and output roadmap (.map) and statistics (.stat) files. “Cfg” is shorthand for “Configuration”.I put the example roadmap output file, 3D environment file, and the 3D boxy robot in vizmo. I successfully used vizmo to create the roadmap below and learned how to navigate the software and how to change its settings. I also played with the system by varying the number of nodes, samples or connectors in the XML and visually observing how it changed the output files.

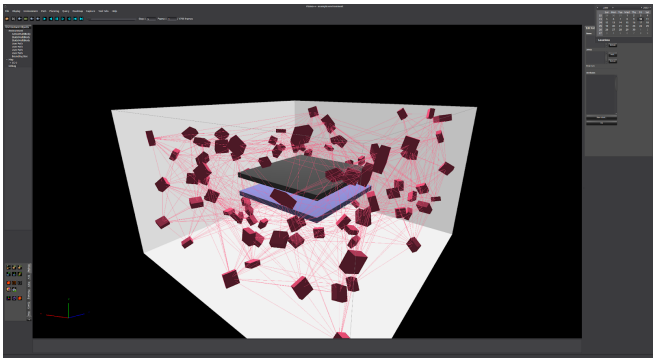


Figure 4. Module 1, Results of My First Experimentation with Vizmo Software

C. Quiz[1] *1*

1. Name five applications of motion planning:

1. Robotics, 2. Graphic Animation, 3. Medical Surgery
4. Virtual Prototyping, 5. Computational Biology.

2. Name four types of MP Algorithms

1. Probabilistic Roadmap (PRM), 2. RRT, 3. OBPRM (Nancy's algorithm), 4. Multi-Agent Motion Planning (MAMP).

3. Discuss in depth one of the applications from question one. What type of motion planning algorithm is used? How is the algorithm modified to cater to the application?

Motion planning is critical to developing high-quality, realistic graphic animation for video games. Gamers are less likely to reject video games where animated characters move in a realistic, natural path in a virtual plane with many obstacles. One main problem that motion planning can solve in graphic animation is when video game entities must move in groups. For example, in the Kingdom Hearts video games by Square Enix and Disney, the player-controlled character, Sora, navigates his path with two or more non-player controlled (NPC) characters, Donald and Goofy (shown below). All entities must find collision-free paths when traveling as a flock without deviating too far from each other or making unnatural movements like walking towards an enemy or moving in manners that aren't human or animal-like. This can lead to characters getting stuck or walking through obstacles instead of around them, which I witnessed multiple times.

[1] Graded: 27/28 (97%)

Figure 5. Kingdom Hearts Characters in Motion. Sora, Donald, Goofy, and Baymax are moving in a flock within the virtual free space without colliding into the obstacles or each other. In the roadmap at the top right corner, the flock of multiple characters is represented by the keychain icon, the free space is the blue area, and the obstacles are represented by white lines. (Source: Square Enix and Disney, 2019).

According to Overmars (2005), path planning in video games is usually computed using a combination of grid-based A* techniques, local methods for obstacle avoidance, flocking for group motion, combinations of scripted motion, and waypoints. Probabilistic Roadmaps are used to create virtual videogame roadmaps and query the paths navigated by the animated character entities. The paper summarizes how in the pre-processing phase of the PRM, the roadmap computes all possible motions for the animated character entity and is represented by a graph in which the nodes correspond to an entity's placements and the edges represent collision-free paths between those placements. During the game, the roadmap is used for answering path planning queries. Unfortunately, the PRM algorithm does not usually compute high quality roadmaps that can make multiple NPCs take long detours.

This motivated Overmars to propose a better way to create better roadmaps by creating nodes that lie far away from obstacles and then sampling near the Voroni diagram to increase the clearance in the configuration space, and then adding additional edges to create cycles in the roadmap graph to avoid long detours. Instead of only finding a straight line between two nodes that are collision free, their method also takes in circle arcs. Their roadmaps can be used in the query phase without the need for any post-processing. Basically, he still used PRM but they added his own techniques to the algorithm to create better roadmaps. The figure below illustrates his modified method.

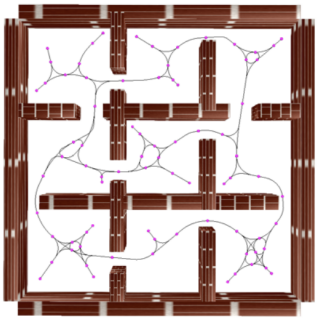


Figure 6. An example of a smooth roadmap completed in Overmars (2005)

4. What motivated the introduction of randomized planning algorithms such as PRMs and RRTs?

For motion planning problems to be successful, they have to run in exponential time. However, previous algorithms had difficulties solving larger problems because the cost was too high to pay. Problems with simple obstacles had complex C-obst, C-spaces had high dimensions, and we needed something that was practical in computing the C-free space of robots with many-DOFs. Computer scientists introduced randomized planning algorithms as a way to approximate motion planning solutions in a significantly more efficient complexity: polynomial time. Although PRMs and RRTs can't completely decide motion planning problems, they remain useful in more complex, increased dimensional C-space.

5. Name 2 features of MP problems that do not have full heuristic or algorithmic solutions. What properties of an environment or robots greatly increase the "hardness" of an MP problem?

Open problems in motion planning that do not yet have full heuristic or algorithm solutions includes (1) planning paths for robots moving in environments with narrow pathways between obstacles without colliding into them and (2) planning paths for multiple robot agents to move together at the same time without colliding with each other. If you combine those issues together, it would be really complex to solve a motion planning problem with multiple robots traveling along an environment with many narrow pathways and limited free space.

Robots traveling along a narrow passage was a major motion planning problem in the past. However, narrow passages increase the hardness of a motion planning problem, there are great algorithms that can solve this problem such as OBPRM (Amato, 1998). Another property that increases a motion planning problem is when a robot has many DOFs. For example, planning the motion of a nonholonomic walking robot that looks like a human and must be programmed to walk human-like.

6. Define degree of freedom. Name/describe as many types of DOF and/or joints for robotic systems as you can.

The degrees of freedom of a robot is the dimension of the configuration space. DOFs also indicate the minimum number of parameters needed to specify the configuration. Types of DOF include **translation**, **rotation**, **orientation**, **velocity**, and **time**. We can count the DOF of an open-chain jointed robot (serial mechanism) by adding the DOF at each joint. I don't think that all the numbers of DOF are always countable for all types of robot systems though. For example, a cloth is an example of a system with infinite DOFs. In a finite curb, you can count the DOFs. You can also count DOFs for closed-chain robots using Grübler's Formula. DOFs are more difficult to count for soft or deformable robots with significantly high counts of DOFs. For surgical needles, a kind of deformable robot, they have wires that change their orientation and length that are hard to model and count.

7. Define Configuration, C-Space, C-Free, C-Obst, and ∂C-Obst.

The **configuration** of a robot system is a complete specification of the position of every point in that system. The **C-space** of a robot system is the space of all possible configurations of the system. The **C-Free** is the set of feasible configurations at which the robot does not intersect any obstacles. The **C-Obst** is the set of non-feasible configurations at which the robot intersects an obstacle in the workspace. The **∂C-Obst** is is the set of points in C-space which have neighbors in both C-Obst and C-Free.

8. If a robot can move and turn in the xy plane and it has a mounted camera with a view radius of 60° that can look in any direction in the plane (not 'up' or 'down'), how many DOF does the robot have?

If the camera can look in any direction in the plane, it needs at least 2 DOF. In this case, the hypothetical robot has 3 DOF.

9. If a quadrotor is carrying a mobile arm attachment with 4 spherical joints, what is the dimension of its C-space?

The dimension is 18 DOF. A quadrotor has 6 DOF. A spherical join thas 3 DOF and the mobile arm attachment has 4 spherical joints, Since 3*4=12, the arm has 12 DOF. The final count is 6 + 12 = 18.

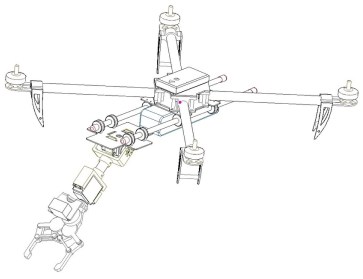


Figure 7. Quadrotor with Mobile Arm Attachment (Hernán, 2015)

10. How do we transform a configuration into a placement of the robot in work space? Work space placement to a configuration?

Transform: C-space -> Work Space

In C-Space, a particular configuration for the robot is always a single point. The single point has coordinates (floats) that represent the degrees of freedom for that robot, such as its position (x, y) or orientation (a float that represents degrees, e.g. 0.5 equals 90 degrees, 1.0 equals 180 degrees). Let's say that we have a robot that has 3 DOF and is in a 2D workspace. Its DOFs allow it to move along the x-axis, y-axis, and rotate. The robot's configuration is c = {5.0, 7.0, 0.5}. That means that in the workspace, the robot configuration point is located on the 5.0 coordinate in the x-axis, 7.0 coordinate in the y-axis, and is oriented at 90 degrees. This robot is also represented as a rectangle, where the rotation of its body is significant. Due to the configuration represented by {5.0, 7.0, 0.5}, we were able to transform the C-space into a placement of the robot in the workspace.

Transform: Work Space -> C-Space

In the workspace, this same configuration for the robot has physical geometry associated with it.The workspace is equal to the C-space and it can easily be transformed because the obstacles and robot maintain their geometry. We can also represent a robot's mass and radius by bordering the dot with a circle, which will end up taking more space and most-likely collide with the obstacle even if the robot's dot center does not. In this case, we will have to enlarge the obstacles in the workspace to account for the change in geometry of the robot. Consequently, the C-space can subtract the unfeasible space taken up by the enlarged obstacles to calculate the new free space, which will shrink. If the robot system's geometry is in collision with an obstacle, then the associated point in C-space lies in C-obst. However, if there is no collision in the workspace, the configuration lies in C-Free.

11. Explain how you would check if a configuration is in collision with an obstacle. Define other examples of "validity" for a robot.

One issue with collision checking is that we can rarely represent the entire C-Free or C-Obst, we can only approximate it. Therefore, it is important to rely on robot geometry and the workspace. In the C-space, each configuration for the robot is a single point. In the workspace, the same configuration has physical geometry. If this geometry is in collision with an obstacle in the workspace, then we can assume that the associated point is a configuration that lies in the C-Obst and a collision call can be made. If the geometry is not in the C-Obst, there is no collision in the workspace and it can be assumed that this configuration lies in the C-Free space and therefore is considered "valid". Generally, without referring to robot geometry and workspace, it is not easily feasible to tell if a configuration is in collision with an obstacle.

12. How do we geometrically compute the C-space of a robot in an environment? How can we account for this infeasibility?
We cannot geometrically compute C-space because it is too expensive. Instead of computing C-space, we have to sample it. Check if one configuration is valid or not. If it is not, select a random sample and test it. If we spend enough time sampling, we can cover the entire space and calculate the obstacles of C-space but there is no way of knowing if we have sampled enough. With enough configuration samples, we can approximate the C-space but not completely geometrically compute it. We really cannot know for sure. This motivates sampling-based methods.

Module 2: Probabilistic Roadmaps and Rapidly-Exploring Random Trees for Sampling-Based Motion Planning

A. Readings

For this module, I read a historic robotics journal paper titled *Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces* by Kavraki et al (1996). The **Probabilistic Roadmap (PRM) algorithm** was designed to solve robot motion planning problems for holonomic robots in static workspaces. The PRM consists of the learning phase — where the roadmap construction and expansion steps occur — and the query phase. My procedural analysis of the PRM algorithm can be found in the Module 2 Quiz 2 section. This paper concluded the PRM can be applied to virtually any type of holonomic robot and can be easily customized to run more efficiently on a large array of problems.

The experiments in Kavraki et al (1996) tested their customized PRM method as well as general path planning methods like the Randomized Path Planner (RPP). The results show that their PRM method can efficiently solve problems for planar articulated robots with many DOFs much more consistent than the RPP. Although the RPP can be very fast on some difficult problems, it may take restricting time on some others. Such a disparity was not observed in the PRM, since it answered queries considerably faster than RRP, usually in a matter of a few dozen seconds. The paper ended with open ended questions such as how the PRM could be applied to more complex two or three dimensional geometries, since the cost of collision checking would be much higher. Another question was how to extend the PRM method to dynamic schemes, where objects can be removed or added and environments are subject to incremental changes, which was a major motion planning problem in the mid-1990s.

I read another historic research paper titled *Rapidly-Exploring Random Trees: A New Tool for Path Planning* by Steve M. LaValle (1998). The **Rapidly-Exploring Random Tree (RRT) algorithm** returns a randomized data structure that was designed for a broad class of motion planning problems and was specifically designed to handle nonholonomic constraints and high degrees of freedom. The RRT, can be applied to robotic motion planning with holonomic, nonholonomic, and kinodynamic constraints. A **kinodynamic constraint** is a restriction on the flexibility of the robot system without any obstacle present. For example, if a robot can rotate but only up to 30 degrees left or right, it cannot position itself to 180 degrees to move toward the space directly behind itself due to its kinodynamic constraint.

The RRT algorithm is programmed to search in a metric space X for a continuous path from an initial configuration x_{init} to a goal region $X_{goal} \subset$ (i.e. is a subset of) X or a goal configuration x_{goal} where C is the configuration space of a rigid body or system of bodies and $X = C$. The RRT avoids with the obstacle region $X_{obs} \subset X$, which may contain states that correspond to velocity bounds that are configurations at which a robot is in collision with an obstacle in the world. An explicit representation of X_{obs} is not available, therefore the RRT can only check whether a given configuration lies in X_{obs}. When an RRT is constructed, its vertices are configurations in X_{free}, the complement of X_{obs}, and all its edges correspond to a path that lies completely in the X_{free}.

Non-holonomic constraints are expressed by a **configuration transition equation** of the form $x = f(x, u)$. Vector u is selected from the set U of inputs and vector x denotes the derivative of the configuration with respect to time. When f gets integrated over a fixed time interval Δt, the next configuration, x_{new} can be determined. For nonholonomic robots, x_{new} has a constraint due to the choice of f. For holonomic robots, x_{new} can move the robot system in any direction relative to x. The configuration transition equation for holonomic robots is express in the form of $f(x, u) = u$ *and* $||u|| \leq 1$, *which means* any bounded velocity can be achieved.

```
GENERATE_RRT(x_init, K, Δt)
  T.init(x_init);
  for k = 1 to K do
      x_rand ← RANDOM_STATE();
      x_near ← NEAREST_NEIGHBOR(x_rand, T);
      u ← SELECT_INPUT(x_rand, x_near);
      x_new ← NEW_STATE(x_near, u, Δt);
      T.add_vertex(x_new);
      T.add_edge(x_near, x_new, u);
  Return T
```

Figure 8. RRT Algorithm Pseudocode (LaValle, 1998)

Let p denote a distance metric on the C-space.
(1) The first vertex of tree T is $x_{init} \in$ (i.e. is an element of) X_{free}.
(2) For each iteration of the k-nearest neighbor value:
(3) Instantiate random configuration x_{rand}.
(4) Find x_{new}, the closest vertex to x_{rand} in terms of d.

(5) Select an input $u \in U$ that minimizes the distance from x_{near} and x_{rand}. In this step, collision detection can be performed by an incremental method, such as the Voronoi-Clip (Mirtich, 1997).
(6) NEW_STATE is called on each input u to evaluate a potential new configuration x_{new}. If U is not finite, it can be discretized or use an alternative optimization procedure. In this step, the new configuration was obtained by applying u.
(7) Add x_{new} as a vertex to *T.*
(8) Add an edge from x_{near} *to* x_{new}. Input u is recorded with the edge because this input must be applied to reach x_{near} *from* x_{new}.
(9) Iteration ends once X_{goal} or x_{goal} is reached. Return RRT tree *T.*

Below is an example to better illustrate the algorithm. The figure below shows the construction of an RRT for the case of X= [0,100] x [0,100], Δt=1, and x_{init} = (50,50). Assume that a holonomic model is used where $f(x, u) = u$, which implies that $f = u$ and $U = \{u \in \mathbb{R}^2 \mid ||u|| \leq 1\}$.

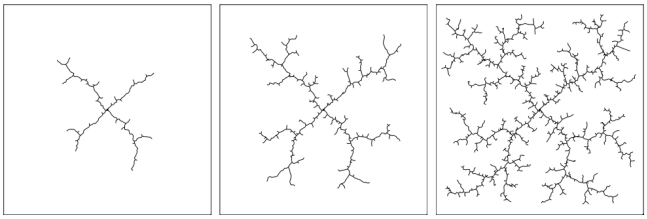


Figure 9. Example of RRT Construction. An RRT is constructed in a bounded, convex region *X* in a plane. The RRT quickly expands in a few directions to explore the four corners of the square (LaValle, 1998).

In his paper, LaValle described in detail the many differences between PRMs and RRTs. In the Module 2 Experiments section, I visualized his comparisons in a chart (Figure 26). My chart describes RRT biases that can explain how the RRT expansion in Figure 9 isn't as "random" as it might seem to be.

B. Experiments

I. Experimenting with PRMs

I experimented with PRMs to discover how the number of samples and edges per vertex affect the quality and connectivity of a roadmap in two distinct environments: Topo, without narrow passage, and Narrow, with a narrow passage.

Topo Environment

Topo has six obstacles spaced out from each other. The PRM baseline had 25 samples and 5 edges per vertex and produced 27 nodes and 120 edges in 4.11 seconds, which produced a weakly connected roadmap due to the very few nodes and edges produced. I will experiment with PRM values when (A) samples are increased to 100, (B) edges are increased to 10, and (C) samples are increased to 100 and edges increased to 10.

Hypothesis: Larger samples and edges metrics in a PRM planner will generate more nodes and edges. The roadmap connectivity will improve but the planning time will decrease.

Topo Data

Samples (number) and Edges on each vertex (k)	Nodes	Edges	Planning Time (seconds)
number=25 and k=5	27	118	2.1
number=100 and k=5	102	576	4.79
number=25 and k=10	27	172	4.1
number=100 and k=10	102	1,102	13.2

Figure 10. Topo PRM Data. Shows the generated nodes, edges, and total planning time results for four PRM planners tested in *Topo* with varying *number* a *k* values.

Topo Results

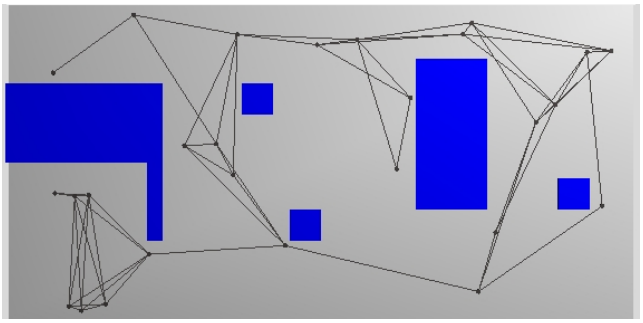


Figure 11. Topo PRM Baseline, where number=25 Samples and k=5.

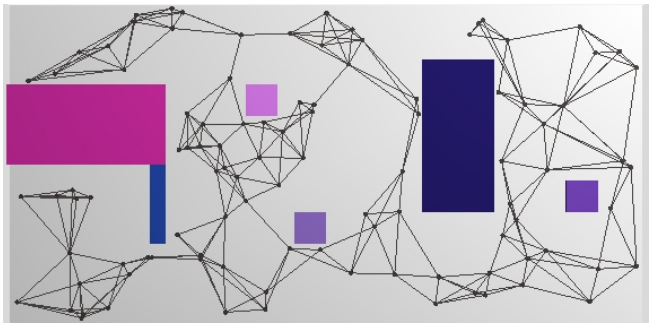


Figure 12. *Topo* PRM Experiment 1 Results, where number=100 and k=5.

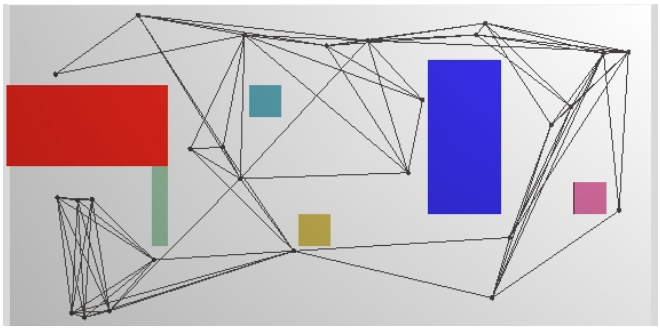


Figure 13. *Topo* PRM Experiment 2 Results, where number=25 and k=10.

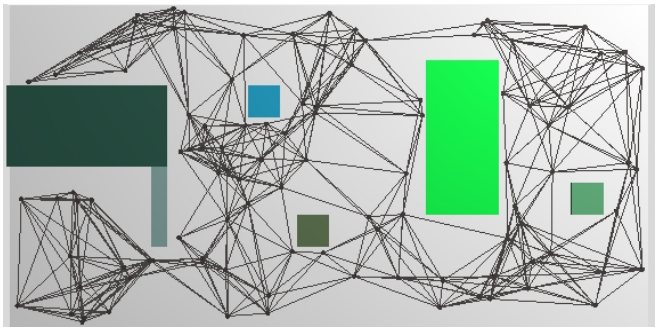


Figure 14. *Topo* PRM Experiment 3 Results, where number=100 and k=5.

Narrow Environment

The *Narrow* PRM baseline had 25 samples and 5 edges per vertex and produced 1,202 nodes and 11,562 edges in 33.7 seconds, which produced a strong, very well-connected roadmap due to the 1,000+ nodes and edges produced. I will experiment with PRM values when (A) samples are increased to 100, (B) edges are increased to 10, and (C) samples are increased to 100 and edges increased to 10.

Hypothesis: As samples and edges per vertex increase, the number of edges will decrease because the narrow passage created by the obstacles will affect the connectivity in the C-free space surrounding the obstacles, thus planning time will be faster.

Narrow Data

Samples (number) and Edges on each vertex (k)	Nodes	Edges	Planning Time (seconds)
number=25 and k=5	1,202	11,562	33.7
number=100 and k=5	1,202	10,792	31
number=25 and k=10	1,202	23,032	82.4
number=100 and k=10	1,202	21,398	68.9

Figure 15. Narrow PRM Data. Shows the generated nodes, edges, and total planning time results for four PRM planners tested in *Narrow* with varying *number* a *k* values.

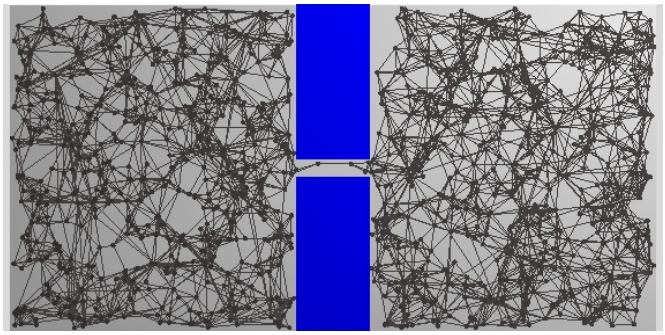


Figure 16. *Narrow* PRM Baseline Results, where number=25 and k=5.

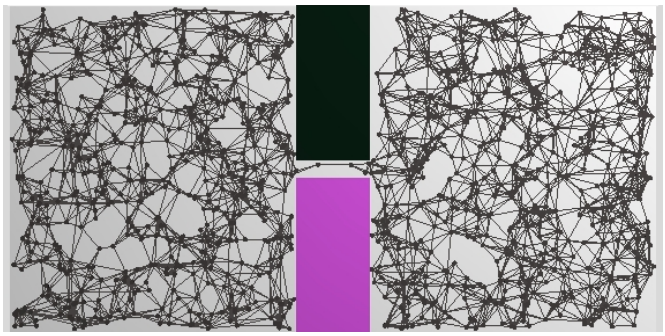


Figure 17. *Narrow* PRM Experiment 1 Results, where number=100 and k=5.

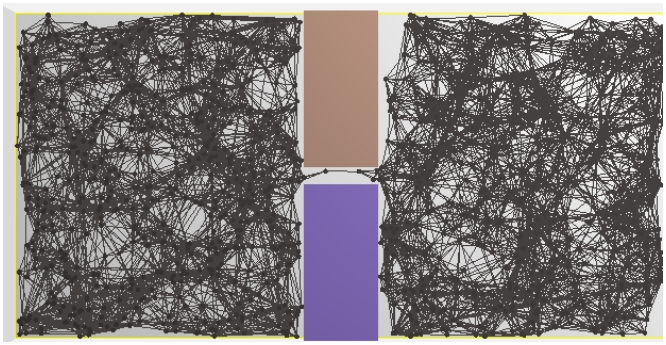


Figure 18. *Narrow* PRM Experiment 2 Results, where number=25 and k=10.

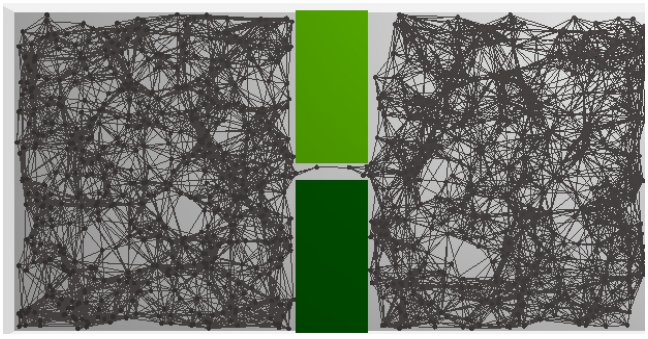


Figure 19. *Narrow* PRM Experiment 3 Results, where number=100 and k=10.

Results

My hypothesis on *Topo* was partially correct. My results showed that as samples and edges increase, more nodes and edges are generated and the images do show a more connected roadmap, which is what I hypothesized. However, whereas I expected faster times, the planning time increased. The less abstract the roadmap connectivity is, the more time it takes to plan it because there are less nodes and edges to generate and plot in the environment. My hypothesis on *Narrow* was incorrect. My results show that when there is a narrow passage in the environment, significantly more nodes and edges are produced by the PRM planner compared to environments with no narrow passage. More planning time is required to generate more edges.

The roadmap's performance in the *Topo* and *Narrow* environments are mainly different but there is one similarity. Both *Topo* and *Narrow* have roadmaps that seem to have been plotted randomly because there are vertices and edges that are on the obstacle or pass through it even if following such a path leads to colliding with that obstacle. This might be due to the fact that it is computationally expensive to deal with collision detection (CD). However, in Topo, some paths can take you from completely different sides of the environment without colliding with any obstacles. On the other hand, *Narrow* does not have a solution; no paths in the roadmap can successfully pass through the narrow passage between the two obstacles without colliding.

There were some questions that arose during the *Narrow* experimentation process. I was unsure why in vizmo, the *Narrow* roadmaps had the exact same number of nodes for all four of its experiments. How could 25 samples with 5 edges per vertex generate the same amount of nodes as 100 samples with 10 edges per vertex? Another question is why the PRM set to number=100 and k=10 produced less edges than the PRM set to number=25 and k=10. I expected the number of nodes and edges to also increase as the samples and/or edges increased, but this part of my hypothesis was wrong. I re-ran the *Narrow* experiments twice to ensure consistent results, but each run returned the same data.

I found it strange that in *Narrow*, the nodes and edges split in two entirely different regions that were separated by the two objects in the middle. There was a cluster of nodes and edges on the left side, and another cluster on the right. There was no connectivity between the two clusters above or below the obstacles. It is as if the narrow pathway obstructed the visibility of the planning performance, and it was not possible to create any other solution in the environment's free space. This is distinct from *Topo*, which has no narrow pathway and therefore has better visibility during planning performance. In *Topo*, there are edges and nodes all around most of the obstacles, and there is better connectivity in the roadmap.

Conclusion

It was interesting to learn how to build a roadmap graph over the C-space and how important sampling is to approximate C-Free. It was also insightful to see how varying parameter values of number samples and *k* edges per vertex defined the quality and connectedness of a roadmap.

II. Experimenting with RRTs

In this experiment, I explored and approximated the C-space using one of the most fundamental algorithms in sampling-based motion planning, the Rapidly-exploring Random Tree (RRT). I used the same ProblemDefinition.xml file and *Topo* and *Narrow* environments that I used in my previous PRM experiment.

Topo Data

Samples (number) and Edges on each vertex (k)	Nodes	Edges	Planning Time (seconds)
k=5	103	102	0.756
k-10	103	102	0.793

Figure 20. Topo RRT Data. Shows the generated nodes, edges, and total planning time results for two RRT planners tested in *Topo* with varying *k* values.

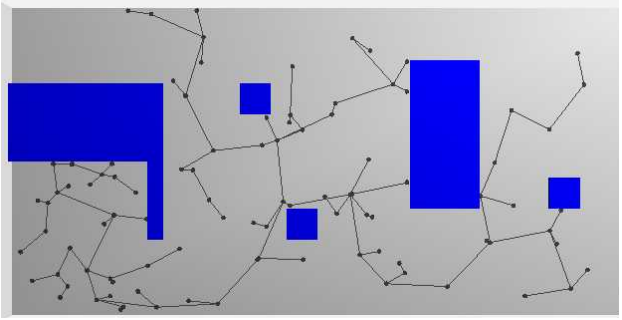


Figure 21. *Topo* RRT Experiment Results, where k=5 or k=10.

Narrow Data

Samples (number) and Edges on each vertex (k)	Nodes	Edges	Planning Time (seconds)
k=5	705	704	3.51
k=10	705	704	3.54

Figure 22. *Narrow* RRT Data. Shows the generated nodes, edges, and total planning time results for two RRT planners tested in *Narrow* with varying *k* values.

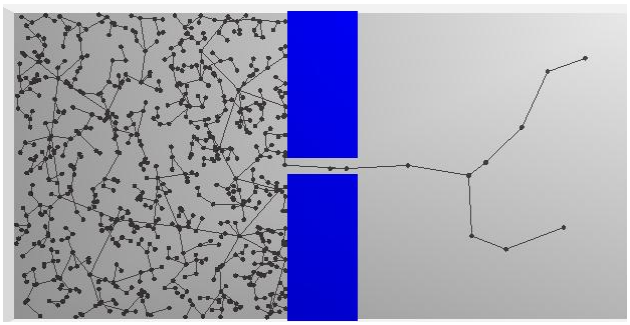


Figure 23. *Narrow* RRT Experiment Results, where k=5 or k=10.

Topo Discussion Questions and Results

1. What do you observe?

I observed a data structure that looks like a tree with a bunch of interconnected branches (edges) and leaves (nodes). This sampling method looks quite natural. There is a fascinating synchronicity between how the RRT traverses *Narrow* with obstacles and how ivy vines grow on walls with windows.

Figure 24. Ivy Vine Growing on Wall and Avoiding Windows (Home & Garden, 2022)

I also observed that *Narrow* looks like the exact same environment that LaValle experimented in with his RRT method in. He characterized his narrow environment as a "tightly constrained 3D holonomic planning problem" (left).

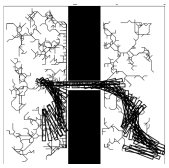


Figure 25. RRT Problem (LaValle, 1998)

2. Is the RRT performance similar in the two environments? If not, what do you think is different?

No, the RRT performed differently in each environment. In *Topo*, the tree loosely branches out in the C-space and grows around the 6 obstacles. There is very little sampling done in the C-Free in *Topo* compared to the very densely sampled C-Free in *Narrow*. The tree works very well and fast in the cluttered *Topo* environment. The RRT does not perform as well in *Narrow* but it does work much harder at sampling the C-Free until it finally finds a path through the narrow passage and eventually stops. One thing I observed is that it's difficult to find the goal configuration in *Topo* but was very easy to spot in *Narrow.*

3. Describe a few differences between PRM and RRT.

Probabilistic Roadmaps	Rapidly-Exploring Random Trees
* best for holonomic robots with many-DOF and higher dimensional C-spaces * not good for nonholonomic robots in connectivity phase * best for static path planning	* best for robots with nonholonomic constraints * also good for many-DOF holonomic robots * can be applied to kinodynamic path planning
* requires the connections of thousands of configurations to find solutions * often suffers because many extra edges are generated in attempts to form a connected roadmap * graph grows through point-to-point convergence	* does not require any connections to be made between pairs of configurations * always connected even though number of edges is minimal * tree iteratively expands by driving system slightly towards randomly-selected points
* biased towards sampling the free C-space and chooses random nodes to connect but has no particular focus on generating a path	* biased towards unexplored larger spaces at each step * vertices with large Voronoi regions are more likely to be selected for expansion
* probabilistically complete but is challenged by narrow passages * random sampling can be inconsistent but it can use uniform sampling methods	* probabilistically complete but is challenged by convergence issues * the distribution of vertices in RRT eventually become more uniformly distributed
* requires more expensive k-nearest neighbor queries	* requires single nearest-neighbor queries
* two-part algorithm with learning and query phases	* simple and easy to implement
* incremental collision detection can become very computationally expensive	* best suited for incremental collision detection to allow the fastest-available collision detection algorithms to be applied for each collision check

Figure 26. Comparisons Between the PRM and RRT (Source: LaValle, 1998)

4. Which method do you believe performed better?

The RRT performed significantly better than the RRT in solving the narrow passage in *Narrow.* However, the PRM was better at sampling the free C-space in both environments. Ther RRT only sampled the C-Free on the left side of *Narrow* but the tree stopped growing after it passed through the narrow passage and reached the goal configuration. Although the RRT generated a faster, more efficient path through the narrow passage between the two obstacles in *Narrow*, two nodes leading to that path were in ∂C-Obst as can be seen in Figure # (right), which is not a bad

thing. The PRM surprisingly found a path within the narrow passage as seen in Figure # (left), but at an expensive cost.

Figure 27. PRM Solving Narrow Passage In *Narrow* Environment

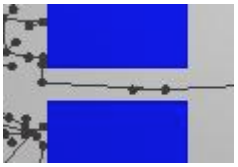


Figure 28. RRT Solving Narrow Passage In *Narrow* Environment

The RRT planner also performed best in *Topo* since it was able to traverse the free C-space around all six obstacles, without colliding into them, in less than one second despite using the same k-values as the PRM planner. I was impressed by the RRT results. I overall recommend the RRT best for environments with cluttered spaces and narrow paths.

5. What are the pros and cons of each method?

The PRM is better at sampling and representing the free space. The RRT is faster at generating valid paths around obstacles in cluttered or narrow environments without requiring expensive k-nearest neighbor queries. The RRT also works better for planning with nonholonomic and kinodynamic constraints. The PRM is not good at planning with non-holonomic robots but is excellent at planning with holonomic robots with many-DOF. One similar pro that PRMs and RRTs have is that both are designed with as few heuristics and arbitrary parameters as possible, which helps lead to better performance analysis, more behavioral consistency, and facilitates the adaption of the methods to related applications (LaValle, 1998). For further insight, I provide more pros and cons for the PRM and RRT in the next section.

C. Feedback

This was a great module. The readings were fascinating and moderately simple to follow, the interactive experiments were fun and easy due to the detailed assignment instructions, and the discussion questions were well-developed to extract the main points of the module. I learned so much about PRMs and RRTs.

D. Quiz[2]2

1. Summarize the PRM algorithm. Use pseudocode in your explanation.

The Probabilistic Roadmap (PRM) algorithm is critical to motion planning because it builds a practical, well-connected graph for a robot to travel through a path in the C-free space of a C-space. The PRM method has two phases: a learning phase and a query phase.

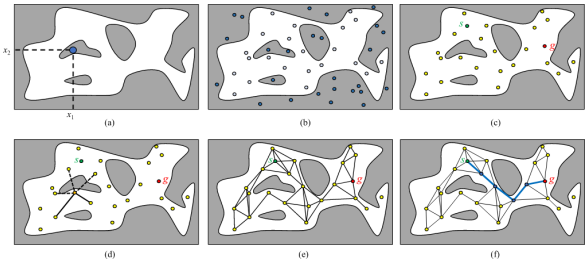


Figure 29. Main Steps of the PRM (Source: https://motion.cs.illinois.edu/RoboticSystems/)

The **learning phase** has two steps: the construction step and the expansion step. The first step constructs and stores a probabilistic roadmap as an undirected graph R(N, E) where the nodes in N represent collision-free configurations in the C-space and the edges in E correspond to feasible paths between these configurations. My explanation of the PRM algorithm follows.

```
(1)   N ← ∅
(2)   E ← ∅
(3)   loop
(4)       c ← a randomly chosen free configuration
(5)       N_c ← a set of candidate neighbors of c chosen from N
(6)       N ← N ∪ {c}
(7)       forall n ∈ N_c, in order of increasing D(c, n) do
(8)           if ¬same_connected_component(c, n) ∧ Δ(c, n) then
(9)               E ← E ∪ {(c, n)}
(10)              update R's connected components
```

Figure 30. PRM Algorithm Pseudocode (Kavraki, 1996)

In the **construction step**, the graph R=(N,E) is initially empty, (1) node set **N** is empty and (2) edge set **E** is empty. Then, (3) in a loop, (4) a random free configuration **c** is generated and (6) added to node set **N**. For each new random free node **c**, (5) we select a number of nodes $\mathbf{N_c}$ of candidate neighbors from the current node set **N** and try to connect random free node **c** to the selected $\mathbf{N_c}$ nodes using the local planner. (7) For all **n** nodes in $\mathbf{N_c}$ chosen in order of increasing distance from **c**, the local planner tries to see if the pair of components **c** and **n** are connected. (8) **If** the local planner succeeds in computing a feasible patch between **c** and the selected node **n**, (9) then the **edge(c, n)** is added to edge set **E**. Finally, (10) graph R's connected components are dynamically updated with the newest values of node set **N** and edge set **E**.

[2] Graded: 18/18 (100%)

The **expansion step** improves the connectivity of the graph R generated by the construction step. Here, the graph is based on the difficult regions that the nodes might be in. When expanding configuration **c**, the procedure selects a new free configuration in the neighborhood of **c**, adds this configuration to **N**, and tries to connect it to other nodes of **N** in the same way that it was done in the construction step. This expansion increases the density of roadmap configurations in the difficult C-free regions.

In the **query phase**, the PRM chooses a random start configuration **s** and goal configuration **g** in the C-free space and adds/connects them to the roadmap created in the learning phase. Then, the PRM checks if it can find an edge between them. Also in this phase, a path is found in the roadmap between connected nodes.

2. What is the complexity of roadmap construction for *n* nodes? What is the complexity of querying a roadmap of *n* nodes?
For the roadmap construction of **n** samples, it takes **O(n)** time to sample but $O(n^2)$ time for local planner attempts in the worst case. It takes O(n) time to add a start or end goal node to the roadmap. The time to add start and end nodes gets added to the query complexity, $O(n^2 + n \log n) + O(n) + O(n)$ but then we drop the lower order terms. The complexity of querying the roadmap is $\mathbf{O(n^2 + n \log n)}$. The worst case, in fully connected graphs, is $O(n^2)$.

3. Summarize the experiments performed in the PRM journal paper, Kavraki et al (1996).
In Kavraki et al (1996), all experiments are 2-dimensional. The parameters included two scenes. Each scene contains polygonal obstacles and a planar articulated robot whose linkages are line segments, which does not limit the methods used. The first scene has a 4-dof robot with three resolute joints and one prismatic joint. The second scene has a 5-resolute joint robot and narrow areas in the workspace.

The experiments basically customizes the PRM method to planar articulated robots and experiments with both a general and a customized implementation of the PRM. The customized implementation was able to solve very difficult path planning queries involving many-dof robots in a fraction of a seconds after a learning time of a couple dozen seconds. The general implementation was much slower and was able to solve the same problems in several minutes but it was still relatively efficient in less difficult problems. The results showed that the general implementation is able to efficiently solve complex planning problems, but when applied to problems involving more DOFs, the learning times required to build an adequate roadmap is significantly longer (e.g. about 25 minutes for one of their tests).

4. What are the major strengths and weaknesses of PRMs?
One major strength of the PRM algorithm is versatility, as it can be applied to virtually any type of holonomic robots as well as nonholonomic robots. A PRM is also reusable since it can run many distinct queries. Another strength is that it works fast. For example, roadmaps can be constructed in seconds for easier problems and queries can be processed in a fraction of a second (Latombe et al, 1996). These fast, small query times makes it possible for the PRM to work with robots with many DOFs performing several point-to-point motions in workspaces with static obstacles.

Another major strength is that the PRM learning and query phase is not limited to be executed sequentially. Each phase can be "interwoven" with each other to adapt the size of the already-created roadmap to address difficulties encountered during the query phase. I think it's beneficial that the PRM isn't a one-shot method. One other major strength is that the local paths are not required to be memorized (unless a non-deterministic local planner is used) during the learning phase because recomputing them during the query time is inexpensive.

A PRM weakness is sampling in difficult regions such as narrow passages. Generally, in sampling-based planning, difficult regions are hard to sample in, which makes PRMs unlikely to capture the connectivity of C-Space in those regions. An example that a laboratory doctorate student gave me was that in a 1-D narrow passage with no volume, a sampling-based method like the PRM will never be able to solve a narrow passage problem.

One major PRM weakness is that it is expensive, tedious and time-consuming to sample for collisions using PRM. An additional major weakness is that it is also not possible to know how many samples we need to show connectivity of all the C-free space. The PRM never guarantees to fully represent free C-space connectivity. If the time spent on roadmap construction is short, PRM may not construct an adequate roadmap. Consequently, it can be difficult to certainly know if the PRM algorithm is failing to find a path because one does not exist or because it needs more samples.

5. How might you address the difficulties of PRMs?
One of the difficulties of PRMs is solving narrow passages. One way we can address this problem is by strategically designing better sampling methods. It is critical for a roadmap to adequately capture the connectivity of the C-free space, especially in "difficult" regions that may contain narrow passages. If the roadmap connectivity is not adequate, the query will frequently fail. One thing we can do is spend more time in the learning phase or simply take advantage of the PRM's ability to interweave the learning and query phases so that you can go back and sample more random configurations in the construction step.

The OBPRM, a sophisticated sampling method, was introduced in Amato et al (1998) to tackle the narrow passage problem. Amato's poster below stated the PRM method did not work well at sampling the C-Free space in the narrow passage. The OBPRM algorithm was able to find a path between the narrow passages by enhancing the PRM sampling method. This algorithm is very intuitive, and Nancy did an excellent job at tackling one of the most challenging PRM problems: the narrow passage.

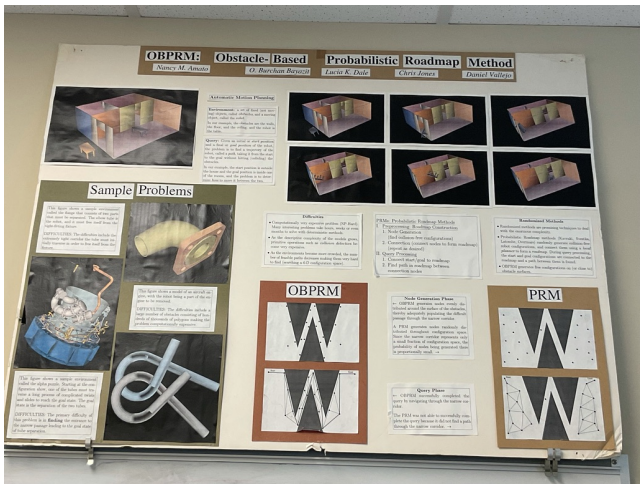


Figure 31. OBPRM Poster by Amato et al (1998) Displayed at Parasol Laboratory

If I had expert knowledge in motion planning, I would create an algorithm that mimics when a person (many-DOF system) wakes up in the middle of the night (with the lights off) in a dark messy room and tries to find their path to the bathroom in a cluttered environment by touching the walls and obstacles around them each time before taking a step. This way, I could create a path from the person's bed to the bathroom by relying on touching the surface of the obstacles instead of relying on explored free space. Since it's very dark, we can't assume there is free space because we cannot see it.

My algorithm would assume no free space exists at first; if you touch obstacles, you can assume the place you're currently in is free, then use your arms to feel more obstacles and if you feel any obstacles, avoid the direction your arm is pointing to. If there are no obstacles at arms-length, move in any direction and stop until you encounter another obstacle, repeat the process until the end goal is reached. Also, if we could build an algorithm that mimics how "blind" people walk about in the world, it would be very fascinating to me. This idea was inspired by my conversation with Diane Uwacu, who intuitively described OBPRM as a robot navigating a narrow passage similar to how humans experience walking in a dark room.

6. In what situation would you prefer RRT over PRM?

I prefer RRT over PRM for motion planning problems where (1) the environment is cluttered with many obstacles, no narrow passages, and less free space and (2) the robot system is nonholonomic and has kinodynamic constraints such as velocity or position. A real life example of this would be finding a path from start to finish with a car that is constrained to only move forward and left (cannot move backward or right). In such a case, the PRM would not do well because it would have difficulty finding the connectivity between generated vertices during the query phase. PRMs do better in environments with more C-free space where there is more connectivity and where robot systems are holonomic and don't have velocity constraints. PRMs require more expensive k-nearest neighbor queries, which is ideal for multi-query problems. On the other hand, RRTs require single nearest-neighbor queries that are less expensive.

Module 3: Visualize Motion Planning Problems with Vizmo

A. Experiments

I. Experiment 1: Testing Maze Benchmark

The benchmark I tested was Maze. I used Vizmo software to visualize a **probabilistic roadmap** on the maze environment. This test was run on the Mate Linux operating system.

Experimental Set-Up

An important portion of this experiment was to get familiar with the code found in core MP components. I broke down the files in the benchmark and visualized the code in the chart below.

maze-vizmo.env	maze.env	maze.robot	robot.g	maze.query
Boundary Box2D [-38.5:38.8 ; -35:35] Multibodies 4 Active 1 robot.g -c(0.5 0.5 0.5 1) Planar Translational Connections 0 Passive block1.g -c(0.15 0.15 0.15 1) -12.5 0 0 90 0 0 Passive block2.g -c(0.15 0.15 0.15 1) 3.25 4 0 90 0 0 Passive block3.g -c(0.15 0.15 0.15 1) 12.5 1 0 90 0 0	Boundary Box2D [-38.5:38.8 ; -35:35] Multibodies 3 Passive block1.g -c(0.15 0.15 0.15 1) -12.5 0 0 90 0 0 Passive block2.g -c(0.15 0.15 0.15 1) 3.25 4 0 90 0 0 Passive block3.g -c(0.15 0.15 0.15 1) 12.5 1 0 90 0 0	1 robot.g -c(0.5 0.5 0.5 1) Planar Translational Connections 0	1 8 12 36 1 12 0.5 0.5 0.5 0.5 0.5 -0.5 0.5 -0.5 0.5 0.5 -0.5 -0.5 -0.5 0.5 0.5 -0.5 0.5 -0.5 -0.5 -0.5 0.5 -0.5 -0.5 -0.5 1 5 -7 1 7 -3 1 2 -6 1 6 -5 1 3 -4 1 4 -2 5 6 -8 5 8 -7 2 4 -8 2 8 -6 3 7 -8 3 8 -4	0 0 34 0 0 -34

Figure 32. A chart that illustrates the code in the five core files of the Maze Benchmark (Parasol Planning Library, 2022). Maze benchmark has an environment file with obstacles (maze.env), an environment file specifically for visualizing in vizmo (maze-vizmo.env), robot files (maze.robot, robot,g), and a query file (maze.query).

Evaluation

1. I changed the highlighted CfgExamples.xml code (below) to match the files of the Maze example that I tested.

```
<!-- Here we specify input files, such as environment and query. The
     'baseFilename in the Problem node is for Simulator stat output only. -->
<Problem baseFilename="maze">

  <Environment filename="Maze/maze.env"
    frictionCoefficient="0" gravity="0 0 0"/>

  <Robot label="maze" filename="Maze/maze.robot">
    <!-- <Agent type="planning"/>-->
    <Agent type="pathfollowing" waypointDm="euclideanPosition"
      waypointThreshold=".05"/>
  </Robot>

  <Task label="query" robot="maze">
    <StartConstraints>
      <!-- WARNING Support for multiple constraints is not yet implemented!-->
      <CSpaceConstraint point="0 34"/>
    </StartConstraints>

    <GoalConstraints>
      <!-- Here multiple constraints indicate a compound goal. Each constraint
           represents a single intermediate goal. -->
      <CSpaceConstraint point="0 -34"/>
    </GoalConstraints>
  </Task>

</Problem>
```

2. I ran PMPL in src: `> ./pmpl -f Examples/CfgExamples.xml`

Results

Running PMPL with the CfgExamples.xml produced a file titled example.12345678.query.map that represents the probabilistic roadmap in the maze environment. In Vizmo, I uploaded the maze-vizmo.env file to visualize the maze environment as well as the example.12345678.query.map file to visualize the roadmap. The roadmap is an undirected graph R composed of nodes and edges. The configurations are the nodes of R and the paths computed by the local planner are the edges of R. The resulting vizmo representation of the benchmark I tested is below. The roadmap has four connected, each shown by a different color.

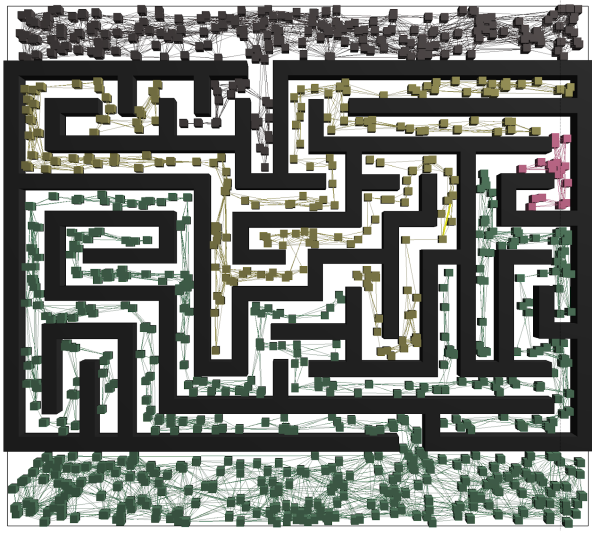


Figure 33. Maze Benchmark Test Results

II. Experiment 2: Varying the Sampling Techniques for PRMs

I will be experimenting with Uniform Random Free, Gaussian, Obstacle-Based Probabilistic Roadmap (OBPRM), Medial Axis Probabilistic Roadmap (MAPRM), and Bridge Test algorithms for sampling.

Running Experiments

First, I created a copy of the CfgExamples.xml file used in Module 3 Part 1 and renamed it MyCfgExamples.xml. This allows me to experiment with the code freely without changing the original file that contains the shared source code. SInce I am no longer using the maze benchmark, I will have to change my code to specify that the 3D environment and the 3D boxy robot will be used and change the labels accordingly.

```
<Problem baseFilename="3d">
  <Environment filename="3D/3d.env"
    frictionCoefficient="0" gravity="0 0 0"/>
  <Robot label="boxy" filename="3D/boxy.robot">
    <!--<Agent type="planning"/>-->
    <Agent type="pathfollowing"
     waypointDm="euclideanPosition"
     waypointThreshold=".05"/>
  </Robot>
  <Task label="query" robot="boxy">
    <StartConstraints>
```

Since boxy.robot has more DOF than the maze.robot, I also have to make changes to the start and goal <CSpaceConstraint> nodes.

```
      <!-- WARNING Support for multiple constraints is
        not yet implemented! -->
      <CSpaceConstraint point="0 0 0 0 0 0"/>
    </StartConstraints>

    <GoalConstraints>
      <!-- Here multiple constraints indicates a
compound
       goal. Each constraint represents a single
       intermediate goal. -->
      <CSpaceConstraint point="20 5 10 0.2 0.8 0.5"/>
    </GoalConstraints>
  </Task>
```

Experimental Setup

Before starting my experiments, I searched for the <ConditionalEvaluator> node for the NodesEval label and changed the value from 1000 to 1. This ensures that at least 1 node exists before terminating for each sampling experiment.

```
<ConditionalEvaluator label="NodesEval"
metric_method="NumNodes" value="1" operator=">="/>
```

Uniform Random Free Sampler

1. In the <MPStrategies> section, I set the <Sampler> label as "UniformRandomFree".

2. I commented out all the Connector nodes. The connectors have been commented out because for now, we will only be evaluating the samples, which are represented by nodes, so we do not need the connectors, which are represented by edges. I also ensured that the <Evaluator> node is set to "NodesEval."
3. I returned to the PRM-build section and changed the UniformRandomFree number from 1 to 100. This states that when sampling, 100 samples are generated with 1 attempt per sample. In other words, we can generate anywhere from 1 to 100 samples.

The code for this experiment should look like this after Steps 1-4.

```
<MPStrategies>
   <!-- Basic PRM where num samples is based on Number
-->
     <BasicPRM label="BasicPRM1" debug="true"
querySampler="UniformRandomFree">
       <Sampler label="UniformRandomFree" number="100"
attempts="1"/>
       <!--Connector label="Closest"/-->
       <!--Connector label="ConnectCCs"/-->
       <Evaluator label="NodesEval"/>
     </BasicPRM>
```

Note that the Sampler label for the BasicPRM node is "UniformRandomFree", so any experiment run with this xml code will be for uniform sampling. I ran PMPL by writing the commands "make pmpl" and " ../pmpl -f MyCfgExamples.xml" to generate a .map and .stat file. In doing so, I completed uniform sampling. Therefore, I renamed those files to Uniform.map and Uniform.stat.

Obstacle-Based PRM (OBPRM) Sampler

4. The same steps were followed for this sampler, except under the <MPStategies> section, the Sampler node's label was set to OBPRM. Then, I ran pmpl and completed OBPRM sampling. I renamed the output files to OBPRM.map, OBPRM.stat, and OBPRM.vd.

```
<!-- Basic PRM where num samples is based on Number -->
     <BasicPRM label="BasicPRM1" debug="true"
querySampler="UniformRandomFree">
       <Sampler label="OBPRM" number="100" attempts="1"/>
       <!--Connector label="Closest"/-->
       <!--Connector label="ConnectCCs"/-->
       <Evaluator label="NodesEval"/>
     </BasicPRM>
```

Medial-Axis PRM (MAPRM)

5. The same steps were followed for this sampler, except under the <MPStategies> section, the Sampler node's label was set to "MAPRM". Then, I ran pmpl and completed OBPRM sampling. I renamed the output files to MAPRM.map and MAPRM.stat.

```
<MPStrategies>
   <!-- Basic PRM where num samples is based on Number-->
     <BasicPRM label="PRM-build" debug="true"
querySampler="UniformRandomFree">
       <Sampler label="MAPRM" number="100" attempts="1"/>
       <!--Connector label="Closest"/-->
       <!--Connector label="ConnectCCs"/-->
       <Evaluator label="NodesEval"/>
     </BasicPRM>
```

Gaussian Sampler

6. The same steps were followed for this sampler, except under the <MPStategies> section, the Sampler node's label was set to "Gauss". Then, I ran pmpl and completed OBPRM sampling. I renamed the output files to Gauss.map, Gauss.stat, and Gauss.vd.

```
<MPStrategies>
   <!-- Basic PRM where num samples is based on Number
-->
     <BasicPRM label="PRM-build" debug="true"
querySampler="UniformRandomFree">
       <!--Connector label="Closest"/-->
       <!--Connector label="ConnectCCs"/-->
       <Sampler label="Gauss" number="100" attempts="1"/>
       <Evaluator label="NodesEval"/>
     </BasicPRM>
```

Bridge Sampler

7. The same steps were followed for this sampler, except under the <MPStategies> section, the Sampler node's label was set to "Bridge". Then, I ran pmpl and completed OBPRM sampling. I renamed the output files to Bridge.map, Bridge.stat, Bridge.vd.

```
<!-- Basic PRM where num samples is based on Number -->
     <BasicPRM label="PRM-build" debug="true"
querySampler="UniformRandomFree">
       <Sampler label="Bridge" number="100"
attempts="1"/>
       <!--Connector label="Closest"/-->
       <!--Connector label="ConnectCCs"/-->
       <Evaluator label="NodesEval"/>
     </BasicPRM>
```

Collecting Data with GNU

After I ran PMPL for each sampler, I used the .stat output files to collect my experimental data. In a **nodes.dat** file, I recorded the number of successful nodes added to the roadmap. In a **cd.dat** file, I recorded collision detector calls made when the roadmap ran. I used a script to graph the two .dat files with gnuplot.

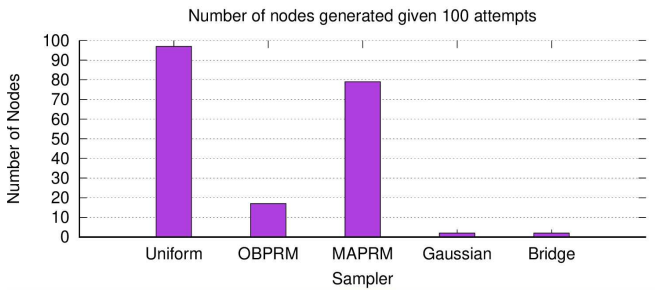


Figure 34. Nodes Generated by Distinct Sampling-based Planners

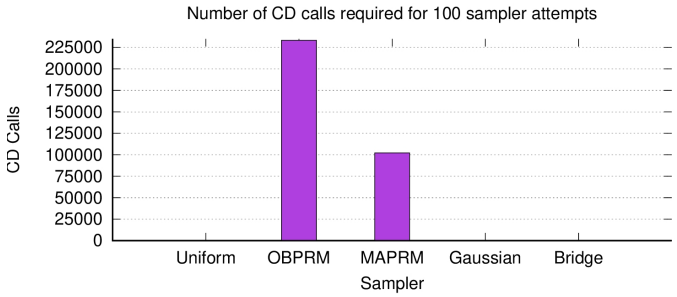


Figure 35. Collision Detection calls Required by Distinct Sampling-based Planners

My Modifications

I did not know what the "attempts" label in the Sample node, so my curiosity led me to see what would happen if I changed the attempts value.

```
<Sampler label="UniformRandomFree" number="n"
attempts="1"/>
```

```
<Sampler label="OBPRM" number="100" attempts="1"/>
```

Incremental Attempts for Uniform Sampler: The attempts made by a Uniform sampler does not significantly affect the success rate of the nodes added to the roadmap. There is no difference between a sampler running 100 samples at 10 attempts versus running 100 samples at 100 attempts. The ratio of number of nodes per CD call is close to 1:1, which is quite uniform.

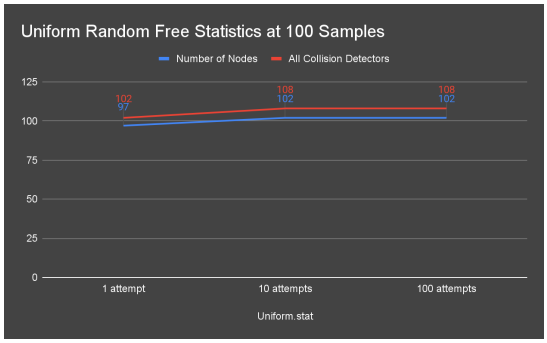


Figure 36. Nodes and CD calls Generated by *Uniform* with Varying Attempt Values

Incremental Attempts for OBPRM Sample: The number of attempts significantly increases the collisions detected for the OBPRM sample. The number of successful nodes generated also increases but not as significantly. The largest increase was going from 1 attempt, which generated 233,287 CD calls, to 10 attempts, which generated 1,056,911 CD calls.

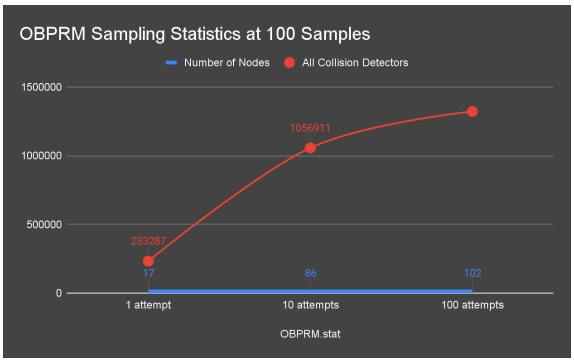


Figure 37. Nodes and CD calls Generated by *OBPRM* with Varying Attempt Values

I discovered that ***attempts*** affect motion planner results. When you want to insist on getting *n* number of samples each time you call the sampler but you only want to insist *x* amount of times, you must set the number of attempts to the value of *x*.

Visualizing Data with Vizmo

I used the .map files to visualize my results on Vizmo. The image below shows samples collected by four samplers. Each cube below is a sample, the thin horizontal rectangles are obstacles, and all C-space elements are in the 3D environment. My main observation is OBPRM is more efficient in sampling in an environment with a narrow passage.

Uniform Random Roadmap

Has more nodes (samples), nodes are located all over the space, nodes seem random. There seems to be a struggle with the nodes near the narrow passage between the obstacles. The success rate is proportional to free space volume.

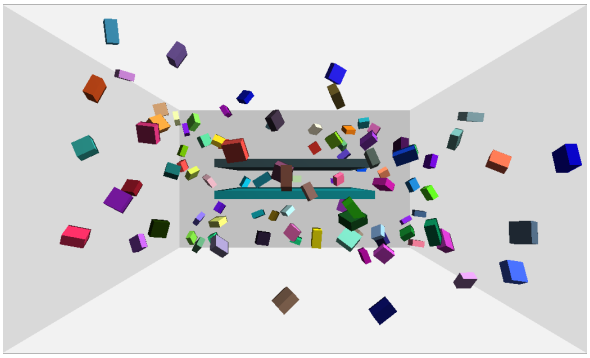


Figure 38. Uniform Random Roadmap Results

OBPRM Roadmap

Has significantly less nodes (samples), nodes don't seem randomly placed. Instead, nodes seem like they have been strategically placed in the environment efficiently and the narrow passage between the obstacles seems to be smoother. OBPRM pushes samples, so it is unlikely to fail.

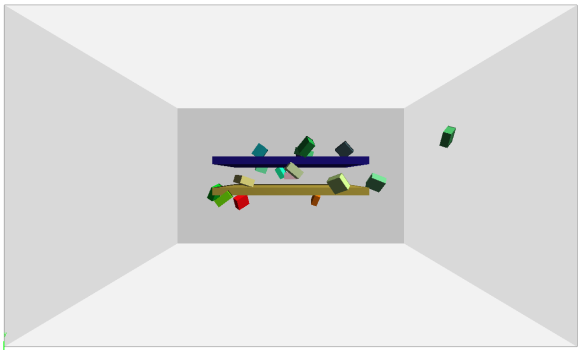


Figure 39. OBPRM Roadmap Results

Medial-Axis PRM Roadmap

The samples generated by the MAPRM appear to be retracted onto the medial axis of the free space but there also seems to be a few couple of samples within the narrow passage. The sampler has explored the narrow passage with less nodes than the PRM.

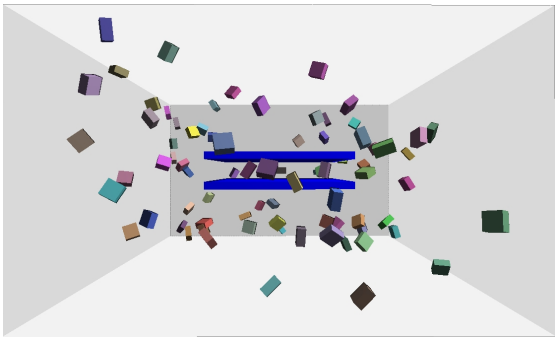


Figure 40. Medial-Axis PRM Roadmap Results

Gaussian Roadmap

The Gaussian roadmap only has 2 samples. One is farther on the outside of the obstacles And the other sample is directly within the narrow C-Free space between the two obstacles.

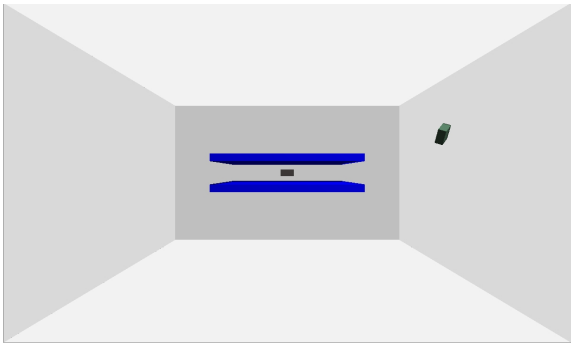


Figure 41. Gaussian Roadmap Results

Bridge Roadmap

The Bridge roadmap only has 2 samples. Like the Gaussian, one sample is farther on the outside of the obstacles and the other sample is directly within the narrow Cfree space between the two obstacles. It looks identical to the Gaussian roadmap. I ran this experiment twice to make sure I didn't make an error because of how identical the Bridge results and Gaussian results looked.

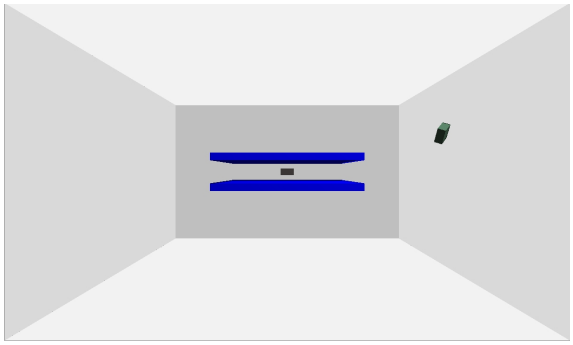


Figure 42. Bridge Roadmap Results

III. Experiment 3 Varying the Connection Techniques for PRMs

This experiment addresses how the value of k-nearest neighbors affects the connectivity of PRMs.

Hypothesis: A higher k-value will improve the connectivity of a roadmap and will make the runtime faster.

Procedures

The CfgExamples.xml file has a "k" tag that represents the k-neighbors used by Brute Force Neighborhood Finders (BFNF).

```
<BruteForceNF label="BFNF" dmLabel="euclidean"
unconnected="false" k="1"/>
```

I ran six independent PRM planner tests, where each test takes in a distinct k value of 1, 2, 4, 8, 16, 32. The files generated for each test will be renamed after the k value used. The k=1 test file was renamed k1.map and k1.stat, the k=16 test file was renamed k16.map and k16.stat, and so on. Below are all the output files I collected from all my experiment tests.

```
aec4@blackwidowers:~/PMPL/pmpl/src/Examples$ ls -d k*
k16.map  k16.stat  k16.vd  k1.map  k1.stat  k1.vd  k2
.map  k2.stat  k2.vd  k32.map  k32.stat  k32.vd  k4.m
ap  k4.stat  k4.vd  k8.map  k8.stat  kaec4@blacaecaec
```

Data

k	Edges	Connected Components	CD calls	Run Time
1	1,940	30	651,103	12 sec
2	3,860	6	1,523,135	23.9 sec
4	7,694	5	3,690,139	51.4 sec
8	15,318	2	9,095,618	119 sec
16	30,288	1	22,633,691	312 sec
32	59,336	1	56,247,675	741 sec

Figure 43. PRM Experiment Results with Varying K-Nearest Neighbor Values

I separated my data into three files for CDs, edges, and Connected Components (CCs) and added them to my GNU graphing script.

cd.dat		edges.dat		ccss.dat	
K-value	CD-calls	K-value	Edges	K-value	CCs
k=1	651103	k=1	1940	k=1	30
k=2	1523135	k=2	3860	k=2	6
k=4	3690,139	k=4	7694	k=4	5
k=8	9095618	k=8	15318	k=8	2
k=16	22633691	k=16	30288	k=16	1
k=32	56247675	k=32	59336	k=32	1

Figure 44. Data from Figure 43 Divided into Three Files for GNU

Results

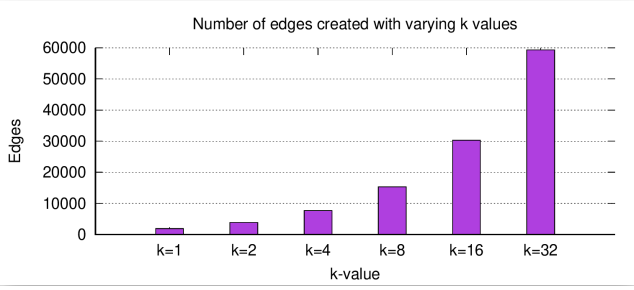


Figure 45. PRM Experiment Results for Edges Generated with Different k-Values

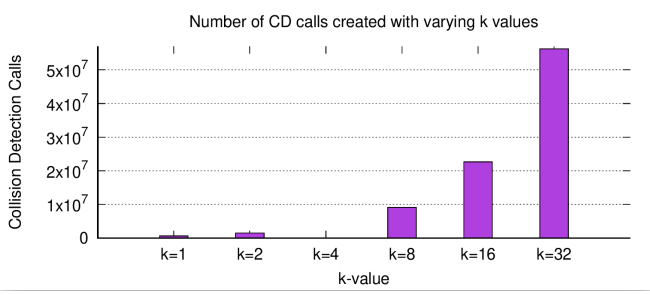


Figure 46. PRM Experiment Results for Collision Detection Calls Generated with Different k-Values

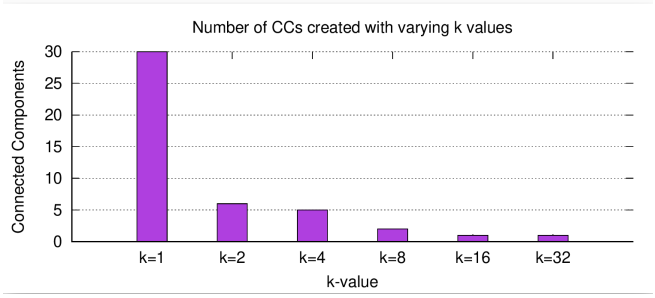


Figure 47. PRM Experiment Results for Connected Components Generated with Different k-Values

A higher k-value significantly improved the connectivity of a roadmap and reduced the distinct connected components towards 1 unified path. However, my hypothesis was partially correct: a higher k-value does improve connectivity. However, the run time increases as the k-value increases because it takes more time to produce higher counts of edges and compute less connected components with those edges. A trade-off to using a higher k-value is a more connected roadmap at a higher computationally expensive cost.

B. Quiz[3] 3

1. What are the output files generated by PPL? What information does each contain? Name the tool to view environments, roadmaps, and paths.

Running PMPL will create a .map, .stat, and .vd file. The output content depends on the specifications run on the XML file. The tool to view environments, roadmaps, and paths is named Vizmo.

The **.map** file contains

- Name of environment file used, when that file starts and ends
- Where graph stops and ends,
- Count and List of the successful nodes that were created by running PMPL with this sampler; there is a line per node

The **.stat** file contains

- Roadmap Statistics: counts number of nodes and edges in map
- Connected Components (CCs), each given an integer and "vid".
- Sampler Statistics that specified sampler used, total number of attempts (also specified by the Sampler node's "number" label), the number of successful attempts, and success rate.
- Collision Detection Calls that count all the collision detectors.
- Other statistics such as the query method
- Clock Data measured in seconds shows the time it takes for sampling algorithm to Connect, Evaluate Map, Generate Goals, Generate Start, Init and Run, Initialize, Run, and Sample.
- CCTracker::AddVertex, ConditionalEvaluator::NodesEval, MPLibrary::Initialize, and the Sampler.

[3] Not graded. Answers may not be correct due to no feedback.

2. Explain the visual differences in sampling methods in Figure 1. List each sampler and state where the samples are located.

The **Uniform Roadmap** has more nodes (samples), nodes are located all over the space, nodes seem random. There seems to be a struggle with the nodes near the narrow passage between the obstacles.

The **OBPRM Roadmap** has significantly less nodes (samples), nodes don't seem randomly placed. Instead, Nodes seem like they have been strategically placed in the environment efficiently And the narrow passage between the obstacles seems to be smoother. Vizmo didn't let me rotate and get another perspective on what's going on in the middle though.

The **Gaussian roadmap** only has 2 samples. One is farther on the outside of the obstacles and the other sample is directly within the narrow C-Free space between the two obstacles. It looks identical to the Bridge roadmap.

The **Medial-Axis PRM Roadmap** generated samples that were retracted onto the medial axis of the C-Free but also has a few couple of samples within the narrow passage. The sampler explored the narrow passage with less nodes than the PRM.

The **Bridge roadmap** only has 2 samples. Like the Gaussian, one sample is farther on the outside of the obstacles and the other sample is directly within the narrow C-Free space between the two obstacles. It looks identical to the Gaussian roadmap. I ran this experiment twice to ensure I did not make a mistake because of how identical the Bridge results looked to the Gaussian results.

3. In the sampling experiment, we were looking at success rates of samplers over 100 attempts. Explain why each sampler had the success rate it did.

The OBPRM is most efficient in C-Spaces with more obstacle space and less free space. In environments with more C-Free, the main problem for the OBPRM sampler is distance. Distance causes 2 issues. Firstly, if the distance between two random nodes is larger, we can miss an obstacle in between those pairs of nodes. Secondly, if the distance between two random nodes is smaller, there will be more Collision Detection calls. My source for this information is Diane. I think that because the 3D environment used for this experiment contains more C-Free space than CObst space, there was a lower success rate on the OBPRM sampler whereas the Uniform Random Free was more successful. The OBPRM roadmap has 17 successful nodes close together with short distances from each other, which explains why it has a high, staggering 233,287 CD calls, significantly less than the Uniform sampler's 97 successful nodes and 102 CD calls.

The Gaussian and Bridge roadmaps only take 2 samples at a time, so it makes sense why each of the samplers only had 2 successful nodes. It is simply how their algorithm works. The MAPRM visually has a bunch of nodes toward the outside of the Vizmo box at the medial axis and has less randomly dispersed samples. Since the samples are mainly around the obstacle, it makes logical sense for there to be more successful nodes since less of them are going to be generated in C-Obst.

4. Why is collision detection a measure of efficiency in motion planning?

Earlier in the crash course, we learned that Collision detection checks whether a given configuration lies in C-Free or C-Obst, or in simpler words, checks the validity of a configuration. This critical, elementary component of sampling that checks validity is a great measuring tool to see if a motion planning algorithm is efficient.

5. In the connection experiment, how does varying *k* affect collision detection calls? Does this trend always hold? Why does the number of calls more than double in your plots?

My data shows that as varying *k* values increase, the collision detection (CD) calls increase significantly. This trend always held true. To specify, CD calls increased exponentially when k equaled 1, 2, and 16. When k=1, 6.51103×10^5 CD calls were made. When k=2, 1.523135×10^6 CD calls were made. When k=16, 2.2633691×10^7 CD calls were made. The number of CD calls more than doubles because as the k value increases, the number of edges increases. Since there are more edges connecting nodes, the planner grinds to check if the edges collide with C-obstacle spaces. More *k-neighbors* means more edges, which means more CD calls.

6. Why do you think we want higher k value for connection?

We want a higher K value because it (1) doubles the number of generated edges between nodes and (2) decreases the number of connected components, which significantly improves the connectivity of our roadmap. More edges between nodes and less quantities of connected components makes it easier to find valid non-collision shorter paths and can help us tackle more complex motion planning problems like the narrow passage.

Module 4: The Parasol Planning Library

A. Programming in C++

Coding a Motion Planning Program

In this assignment, I followed complex instructions to code an MP Strategy that generates a roadmap with 4 nodes connected by 3 edges. I worked in a 6-DOF environment where each coordinate is represented as a double. I used the 3D environment and boxy robot for my algorithm and visualized my results in vizmo.

Part 1 >

1. Created MyStrategy.h file in src/MPLibrary/MPStrategies with this code.

2. Opened CfgTraits.h in src/Traits to include my header file with the other included MPStrategies as well as in the list<> of strategies.

> CfgTraits.h

```
//mp strategies includes
<... more code ...>
#include "MPLibrary/MPStrategies/MyStrategy.h"

//types of motion planning strategies available in our
world
 typedef boost::mpl::list<
   <... more list items ...>
   MyStrategy<MPTraits>,  <... more list items ...>
     > MPStrategyMethodList;
```

Now, MyStrategy has been included in PPL and it successfully compiled with everything else when I ran **make pmpl -j2** from src.

4. Open CfgExamples.xml and add a MyStrategy node with its label, then go to the Solver node and change its mpStrategyLabel to "MyStrategy". I like changing my baseFilename to match the experiment I'm doing so that my files get appropriate names and don't get mixed up.

CfgExamples.xml >

```
        <!-- My Strategy-->
    <MyStrategy label="MyStrategy"/>
</MPStrategies>
  <Solver mpStrategyLabel="MyStrategy" seed="12345678"
    baseFilename="MyStrategy" vizmoDebug="true"/>
```

Part 2 >

I executed PPL with the CfgExamples.xml file.

```
aec4@blackwidowers:~/PMPL/pmpl/src/Examples$ ../pmpl -f
CfgExamples.xml
Automatically computed position resolution as 0.01
MPLibrary is solving with MPStrategyMethod labeled
MyStrategy using seed 12345678.
MyStrategy::MyStrategy::Initialize: 0 sec
: 0 sec
```

I was able to see that my Blank method output a file:

```
aec4@blackwidowers:~/PMPL/pmpl/src/Examples$ ls MyS
MyStrategy.12345678.query.vd
```

This means that the motion planning program was set up correctly. Now, I have a blank, testable method that is ready to build on!

Part 3 >

```
protected:
   CfgType GenerateNode();
   bool ConnectNode(CfgType&_c);

 private:
   // member variables for GenerateNode()
   string m_vcLabel; // for validity checker
   string m_dmLabel; // for distance metric
   string m_lpLabel; // for local planner
   // member variables for ConnectNode(CfgType&_c)
   CfgType m_lastNode; // holds a copy of the last node
in roadmap
   double m_length; // holds the total chain length
```

Part 4 >

Initialized global member variables under the template ParseXML method so that it can be easier to change different metrics for experiments and reduce procedures to a one-word substitution in the XML file. I set each variable's required parameter to false for all three m_labels so that we can use the default values.

```
// Inherited Methods from parent MPStrategyMethod.h

template<class MPTraits>
void
MyStrategy<MPTraits>::
ParseXML(XMLNode& _node) {
 // m_var  = <stuff>(<XML label>, <required>, <default>,
<type>);
 m_vcLabel = _node.Read("vcLabel", false, "pqp_solid",
"Validity Checker");
 m_lpLabel = _node.Read("lpLabel", false, "sl", "Local
Planner");
 m_dmLabel = _node.Read("lpLabel", false, "euclidean",
"Local Planner");
```

Part 5 >

I wrote the code below to generate one node in the GenerateNode() function. I also added comments.

```
template<class MPTraits>
typename MPTraits::CfgType // needed because the typedef
in the class def is not in this scope
MyStrategy<MPTraits>::
/////////////////////////////////////////////////////////
////////////////////////
//   Makes a single node using the Cfg class (typedef
typename MPTraits::CfgType)
//   and collision check it with the ValidityChecker
stored in MPProblem.
/////////////////////////////////////////////////////////
////////////////////////

GenerateNode() {
```

```
 // Get a pointer to the ValidityChecker and Environment,
create a new CfgType

 // this-> pointer allows us to get access to these
objects
 //        through the MPBaseObject (parent of
MPStrategyMethod)
 auto vc = this->GetValidityChecker(m_vcLabel);
 auto env = this->GetEnvironment();
 CfgType newCfg(this->GetTask()->GetRobot());

 // Generate a random CfgType until we find one that is
inside the
 // bouding box and not in collision
 do {

newCfg.GetRandomCfg(this->GetEnvironment()->GetBoundary()
);
 }
 // if the node is either out of bounds or invalid, then
resample the node
 while(!newCfg.InBounds(env) || !vc->IsValid(newCfg,
this->GetNameAndLabel()));
 return newCfg;
 // GenerateNode is now a simple, one-node sampler that
we can use inside Run to
 // keep things orderly and readable.
}
```

Part 6 >

I set the return value for ConnectNode() to false so that I can test what I did for GenerateNode().

```
template<class MPTraits>
MyStrategy<MPTraits>::
bool
ConnectNode(CfgType&_c) {
 return false; // false value allows us to test
}
```

I also added a cout statement to Initialize() to confirm that a specific chunk of my code compiles when it gets printed in the terminal after I run my method. My cout message is adorable and very roboty…"beepboop"! This is a favorite, go-to method.

```
template<class MPTraits>
void
MyStrategy<MPTraits>::
Initialize() {
 cout << "Robots go beepboopbeepboop" << endl;
}
```

```
MPLibrary is solving with MPStrategyMethod labeled MyStrategy using seed 1234567
8.
Robots go beepboopbeepboop
MyStrategy::MyStrategy::Initialize: 0 sec
: 0 sec
MPLibrary::Initialize: 0 sec
aec4@blackwidowers:~/PMPL/pmpl/src/Examples$
```

I learned the importance of testing my code at each little step to make debugging easier. If I had tested my code after writing a couple sections, it would take more time to discern what chunk caused my program to crash.

Part 7 >

1. I introduced two private member variables in class definition, CfgType m_lastNode and double m_length, which will be used in the ConnectNode method.

```
   CfgType m_lastNode; // holds a copy of the last node
in roadmap
   double m_length; // holds the total chain length
```

I also included a default initialization for m_lastNode and m_length in both MyStrategy constructors so that we can initialize these variables when the class object is constructed.

```
template<class MPTraits>
MyStrategy<MPTraits>::
MyStrategy()
: m_lastNode(), m_length(0.)  {
 this->SetName("MyStrategy");
}
template<class MPTraits>
MyStrategy<MPTraits>::
MyStrategy(XMLNode& _node)
 : MPStrategyMethod<MPTraits>(_node), m_lastNode(),
 m_length(0.) {
   this->SetName("MyStrategy");
}
```

2. Then, I initialized m_lastNode to an empty CfgType object in the Initialize() method.

```
Initialize() {
 m_lastNode = CfgType(this->GetTask()->GetRobot());
 cout << "Cute Robots go beepboopbeepboop :D" << endl;
}
```

3. I went back to the ConnectNode method and set up pointers to access the Environment, Graph, DistanceMetric, and the LocalPlanner through the get-chain.

```
auto lp = this-> GetLocalPlanner(m_lpLabel);
 auto dm = this-> GetDistanceMetric(m_dmLabel);
 auto r = this->GetRoadmap();
 auto env = this->GetEnvironment();
```

4. Added code in ConnectNode to test if this is the first node. If it is, add it to the Roadmap right away because there is nothing for this node to connect to, thus it has already been determined valid in GenerateNode. Then, update the information for the last node in the chain.

```
CfgType blank(this->GetTask()->GetRobot());
 if(m_lastNode == blank) { // checks if m_lastNode still
holds a config of value 0 for all DOF (default)
   r->AddVertex(_c);
   m_lastNode = _c;
   return true;
```

5. For nodes subsequent to m_lastNode, we need to determine whether this node can be connected to m_lastNode. We need the following temporary nodes to do this.

```
// @TemporaryNodes
 // Needed by the LocalPlanner to store the output edge
 // and the Cfg where collision occurs
 LPOutput<MPTraits> lpOutput;
 auto robot = this->GetTask()->GetRobot();
 CfgType collisionCfg(robot);
```

6. Now, I will test connection with the LocalPlanner and add the new node and edge to the Roadmap Beforehand, I will include this code to add the distance between the nodes to the running total:

```
 // Test connection with the LocalPlanner, and
 // add the new node and edge to the Roadmap if it
succeeds
 if(lp->IsConnected(m_lastNode, _c, collisionCfg,
&lpOutput,
   env->GetPositionRes(), env->GetOrientationRes())) {
   m_length += dm->Distance(m_lastNode, _c);  // add
distance between nodes
   VID newNode = r->AddVertex(_c);
   r->AddEdge(r->GetVID(m_lastNode), newNode,
lpOutput.m_edge);
   m_lastNode = _c;
   return true;
```

Part 8 >

I tested my implementation so far.

```
[aec4@nancydrew Examples]$ ../pmpl -f CfgExamples.xml
Automatically computed position resolution as 0.01


MPLibrary is solving with MPStrategyMethod labeled MyStrategy using seed 12345678.
Robots go beepboopbeepboop :D
MyStrategy::MyStrategy::Initialize: 0.000225 sec
: 0 sec
MPLibrary::Initialize: 0 sec
```

Part 9 >

I added this code chunk to create and connect a new Cfg node.

```
template<class MPTraits>
void
MyStrategy<MPTraits>::
Run() {
  auto r = this->GetRoadmap();
 size_t numNodes = 4;

 while (r->get_num_vertices() < numNodes) {
   cout << "Num Vertices: " << r->get_num_vertices() <<
endl;
   CfgType newCfg = GenerateNode(); // create a new
configuration node
   ConnectNode(newCfg); // connect each new configuration
node
 }
}
```

Part 10 >

I added this code and comments. See comments for information.

```
template<class MPTraits>
void
MyStrategy<MPTraits>::
Finalize() {

 //  outputs the path length with a simple cout
statement:
 cout << "MyStrategy Path Length: " << m_length << endl;

 // always start with the base filename, which is owned
by the base class MPStrategyMethod
  // set the file name export .map files as
"filename.map":
 string fileName = this->GetBaseFilename() + ".map";

 // call on the Roadmap class to write its data to
complete the Finalize method:
 this->GetRoadmap()->Write(fileName,
this->GetEnvironment());
}
```

```
template<class MPTraits>
void
MyStrategy<MPTraits>::
Print(ostream& _os) const {
 // all classes in PPL are required to define their Print
function to return their name and label.
 _os << this->GetNameAndLabel() << endl;
}
```

I tested my most recent code above and it compiled. All the code in MyStrategy.h compiles when I run CfgExamples.xml.

Part 11 >

When MyStrategy.h compiled and I ran CfgExamples, it successfully generated a roadmap with the four connected nodes. Then, I visualized the roadmap in Vizmo using the 3D environment and the boxy robot. Overall, this was an insightful coding experience and I learned the importance of testing each small section of code before writing the next section. This ensures my ability to dissect and isolate errors more efficiently and effectively during the debugging process. Ultimately, I understood and accomplished my goals and completed this assignment.

Results

> MyStrategy.12345678.query.map

```
####ENVFILESTART#####
3D/3d.env
#####ENVFILESTOP#####
#####GRAPHSTART#####
4 6 4
0 0   -2.1236499930773611e+01    7.2997435884681749e+00   -1.3396965768600388e+01    6.1612597870943375e-01   -7.4840267825544959e-02
1 0   -2.1395717780933943e+01    1.1101307876063373e+01   -3.3778336561001154e+00    5.5307073629249714e-01    9.1976031562229110e-01
2 0   -1.8106546354536057e+01    1.0000202268113156e+00   -9.8268815049825875e+00   -7.5212023591035404e-02   -2.5109501501783171e-01
3 0   -9.1639667406335334e+00   -8.1949233541963196e+00    1.1267405696099253e+01    8.4004666454509902e-01    3.0100230155789376e-01
#####GRAPHSTOP#####
```

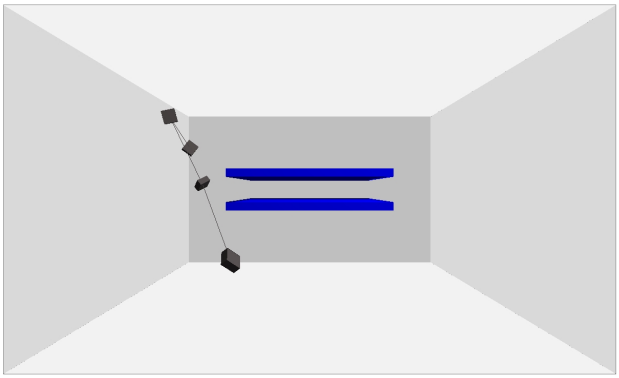


Figure 48. Visualization of My Motion Planning Program (Front View)

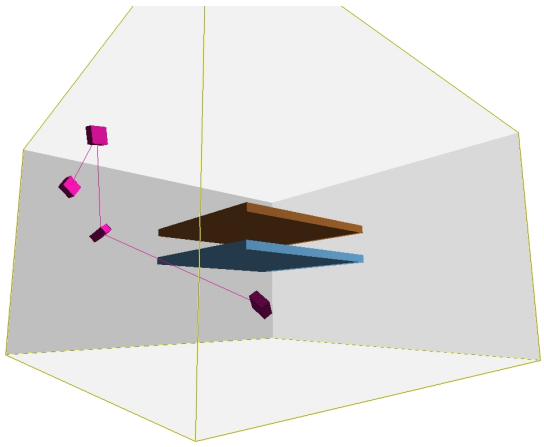


Figure 49. Visualization of My Motion Planning Program (Bottom Side View)

A. Readings

The required reading was Chapter section 3.4, "The Topology of the Configuration Space" found in *Principles of Robot Motion: Theory, Algorithms, and Implementations* by Choset et al (2005). This section introduces **topology** as "a branch of mathematics that considers the properties of objects that do not change when the objects are subjected to arbitrary continuous transformations such as stretching or bending" without tearing it. To a topologist, a coffee mug and a torus are topologically equivalent: both have one hole and can be transformed to one another by bending and stretching, as demonstrated below in Figure 50.

Figure 50. Continuous Transformation of a Coffee Mug to a Donut. A coffee mug and a donut are topologically equivalent, which is illustrated by this homeomorphic transformation. (Source: https://cems.riken.jp/en/laboratory/qmtrt).

Two spaces are **topologically different** if cutting or pasting is required to turn one into the other. This is because cutting or pasting are not considered continuous transformations. The two main continuous transformations introduced in the research paper are homeomorphism and diffeomorphism, which will be defined in the Quiz 4 section.

B. Further Research

I found a lecture video on configuration space topology given by Dr. Kevin Lynch, a professor at Northwestern University. The lecturer stated that two spaces are **topologically equivalent** to each other if one can be smoothly transformed to the other without cutting, pasting, or gluing. Dr. Lynch makes a distinction: the topology of a shape affects how we use coordinates to represent a space whereas he topology of a space= is not affected by the choice of how to represent the space with coordinates. Topologically distinct one-dimensional spaces are lines, closed intervals of a line, and circles. Topologically distinct two-dimensional spaces are planes, spheres, torus, and cylinders.

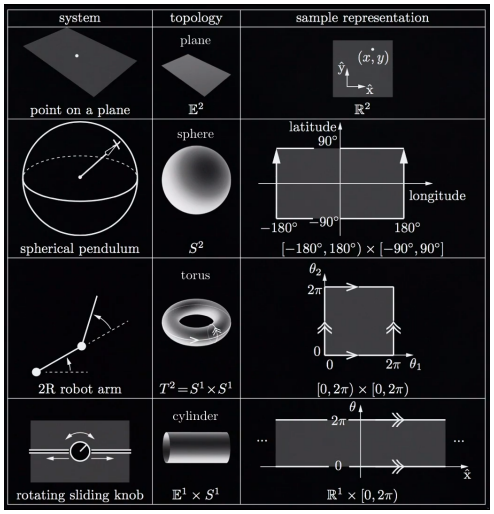
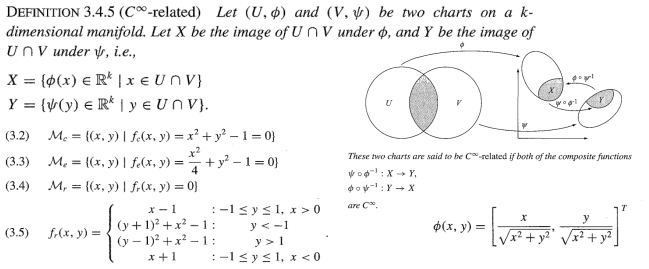


Figure 51. Topological Representations of Different Systems Found in Robotics
(Source: https://www.youtube.com/watch?v=FyLNR3edOds

The lecture taught me how the topology of a 2-Revolute robot arm is identical to the topology of a torus by using simple visuals and explanations. Learning a mathematical concept I thought I could not understand is a great feeling. I will use the information from Figure 51 to explain how the topology of a configuration space is relevant to motion planning in the Quiz 4 section.

C. Feedback

The required reading was extremely difficult to understand with the advanced mathematics formulas. I suggest making this module more interactive and I especially recommend focusing on the fundamentals of topology with strong visuals or playdough before you introduce terms and formulas for homeomorphism, diffeomorphism, and manifolds. We can memorize the definitions and look at the formulas to overcome the reading but that is not an effective way of truly understanding how topology matters in robotics. One thing we never did in the crash course lecture was analyze the formulas in the readings. I would have loved to dedicate time to dissect topology formulas and practice using them to solve robotics problems. Because this was not done, some of the advanced formulas (below) remain a mystery to me.



DEFINITION 3.4.5 (C^∞-related) *Let (U, ϕ) and (V, ψ) be two charts on a k-dimensional manifold. Let X be the image of $U \cap V$ under ϕ, and Y be the image of $U \cap V$ under ψ, i.e.,*

$$X = \{\phi(x) \in \mathbb{R}^k \mid x \in U \cap V\}$$
$$Y = \{\psi(y) \in \mathbb{R}^k \mid y \in U \cap V\}.$$

(3.2) $\mathcal{M}_c = \{(x, y) \mid f_c(x, y) = x^2 + y^2 - 1 = 0\}$

(3.3) $\mathcal{M}_e = \{(x, y) \mid f_e(x, y) = \frac{x^2}{4} + y^2 - 1 = 0\}$

(3.4) $\mathcal{M}_r = \{(x, y) \mid f_r(x, y) = 0\}$

(3.5) $$f_r(x, y) = \begin{cases} x - 1 & : -1 \le y \le 1,\ x > 0 \\ (y+1)^2 + x^2 - 1 & : y < -1 \\ (y-1)^2 + x^2 - 1 & : y > 1 \\ x + 1 & : -1 \le y \le 1,\ x < 0 \end{cases}$$

These two charts are said to be C^∞-related if both of the composite functions $\psi \circ \phi^{-1} : X \to Y$, $\phi \circ \psi^{-1} : Y \to X$ are C^∞.

$$\phi(x, y) = \left[\frac{x}{\sqrt{x^2 + y^2}}, \frac{y}{\sqrt{x^2 + y^2}}\right]^T$$

Figure 52. Collage of Confusing Math Definitions in the Module 5 Topology Reading

D. Quiz[4] 4

1. How is the topology of a configuration space relevant to motion planning?

A configuration space is a topological space, therefore we can concisely describe the C-space of a robot using terminology from topology. This allows us to ignore differences in the construction of spaces that don't affect certain properties. The C-Space topology of a 2-Revolute-joint robot arm is represented by a torus, which is represented by two joint-angle coordinates between 0 and 2π. When the configuration of the torus moves smoothly, the coordinate representation changes discontinuously at $[0,\ 2\pi) \times [0, 2\pi)$. In Figure 53, the robot arm's movement (left) is shown on the torus (middle), and the coordinate representation jumps from one edge of the coordinate square to the other (right).

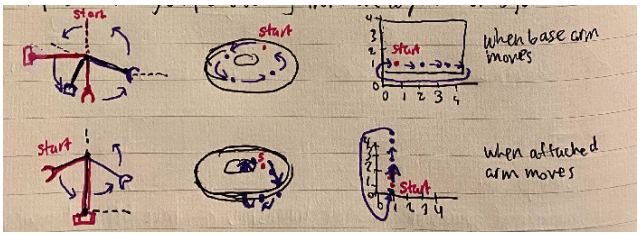


Figure 53. My Sketch that Illustrates 2-Revolute-Joint Arm Movement on a Torus

2. Define homeomorphism, diffeomorphism, manifold, differentiable manifold, and homotopy class.

Homeomorphism is a mapping from one set to another which is a continuous bijection. Two sets are homeomorphic *iff* this mapping exists. This is a type of continuous transformation found in topology and can be best visualized by squishing a coffee mug into a donut, as seen in Figure 50. A **manifold** is a set which is locally homeomorphic to $\mathbb{R}^k$. **Diffeomorphism** is a smooth mapping from one set to another which is a continuous bijection. Two sets are diffeomorphic *iff* this mapping exists. A **differentiable manifold** is a set which is locally diffeomorphic to to $\mathbb{R}^k$.

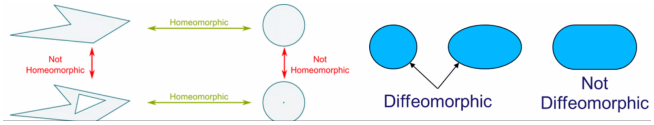


Figure 54. Homeomorphic and Diffeomorphic Shapes (Source: Parasol Laboratory)

Homotopic classes are those in which, given two topological spaces *X* and *Y*, place an equivalence relationship on the continuous maps $f : X \to Y$ using homotopies. The two spaces *X* and *Y* are homotopically equivalent if they can be transformed into one another by bending, shrinking, and expanding. An example of homotopy is the case of continuous maps from one

[4] Not graded. Answers may not be correct due to no feedback.

circle to another circle. Suppose you have an infinitely stretchable string that can be tied around a tree. The string can be stretched to be the first circle S^1, and the tree trunk's surface is the second circle S^1. There are many ways to tie the rope around the tree. For any integer *n,* the string can be wrapped around the tree *n* times. Each integer *n* corresponds to a homotopy class of maps from $f\colon S^1 \to S^1$ (Wolfram MathWorld, 2023).

Figure 55. A Continuous Map from One Circle to Another Circle. The circled rope is homotopically equivalent to the circled tree trunk it is wrapped around.

<u>3. Considering homeomorphism, how many holes does a shirt have? What about pants? Socks?</u>

Figure 56. A Cup Without a Handle. The cup opening only lets you pour drinks inside the enclosed cup. However, your drink cannot spill through the other side of the cup because there is no hole at the bottom due to the sealed-shut glass. Thus, the handleless **cup has 0 holes**. Apply this logic to a sock.

Figure 57. A Sock. The sock opening only allows you to put your foot inside the enclosed sock. Likewise, you cannot put your foot through the other side of the sock because there is no hole at the bottom of it due to the sewn-shut fabric. Thus, a **sock has 0 holes**.

Figure 58. A Coffee Mug. Add a handle to a handleless cup. Now you have a coffee mug, a completely different object according to a topologist! The handle makes the topology of a mug distinct from the topology of a cup.

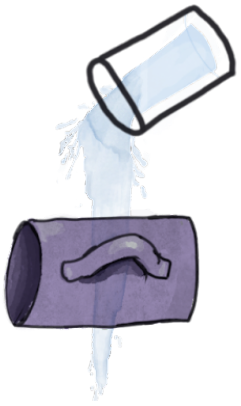
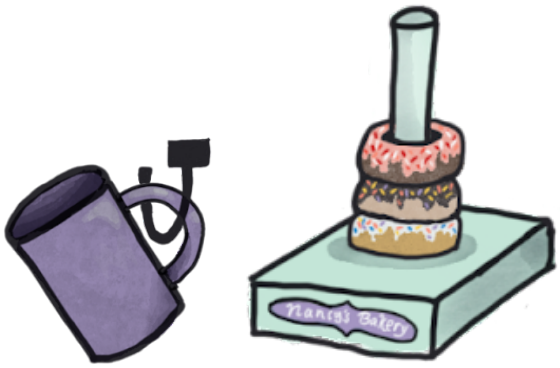


Figure 59. A Body of Water Passing Through the Handle of a Coffee Mug. Tilt the mug on its side, aim the cup at the mug's handle, and pour the cup's drink inside the mug handle's hole. Look, the water is falling through the other side of the handle! AHA, the mug's handle is a hole! Thus, a coffee mug has one hole.

Figure 60. A Coffee Mug and a Donut. Like a coffee mug, a donut also has one hole. Below, the coffee mug and donuts are displayed by passing their singular hole through a hanging hook and a pastry tray pole, respectively. A reason that the topology of the C-space is important is that if we can derive a motion planning algorithm for one topological space, the algorithm can carry over to other spaces that are topologically equivalent (Choset et al, 2005). So, if we want to create a funky motion plan to put a donut on a hook or a coffee mug handle on a pastry tray pole, we know that a possible solution can exist!



Given our previous observations, we can logically conclude that the opening spaces of a pants or shirt is not a hole, it simply allows us to blindly enter the geometric space of the clothes. In pants, the first hole is for the left leg and the second hole is for the right leg. Thus, **pants have two holes**. In a shirt, the first hole is for the cranium to go through, the second hole is for the left arm, and the third hole is for the right arm. Thus, a **shirt has three holes**.

<u>4. Use this information to answer questions 4 a-c:</u>
Imagine you have a robot with a mobile base and a jointed manipulator attached at the top, as in the following figure.

Assume the base can only move in one dimension (along a straight line) and the manipulator only has one revolute joint (which can freely move in a circle).

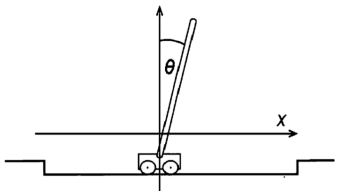


Figure 61. A robot with a Mobile Base and Single Revolute Joint

4a. What is the shape of the C-Space for this robot?

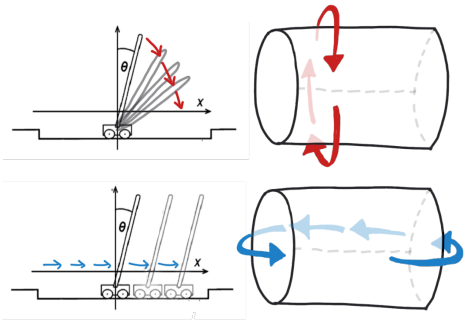


Figure 62. Topologically Equivalent 1-Revolute-Joint Mobile Robot and Cylinder.

The shape of the C-Space for the robot in Figure 61 is a cylinder. I added some motion and arrows to each pair of robot and cylinder in Figure 63 to visualize their topological relationship.

b. How would you specify the coordinates of a configuration in this C-space?

I would specify the configuration with one coordinate ranging between 0 and 2π and represent it as $\mathbb{R}^1 \times [0, 2\pi)$.

c. How could you compute the distance between two configurations in this space?

I would cut the cylinder to get a square subset of the plane in order to get the coordinates of the two configurations and use distance metric formulas to compute the distance between them.

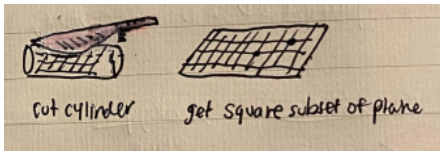


Figure 63. Cutting Cylinder to get Square Subset of Plane.

Acknowledgements

My utmost sincerest thank you to the MSCS Consortium Pathways to Computing for funding my Distributed REsearch Apprenticeships for Master's (DREAM) Program. Thank you to Kathleen Kelly, Christine Liebe, and all other DREAM program associates for the opportunity to participate in my first computer science research experience. Many thanks to my advisor, Dr. Nancy M. Amato, for the opportunity to learn about robotics as part of her laboratory and for her emotional support throughout my Parasol and iCAN experience. Thanks to my algorithms professor, Dr. Yael Gertner, for the recommendation. Thanks to Dr. Marco Morales for his guidance in the weekly Parasol meetings and insight on how to test code effectively. Many thanks to Diane Uwacu, my main graduate mentor, for her utmost patience and dedication to teaching me motion planning, helping me debug my code, and advising me on how to be a better computer science student. Thanks to graduate students, especially my other mentors Dr. Homa and Hannah, for your time in helping me whenever I struggled with the crash course, bugs, terminal, and Vizmo. As a graduate student who specialized in political science and linguistics in undergraduate school, learning robotics was my most challenging academic experience. It took a village! Thank you all for being a part of my research journey.

References

1. N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones, D. Vallejo. OBPRM: An Obstacle-Based pRM for 3D Workspaces. 1998. {https://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/PRM/prmsampling_01.pdf}

2. H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, S. Thrum. Principles of Robot Motion: Theory, Algorithms, and Implementations. Chapter 3. 2005. {https://parasollab.web.illinois.edu/ppl_crashcourse/documents/principles_robot_motion.chapter_3.pdf}

3. C. Foster. House & Garden. "How to beautify the exterior of your house with plants". 2022. {https://www.houseandgarden.co.uk/article/plants-for-house-exteriors}

4. H. A. Gonzales. Quadrotor Vehicle with Manipulator Arm. 2015. {https://www.researchgate.net/figure/CAD-sketch-of-the-quadcopter-with-its-robotic-arm_fig20_286029560}

5. K. Hausner, Intelligent Motion Library. Robotics Systems (Draft). Section III, Motion Planning. 2020. {http://motion.cs.illinois.edu/RoboticSystems/MotionPlanningHigherDimensions.html}

6. L. E. Kavraki, P. Svestka, J.-C. Latombe, M. Overmars. Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces". 1996.

{https://www.cs.rice.edu/CS/Robotics/papers/kavraki1996prm-high-dim-conf.pdf}

7. S. Keshav. How to Read a Paper. 2007. {https://www.albany.edu/spatial/WebsiteFiles/ResearchAdvices/how-to-read-a-paper.pdf}

8. J.-C. Latombe. A Journey of Robots, Molecules, Digital Actors, and Other Artifacts. 1999. {http://robotics.stanford.edu/~latombe/cs326/2009/class1/ijrr.pdf}

9. S. M. LaValle. Rapidly-Exploring Random Trees: A New Tool for Path Planning. 1998. {http://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf}

10. N. J. Nilsson, SRI International. Shakey the Robot. 1984. {http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/shakey-the-robot.pdf}

11. M. H. Overmars. 2005. Path Planning For Games. {http://www.cs.uu.nl/centers/give/movie/publications/PDF/O05.pdf}