Real-time Robot Motion Planning

Sterling McLeod

I. Introduction

Robot Motion Planning is the problem of moving a robot from an initial state to a goal state in some environment. States can be positions, or they can be high-dimensional configurations. They can also include dynamic information, such as a specific velocity and/or acceleration. Motion planning is required in almost every robotics tasks that exists. Unfortunately, this problem is made difficult by several things:

- *Kinematic structure of a robot*
  Most path-planning algorithms will work in a robot's Configuration Space (Section II). The number of dimensions in this space scales with the degrees of freedom in a robot's kinematic structure. Mobile bases will only have 2-3 dimensional C-spaces, but manipulators can have up to 7. Humanoid robots can have over 20. Finding a path in such high-dimensional spaces is extremely computationally expensive.
- *Complexity of an environment*
  Environment complexity is defined by the obstacles existing in it. If all obstacles are static, then the planning will not be too hard. If some or all of the obstacles are dynamic, then the planning is made much more complicated. Some planning algorithms build graphs that assume the obstacle are static. If the obstacles are dynamic, then the algorithms must rebuild a graph representing the state space. Rebuilding this space can be too computationally expensive to execute at real-time.
  The dimensionality of an environment is also an important factor. Generally, mobile bases operate in a 2D environment. Manipulators and humanoids, however, operate in a 3D environment. The added dimension brings many more computational costs when performing collision detecton.
- *Amount of obstacle information*
  The inherent complexity of dynamic obstacles can be diminished if a planner knows useful information about the obstacles. If we know exactly how an obstacle will move, then a planner can know where it will be in the future and plan accordingly. Partial information about obstacle movement would significantly reduce the planning problem complexity because it allows a system to make assumptions. For instance, in the domain of self-driving cars, an assumption can be made that the surrounding cars will only move forward. This means that a planner does not have to consider the case where an obstacle may move directly at the robot. Other domains, such as warehouse robots, cannot have this assumption applied. In order to have a general planner, one must assume as little as possible about obstacles. The algorithm must be developed to handle lots of various situations.

The path-planning work is normally carried out in a robot's Configuration Space (Section II) to simplify making connections from one state to another.

The earliest works revolve around complete planners operating on a roadmap of the robot's environment. However, building such a roadmap in a 3D environment is extremely computationally expensive. Methods that

II. Configuration Space

A robot's *Configuration Space*, or C-space, is a manifold mapping the set of all transformations that can be applied to a robot to Euclidean space. The dimensionality of this space is determined by the number of degrees of freedom in a robotic system. A robot's *configuration* is a complete specification of the positions of a robot's points. It is typically represented as a vector specifying the values for each of a robot's degrees of freedom. A more intuitive way to think of C-space is as the set of all possible configurations that a robot can have. Each point in the C-space is a vector specifying a transformation of the robot.

In order to move a robot's end-effector, actuators have to produce rotations or translation around the joints of a robot. This means that any motion of a robot is entirely controlled by the motion of its joints, or the change in configuration. Therefore, robot motion planning is concerned with how to move a robot from an initial configuration to a goal configuration.

To define the C-space for a robot more formally, both the range and type of motion of the robot must be considered. If a robot is a rigid-body object translating in a plane, then its transformation can be expressed in terms of x and y. This yields a manifold $M = \mathbb{R}^2$. If rotations can be applied to the robot, then another dimension must be added to the C-space manifold. Let $\theta = [0, 2\pi]$ be the range of rotation for the robot. The C-space manifold for a rigid body robot that can translate along x and y and rotate $[0, 2\pi]$ in a plane becomes $M = \mathbb{R}^2 \times \mathbb{S}^1$.

For rigid bodies moving in a 3D space, the C-space becomes more complicated. Generally, objects in a 3D space can translate in three dimensions and can rotate about three axes independently. This set of transformations is characterized by a position vector, p, and a rotation matrix, R, relative to a fixed frame. The resulting C-space will be six-dimensional: $C = \mathbb{R}^3 \times \mathbb{RP}^3$.

Manipulators consist of multiple rigid bodies that can either translate or rotate independently of one another. When computing the C-space of articulated bodies, the C-spaces of *each* body must be combined. If a manipulator is a chain of n links connected by revolute joints and each joint can achieve full rotation $\theta_i = [0, 2\pi]$, then the C-space is defined as

$$C = \mathbb{S}^1 \times \mathbb{S}^1 \times ... \times \mathbb{S}^1 = \mathbb{T}^n \tag{1}$$

Visually, this C-space is an n-dimensional torus. If a manipulator can translate in a 3D space, then an additional $\mathbb{R}^3$ must be considered: $C = \mathbb{R}^3 \times \mathbb{T}^n$. In the real world, however, it may not be possible to achieve a full rotation around each joint. If that is the case, then the topological space representing rotation in the C-space would be $\mathbb{R}^n$ instead of $\mathbb{T}^n$.

The C-space in a motion planning problem is separated into two subspaces, C_{obs} and C_{free}. The subspace C_{obs} represents the set of configurations containing obstacle regions and $C_{free} = C \backslash C_{obs}$. Let A be a rigid body robot and O be an obstacle region. Both A and O exist in a world space W. The C-space of A is C and a configuration $q \in C$ is defined as $q = (x, y, \theta)$. The obstacle region, C_{obs}, is defined as

$$C_{obs} = \{q \in C | A(q) \cap O \neq \emptyset\} \tag{2}$$

In order to solve a motion planning problem, C_{obs} must be computed or approximated. In the case of a rigid body robot translating in a plane that contains polygonal obstacles, C_{obs} can be found by *growing* the obstacles. For instance, if the robot is a circle robot, all obstacles have each vertex extended to the size of the robot's radius. There is a more systematic way to compute how the obstacles grow if the robot is a convex polygon. The Minkowski Sum is computed for the set of robot and obstacle vertices:

$$A \oplus B = \{(a, b) | a \in A, b \in B\} \tag{3}$$

where A is the set of robot vertices and B is the set of obstacle vertices. The convex hull of $A \oplus B$ represents the final C-space obstacle region. To picture this algorithm visually, it is the result of sliding the robot along the obstacle's edges at the robot's reference vertex. For that reason, it is often referred to as the *sliding algorithm*. This sliding process was first proposed in [5]. The works [6] and [7] show how the sliding algorithm can be extended to 3 dimensions and manipulator models. The overall result is the same (obstacles are grown based on the robot size), however, the time to compute the grown obstacles is significantly increased because this algorithm has complexity $O(n + m)$ where n is the number of robot edges and m is the number of obstacle edges.

Path-planning problems run searching algorithms in a robot's C-space. Since motion planning problems need to obtain a trajectory, searching is done in a robot's Configuration-Time Space (CT-space). A robot's CT-space only differs from its C-space by adding a dimension to represent time.

III. Problem Formulation

Suppose a robot A is a rigid-body moving in some world space W. Let Q be the Configuration-Time Space of A as defined in the above section. Let Q_{obs} be the obstacle region occupied by obstacle(s) O in Q. The free region is defined as $Q_{free} = Q \backslash Q_{obs}$. A motion planning query would require the robot A to move from an initial state, q_S, to a goal state, q_G. Based on this description, a full robot motion planning problem has the following components:

- A: A rigid-body robot
- W: A world space that a robot moves in
- Q: The Configuration Space of robot A
- O: A description of obstacles in order to form Q_{obs} and Q_{free}.
- q_S: The initial state of A
- q_G: The goal state

These components can be grouped into a tuple: (A, W, Q, O, q_S, q_G).

A solution to the problem is a trajectory Γ that moves the robot from its initial state to the goal state while avoiding obstacles. Knot points for the trajectory can be generated by a path-planning algorithm that outputs a path $\rho[0, 1] \rightarrow Q_{free}$ such that $\rho(0) = q_S$ and $\rho(1) = q_G$. Or, the trajectory can be generated while a path is being generated, which is known as *Kinodynamic Planning* and will be covered in Section VIII.

This paper focuses on the planning aspect of motion planning because the essence of motion planning is in exploring a state space as rapidly as possible. Trajectory-following is closer to the field of *control* and is performed only after a state space has been searched to form a path.

IV. Combinatorial Methods

Combinatorial motion planning methods are complete planning algorithms that construct a roadmap in a known, continuous space in order to search through for path planning queries. These methods compute the exact configuration space for a robot and its environment in order to construct the roadmap. This class of algorithms works well for simple robotics problems, but does not scale up to many real-world applications seen today.

A. Roadmaps

Let G be a graph structure of a topological space that maps to a robot's configuration space, C_{free}. The set S is the points reachable in C_{free} by G. In order for G to be a roadmap, two conditions must be satisfied:

1) It is simple to compute a path from any initial node to any goal node in the space represented by G.
2) Given initial and goal configurations q_I and q_G, respectively, in C_{free}, it is possible to connect corresponding nodes s_1 and s_2 in S. More formally, if there exists a path $\tau : [0, 1] \rightarrow C_{free}$, such that $\tau(0) = q_I$ and $\tau(1) = q_G$, then there also exists a path $\tau'[0, 1] \rightarrow S$, such that $\tau'(0) = s_1$ and $\tau'(1) = s_2$, where s_1 and s_2 are nodes on the roadmap.

An extremely important implication of the second condition is that solutions to path-planning queries are not missed

due to the roadmap not capturing the connectivity of the configuration space. Essentially, the second condition ensures that algorithms searching on a roadmap can be complete. The first condition makes it always possible to connect an initial and goal configuration to a roadmap. This is essentially because the initial and goal nodes are not connected to the roadmap when it is constructed. They are only connected when beginning a query.

B. Cell Decomposition

The goal of cell decomposition is to partition a topological space into cells that lie entirely in C_{free}. From such a partitioning, a roadmap can be constructed in order to reduce the path planning problem to a graph-search problem. There are several choices for decomposition. Some methods are best suited for polygonal spaces, some are better suited for higher-dimensions. However, all cell decompositions must satisfy three properties:

1) A path from any point in a cell to any point in a separate cell should lie in C_{free}.
2) Adjacency information is easy to extract.
3) Given any configuration q, calculating which cell q lies in should be simple.

The first condition is the most important. If it is known that the robot can move from one cell to another without encountering C_{obs}, then the problem is reduced to finding a path among a small number of cells, i.e. a graph-search problem. The second and third conditions are necessary to make the querying fast.

1) Complexes: A *complex* is defined as a collection of cells and their boundaries. The difference between the terms *cell decomposition* and *complex* is that complexes contain additional pieces of information about how cells fit together, whereas cell decompositions only contain the actual cells themselves. Cell decompositions are derived from complexes.

The type of complex for a space will dictate the cell decomposition method that can be derived for the space. Two types of complexes are described below.

A *simplicial complex* satisfies the following conditions:

1) Any face of the complex is also in the complex.
2) The intersection of any two simplexes in the complex is a common face of both simplexes

Essentially, all of cells must be connected and share boundaries. Visually, this results in a set of cells that fit together nicely, as in Figure **??**.

A *singular complex* is a generalized version of simplicial complexes that can be defined on a manifold of $\mathbb{R}^n$. Singular complexes also satisfy two conditions:

1) For each dimension k, the set $S_k \subseteq X$ must be closed.
2) Each k-cell is an open set in the topological subspace S_k. Note that 0-cells are open in S_0, even though they are usually closed in X.

2) Approaches: Vertical Cell Decomposition [**?**] is perhaps the most common method for cell decomposition in 2D. The method partitions the free C-space into triangles and trapezoids. These simple shapes represent the cells of C_{free}.

At each vertex in C_{obs}, vertical rays are extended above and below the vertex until another points in C_{obs} is reached. These rays will define edges for the polygonal cells that partition C_{free}. Refer to Figure **??** for a visual description of the method. One can see that the polygons creating C_{obs} can be either convex or concave and that the cells in C_{free} will be either triangular or trapezoidal.

Building a roadmap from the partitioning amounts to choosing sample points from each cell and connecting them to edges of the cell. More formally, a roadmap $G(V, E)$ is defined as follows: For every cell, C_i, define an interior point q_i. For each vertical edge in C_i, define a point s_i. Add the points q_i and s_i to V. Then, create an edge, e_i, between q_i and the sample points on its cell's vertical edges. Add e_i to E. The resulting graph is a roadmap that can be used for efficient path planning.

The running time of this algorithm is $O(nlogn)$ when using the *line-sweep* principle. This principle is the idea of a line sweeping across the space and noting where there are interesting points in the space. In this application, the line is vertical and the interesting points are the vertices of obstacles.

In Figure **??**, the vertical lines show where the line would find interesting points and stop. An *event* is each time the line finds an interesting point. Figure **??** shows the status of the list of edges after each event.

Triangulation [**?**] is an alternative to vertical decomposition for polygonal spaces. The cells formed from triangulation will be, discernibly, triangles, whereas the cells formed from vertical decomposition are a mix of triangles and trapezoids. Refer to Figure **??** and note the difference between the partitioning and the partitioning in Figure **??**. The running time of a naive approach to triangulation is $O(n^3)$. That is far too long, even for some 3D spaces. However, the running time can be reduced to $O(n^2logn)$ with radial sweeping [**?**].

Cylindrical decomposition [**?**] is similar to vertical decomposition in that it works by extending rays from all the obstacle vertices, but in the cylindrical method the rays do not stop until they reach the boundary of the C-space. This results in cells that alternate between regions of C_{free} and C_{obs}.

3) 3D Decomposition: Cell decomposition in 3 (or more) dimensions is desirable since most motion planning problems have a C-space of more than 2 dimensions. The methods discussed previously work well for 2D, but only cylindrical decomposition generalizes well to higher dimensions. Vertical decomposition can be recursively applied to k-cells of higher dimensions, but it only succeeds if C_{obs} is represented as a semi-algebraic model with linear primitives. That format for C_{obs} is rare for a configuration space since many high-DOF robot models contain revolute joints.

4) Querying: To solve a path planning query using the roadmap generated by a cell decomposition method, the initial and goal configurations, q_I and q_G respectively, need

to be added to the roadmap. The third condition for cell decomposition listed in the previous section comes into play here. It should be simple to determine which cells q_I and q_G are in. Let C_I and C_G denote the cells that contain q_I and q_G, respectively. Once C_0 and C_G are identified, edges are found between the configurations and the corresponding cell's sample point. Thus, an edge is formed between q_I and the sample point of C_I. Similarly, an edge is formed between q_G and the sample point of C_G. Both edges are added to the roadmap $G(V, E)$. Graph search algorithms can be applied to the roadmap to find a path from q_I and q_G or find if a path does not exist.

C. Combinatorial Motion Planning Complexity

The crux of combinatorial methods is that the upper bound on the number of possible paths in a roadmap scales factiorally with the number of edges and vertices in a graph.

Consider the simpler problem of a robot moving in a directed acyclic grid with dimensions $m \times n$ and each edge is the same length. If the robot is at an inital location (s_x, s_y) and the goal is given as (g_x, g_y), then the total number of solutions is

$$|P| = \binom{|g_x - s_x| + |g_y - s_y|}{|g_x - s_x|} \tag{4}$$

Given (4), the total number of paths to move from one corner to the opposite corner can be computer as:

$$|P| = \binom{m+n}{m} \tag{5}$$

These equations assume that the robot would never have a cyclic path. This assumption may hold for path planning in static environments, but it is one that cannot be made for the more general problem of path planning in dynamic environments. A robot may need to move back to a previous position due to the environment changing or the robot made need a path with a new direction to avoid an obstacle. Thus, the number of possible paths for a real-time motion planning system would be even more than the directed acyclic graph case. While it is good to understand the theoretical components of these algorithms, it would be far too time-consuming to run any combinatorial method on a real-time motion planning system.

V. Artificial Potential Fields

In an artificial potential field approach, a robot is treated like a particle in a gradient field. The robot's goal emits an attractive force that acts on the robot and the obstacles emit repulsive forces. The combination of forces results in a vector field leading the robot from its initial state to the goal state. Figure **??** illustrates an example.

At each location, a robot's motion is dictated by the magnitude and direction of the vector at the location.

The Artificial Potential Fields approach can react to changes in an environment quickly by simply editing the function coefficients.

VI. Elastic Strips

Elastic Strips [8] is an approach to real-time motion planning in unstructured dynamic environments. It aims at generating motion for robots with high degrees of freedom, such as humanoid robot models. It utilizes the operational space formulation [9] that decomposes robot control into different behaviors in order to produce motion that allows a robot to avoid obstacles while also achieving a task.

An elastic strip is a volume formed by the union of a set of homotopic paths. These paths are slight modifications to a candidate path that satisfies global constraints of a robot's task, such as moving to a goal state. The volume is "elastic" because it can be slightly modified by modifying one of the homotopic paths while maintaining topological properties of the candidate path and local constraints such as obstacle avoidance.

A. Obstacle Avoidance

The Elastic Strips framework employs local potential fields to modify trajectories in response to changes in the environment.

Let P_c be the candidate path for a robot. To avoid obstacles, a new candidate path, P_c', must be formed by modifying P_c so that P_c' lies within the elastic tunnel. A path is represented as a discrete set of configurations. To modify a path, two work space forces are created from two potential fields and applied to the path's configurations. The two forces are $V_{external}$ and $V_{internal}$. The external force is based on the robot's proximity to obstacles. For a point p on the robot, V_{ext} is defined as

$$\begin{cases} \frac{1}{2} k_r (d_0 - d(p))^2 & \text{if} d(p) < d_0 \\ 0 & \text{otherwise} \end{cases}$$

where $d(p)$ is the distance between p and the closet obstacle, d_0 is the region of influence around obstacles, and k_r is a scalar. The resulting force from V_{ext} is

$$\mathbf{F}_p^{ext} = -\nabla V_{ext} = k_r (d_0 - d(p)) \frac{d}{||d||} \tag{6}$$

where d is the vector between p and the closest point on an obstacle. This repulsive force, $\mathbf{F}_p^{rep}$, will modify the existing configurations by pushing them away from obstacles. If the robot is not close to an obstacle, then the repulsive force has no effect.

If an obstacle moves away from the robot after applying a repulsive force, then the robot's path should contract in order to make the path shorter. This is achieved by an internal force generated by virtual springs at each of the robot's control points along a path. This force is meant to bring each configuration closer to its adjacent configurations on a path. The internal force acting at control point p_j^i on the jth link at configuration q_i connected to configurations q_{i-1} and q_{i+1} is defined as

$$\mathbf{F}_{i,j}^{int} = k_c \left(\frac{d_j^{i-1}}{d_j^{i-1} + d_j^i} (p_j^{i+1} - p_j^{i-1}) - (p_j^i - p_j^{i-1}) \right) \tag{7}$$

There are two scenarios that result in a failure to find a new path. One scenario is when a topological change occurs in the robot's configuration free space. Because the modification method maintains the topological properties of the candidate path, there will not be a successful modification if a topological change occurs.

The second scenario is when a structural local minimum occurs. The configuration free space around the candidate path has become so narrow that it cannot be modified enough to avoid an obstacle.

VII. Sampling Based Methods

Computing the exact configuration space in high dimensions becomes extremely expensive. This makes previous methods, such as potential fields and combinatorial roadmaps, inapplicable to motion planning problems of high dimensions, such as a 6-DOF manipulation problem. Sampling-based methods were introduced in order to handle motion planning problems with such high complexity. Sampling algorithms sacrifice completeness and optimality, but obtain higher performance results.

Sampling based methods do not access the C-space directly. An associated collision detection algorithm will test a the robot at a sampled configuration with obstacles in the environment. Based on the result of collision detection, the sampling algorithm knows if a sample is free or obstacle space. After sampling many points, the result is an approximation of the C-space, rather than a complete description.

Sampling has become the method of choice for most modern robotics applications. A sampling based method was first proposed in 1996 [1] that builds a graph representation of the C-space approximation and then uses graph-search techniques to find a final path. A tree-based approach was introduced shortly after that could take kinodynamic constraints into account [2]. These two approaches have been extended and iterated upon many times and remain the top approaches in the field.

A. Probabilistic Roadmaps

The basic Probabilistic Roadmaps algorithm [1] works in two phases: learning and querying.

1) Learning Phase: The goal of the *learning* phase is to build the C-space approximation. An overview of the learning phase is summarized below.

The goal is to build an undirected graph containing N nodes. These nodes correspond the collision-free points in a robot's configuration space. When a point is sampled and determined to be collision-free, a neighborhood of points from the existing graph is found. For each of these neighborhood points, if a local planner can find a collision-free path from the sampled point to the neighborhood points, an edge is formed connecting the nodes.If a sampled node is found to be in collision, the node is thrown away and the algorithm samples again.

A local planner is utilized in order to create edges within the graph. Designing a local planner can be difficult, however, because it still needs to plan in a high-dimensional space. Cheap and fast planners are possible since we do not need to consider a global objective. Even if fast local planners are not complete, they are desired over more powerful methods because they allow for more connections to be considered during the learning phase. They allow this because if a local planner is fast, then checking connections between many nodes is fast. Thus, the faster a local planner is, the more connections can be made in a small amount of time. It is desirable to sample as many nodes and make as many connections as possible since more nodes and connections will generally lead to a more accurate C-space approximation. For these reasons, local planners that connect nodes via straight lines are commonly used.

Algorithm 1 PRM Learning Phase

```
1:  N : number of nodes to include in the roadmap
2:  V ← ∅ // Initialize set of vertices
3:  E ← ∅ // Initialize set of edges
4:  i ← 0
5:  while i < N do
6:      q ← randomly sampled configuration
7:      if q ∈ C_free then
8:          V ← V ∪ {q};
9:          V_c ← neighborhood set of q;
10:         for each v ∈ V_c do
11:             if Can_Connect(v, q) then
12:                 e ←new edge(q, v)
13:                 E ← E ∪ {e}
14:             end if
15:         end for
16:     end if
17: end while
```

Determining the neighborhood of a node is based on the value of a distance function between two nodes. The distance function is closely related to the choice of local planner because the distance should be proportional to the chance that the local planner can connect the two nodes with a collision-free path. For straight line local planners, the distance function can be the Euclidean distance between the nodes' positions in the robot's workspace. For more complicated local planners, however, more involved distance functions must be defined.

Narrow passages are a critical concern during the learning phase. A narrow passage is a region that is mostly enclosed by obstacles. These regions are of concern because it is difficult to sample inside of them and to find connections to nodes inside the regions. If a robot must pass through a narrow passage to reach its goal, then the learning phase may be prolonged to the point of being impractical. In Figure **??**(a), the free space between the two obstacles could be considered a narrow passage. With uniform sampling, there is a small chance that a node in the passage will be found.

The PRM method tries to address narrow passages by performing an *expansion* step after constructing the graph, in order to improve the graph's connectivity. To accomplish that, nodes that lie in narrow passages are selected and an

attempt to find neighboring points in C_f is made.

The difficulties in the expansion step is to identify which nodes lie in a narrow passage (those are the nodes that need expansion) and how to find neighboring points.

To find potential expansion nodes, a heuristic needs to be defined that predicts the likelihood of some node c being unconnected. In [1], the following was used:

$$r_f(c) = \frac{f(c)}{n(c)+1} \tag{8}$$

where $n(c)$ is the number of times the planner tried to connect node c and $f(c)$ is the number of successful connects made to c. This function is the *failure ratio* of node c.

Random walks are a common method for actually expanding a node. A direction is chosen at random and the direction is followed until an obstacle is reached. An edge is then formed between node c and a new node n that is a node somewhere in between c and the obstacle that was reached in the chosen direction. This process repeats by selecting new random directions for a pre-specified amount of time.

2) Query Phase: The *query* phase searches the roadmap constructed during the learning phase to find a path from an initial configuration to a goal configuration.

The first step in this phase is to connect the starting configuration, s, and goal configuration, g, to the roadmap. The local planner used in the learning phase is used for this task. The procedure for connecting s to the roadmap, R, begins by identifying the closest node on R to s. The local planner attempts to form a path between s and the closest node. If it cannot find a path, the planner uses the next-closest node on R to connect to s. This process repeats until a connection is successful. Once s has been connected, the goal configuration g is connected in the same way. Once both s and g are connected to R, any graph-search algorithm will return a path from s to g in the robot's configuration space.

3) Further Discussion: The PRM planner is considered *probabilistically* complete. In most cases, it will not explore every point in the C-space. Given enough time, it will explore all points in the C-space and be able to determine if a path does not exist.

Dynamic environments are difficult to address with the PRM approach. Constructing the roadmap can take significant computation time and is only useful for the state of the environment while constructing the roadmap. If any obstacles move, then the roadmap needs to be modified or completely remade based on the new state of the environment.

B. Rapidly-Exploring Random Trees

Rapidly-exploring Random Trees (RRT) is a popular sampling based approach to planning in high dimensional spaces [2]. The key difference between RRT and PRM is that RRT uses a tree structure to approximate a robot's C-space, rather than the undirected graph structure used in PRM. This approach leads to several attractive features: it is suitable for handling non-holonomic and kinodynamic constraints, it requires few parameters, and is more apt to handle real-time planning in dynamic environments. RRT was specifically designed for kinodynamic planning (discussed more in Section VI), i.e. it considers both configurations and differential constraints. For this reason, it is also referred to as a randomized kinodynamic planner [10].

The kinodynamic state space contains both the set of the configurations and the set of possible velocities. A single state would represent the robot's velocity at a particular configuration. Let Q denote the configuration space of a rigid or articulated object in space. Each state $q \in Q$ represents a single configuration of the object. For kinodynamic planning, a state $x \in X$ would be defined as $x = (q, \dot{q})$ and X is the set of all such states. Other types of spaces are viable, however. The state space could include higher-order derivatives or only the configuration q. If RRT is considering differential constraints, then the output will be a trajectory as opposed to only a geometric path.

A state transition function in the form $\dot{x} = f(x, u)$ must be defined to consider kinodynamic constraints. The inputs are a state, x, and an input, u, and the output is the derivative of the state over time. This function can be integrated in order to find the next state resulting from applying u to x. This allows the algorithm to consider dynamics constraints while planning.

Pseudocode for building an RRT can be seen below:

Algorithm 2 RRT Overview

```
1:  function CONSTRUCT RRT(x_init, x_goal, Δt)
2:      T.init(x_init)
3:      while x_goal ∉ T do
4:          x_rand = newly sampled state
5:          x_near = nearest neighbor of x_rand
6:          u = input needed to move from x_near to x_rand
7:          x_new = {x_near, u, Δt}
8:          T.add_vertex(x_new)
9:          T.add_edge(x_near, x_new, u)
10:     end while
11:     Return T
12: end function
```

At each iteration, a new state, x_{rand}, is randomly sampled. The parent of x_{rand} will be the vertex on the tree that is nearest to the sample. The control input, u, is then selected to move from x_{near} to x_{rand} without moving into any regions occupied by obstacles in the state space. The new state, x_{new} is formed from the nearest neighbor, the control input, and a small time increment. The new node is added as a vertex on the tree and the edge connecting x_{near} to x_{new} by applying u is added to the tree as a new edge. This process repeats until the goal state has been added to the tree. Since every state has only one parent on the RRT, adding the goal state to the tree means that a path has been found.

The RRT approach has several nice properties. Similar to PRM, the RRT algorithm is probabilistically complete. The chance of determining whether or not a path exists increases as times goes on. As previously discussed, RRT

is able to consider kinodynamic constraints. This is largely due to using a tree over an undirected graph. Since each state will have only one directed connection, it is easy to ensure that a robot's dynamic constraints will be satisfied as it move from configuration to configuration. Another attractive property is that the tree will always remain connected, even in narrow passages. Graphs constructed with the PRM are not guaranteed to remain connected because the connections are based on a vertex's neighborhood. If there are no existing vertices nearby the new vertex, then the vertex will have no connections, but will still be added to the graph.

One of the most appealing aspects of RRT is that it is bias towards unexplored areas of a space. This occurs because the selection of new states is based on the nearest neighbor calculation. As RRT explores a space, the regions separated by the tree's edges become close to voroni regions of the space.

Bi-directional searching was presented in [**?**] to significantly increase the performance of RRT. Two trees are instantiated - one at the initial state x_{init} and one at the goal state x_{goal}. When the trees connect, the query is complete. At each iteration of the algorithm, one of the trees, T_0, is extended by the usual set of procedures in RRT. Let x_{new} be the newly added state to T_0. A new procedure, $Connect$, is called to attempt to add x_{new} to the second tree, T_1. If this connection is unsucessful, then the trees are swapped so that T_1 can be extended.

After its initial presentation, significant efforts were made to improve the performance of RRT for real robot execution [11]. This work proposed the Extended RRT (ERRT) algorithm in order to use RRT on a real robot in a dynamic environment. To use RRT in a dynamic environment, an entirely new tree must be created each time the robot needs to change trajectories to adapt to its environment. Using the intuition that prior trees contain states that may lead to the goal, the authors used a waypoint cache to reduce the iterations needed to find the goal. A second addition from this work was an adaptive beta search, which adjusts the distance function to adaptively change the distance between nodes. Simulation experiments show that this approach can significantly reduce planning time when the queries become moderately complicated to solve. This work was also implemented onto the RoboCup F180 robot system, which was the first time RRT was shown to work on a real robot. This system read the positions of static dynamic obstacle through vision data at discrete time steps and used this information when executing the ERRT algorithm. The goal was to plan quickly enough to move around the obstacles as they executed their own independent motion. The authors found that the ERRT algorithm performed poorly when considering continuity in position in velocity. After reverting to planning with no kinematic constraints, ERRT was able to execute quickly enough for real-time motion, but contained noticeably sub-optimal motion. A post-processing step was employed that smoothed out the resulting path well enough for the robots to move at speeds up to 1.7m/s.

C. Discussion

Both sampling-based approaches sample stochastically in order to build an approximation of a C-space. They differ in the data structure used as the C-space approximation. Because each approach builds an approximation of the C-space, the basic versions are not able to find an optimal path to the goal. Several extensions have been made to these approaches to improve the optimality of the resulting paths.

For PRM, most of the immediate work following [1] focused on the learning phase of PRM to improve its ability to capture narrow passageways in the C-space. These extensions target one of two aspects: the node generation process or the process of finding a neighborhood of candidate connections. Two improvements to the node generation process are still used today [12], [13].

Obstacle Based PRM (OBPRM) [12] generates nodes based on obstacle surfaces. Thus, the obstacle information guides the node generation process. It can generate both contact configurations (for manipulation planning) and configurations in free space for general path-planning. Each sampled state is translated towards an obstacle until collision is made, and then the configuration is recorded as c_{in}. A random direction is selected to translate c_{in} on until a free configuration, c_{out}, is found. The contact configuration between c_{in} and c_{out} can be found through binary search. Medial Axis PRM (MAPRM) [13] proposes a node generation strategy based on the medial axis of a space. Each new sampled state is retracted onto the medial axis of the robot's configuration-free space. The exact medial axis does not need to computed with this approach. Each sampled configuration is moved towards its nearest neighbor on the graph via bisection until there is no unique closest node on the graph. Both of these strategies help the planner deal with narrow passageways by generating nodes close to obstacles (OBPRM) and generating nodes between obstacles (MAPRM).

A visilibity roadmap approach [14] significantly reduces the number of nodes by only keeping nodes that either connect two connected components or are not visible by any "guard" nodes in the roadmap. A node is visible by another node if they can be connected with a straight-line that lies entirely in the free portion of a robot's C-space. Guard nodes are ones that are not visible by other guard nodes. Other strategies used for this

Using these approaches to find an optimal path was explored in [15].

VIII. Kinodynamic Motion Planning

Kinodynamic Planning is an approach to robot motion planning that integrates the kinematic and dynamic constraints of a robot into the planning aspect of the solution. In other words, it solves both path-planning and trajectory-following simultaneously. Prior motion planning approaches required these two problems to be done separately. Path-planning would be performed without considering a robot's dynamic constraints, and then a trajectory-following algorithm would interpolate points to move the robot along

the path. The tuple describing a general motion planning problem from Section III will contain two new values: v_{max} and a_{max}, the maximum velocity and maximum acceleration, respectively, of the robot.

Since the goal of planning is usually to find an optimal solution (or the most-optimal available solution), it is possible that a kinodynamic planner may lead to an unsafe trajectory based on the robot's dynamic constraints. Therefore, a safety margin can be used to ensure that a solution maintains a certain distance from obstacles. A safety margin, δ, is defined based on a robot's dynamic constraints. A δ_v-*safe kinodynamic solution* will move a robot from a starting state (position and velocity) to a goal state while avoiding obstacles by a safety margin δ_v. Mathematically, the effect of the safety margin is the following:

$$\delta_v(c_0, c_1)(\dot{p}(t)) = c_0 + c_1||\dot{p}(t)|| \tag{9}$$

where c_0 and c_1 are positive scalar values.

An additional parameter to this problem is l, the side length of a cube that bounds all free condigurations. The complete tuple for a kinodynamic planning problem is $(A, W, Q, O, q_S, q_G, l, a_{max}, v_{max})$. The complete tuple for a δ_v-*safe kinodynamic planning* problem is $(A, W, Q, O, q_S, q_G, l, a_{max}, v_{max}, c_0, c_1)$. The values a_{max}, v_{max}, l, c_0, and c_1 are *kinodynamic bounds*.

The problem was first proposed in [16], where an approximate solution was presented based on graph-search. The state space for this approach, denoted as TC, is the phase space of a robot's C-space. A point in this space represents both the robot's position and velocity, e.g. $S = (s, \dot{s})$. The state space is transformed into a directed graph where the vertices are discretized values from TC and the edges represent trajectory segments.

The trajectory segments are formed from (a, τ)-*bangs*. Given a maximum acceleration, a, let the set of constant acceleration values $[-a, a]$ be denoted as A. A timestep τ can be chosen such that the velocity bound v_{max} is an integer multiple of $a\tau$. A (a, τ)-*bang* is the result of applying an acceleration value from A for duration τ.

The timestep τ is based on a function of a_{max}, v_{max}, an approximation parameter ϵ, and scalar values c_0 and c_1. The scalar values define a safety margin and the approximation parameter defines a bound for closeness to the desired start and goal states. The chosen timestep is the largest value satisfying the following condition:

$$\tau \leq \frac{\epsilon}{13} min\left(\sqrt{\frac{2c_0\epsilon}{a(c_1+1)}}, \frac{c_0\epsilon}{a(c_1+1)}\right) \tag{10}$$

Given τ, the initial state $S^* = (s, \dot{s})$ is chosen such that $s^* = s$ and $\dot{s}^*$ is the multiple of $a\tau$ closest to $\frac{\dot{s}}{1+\epsilon}$.

Breadth-first Search is then applied to the embedded graph in TC. The initial state is S^* and the goal state is any vertex within a certain threshold of the desired goal state, e.g. any vertex within $\left(\frac{a\tau^2}{2}, \frac{a\tau}{2}\right)$ of $\left(g, \frac{\dot{g}}{1+\epsilon}\right)$. Safety margin constraints are checked for each trajectory segment that the search algorithm considers.

A theorem is presented in this work to show that the complexity of the algorithm. Given the kinodynamic bounds of a problem, the upper bound for this algorithm's running time is

$$O\left(n\left[\frac{lv\gamma^3}{\epsilon^6}\right]^d\right) \tag{11}$$

where $d = 2, 3$, n is the number of bounding halfplanes on all obstacles, and

$$\gamma = max\left(\frac{a_{max}(c_1+1)}{c_0}, \sqrt{\frac{a_{max}(c_1+1)}{2c_0}}, \frac{a_{max}}{v_{max}}\right). \tag{12}$$

This work can be used for kinodynamic planning of a rigid robot translating in a 3D space that contains convex polyhedra. This initial work is sufficient for problems with low degrees of freedom, but could not be shown to be a feasible approach to more complicated problems. Randomized approaches were proposed to handle kinodynamic planning problems with high degrees of freedom.

An RRT approach was proposed in [10]. RRT lends itself to kinodynamic planning very easily because part of forming the connections in RRT is based on the control needed to move from node to node. The main difference is to change the state space from the robot's C-space to the phase-space of the C-space so that each state is a (*position*, *velocity*) pair.

Differential constraints, such as the non-holonomic constraint, are represented as the implicit function

$$\dot{x} = f(x, u) \tag{13}$$

where $u \in U$ and U is the set of all inputs to a system. A numerical approximation for (13) is needed to find the resulting state from applying u to x. The work in [10] uses fourth-order Runge-Kutta integration, but a similar numerical approximation technique will suffice.

The randomized kinodynamic planner based on RRT assumes that an environment is static. It extends the notion of an obstacle region in a state space to be a region of inevitable collision. This region contains all the states where the robot is either in collision or at a state where it will not be able to avoid an obstacle due to its dynamics. The region of inevitable collision subsumes the obstacle region in a state space.

This approach has been shown to work for several systems. For each system, the dynamic model must be defined as well as a distance metric in order to approximate a nearest neighbor for each node. The most complicated system described in experimental results is a three-dimensional spacecraft with a 12-dimensional state space. A bidirectional search method can also be applied to this work as in [17].

The RRT approach did not consider dynamic obstacles, however. Dynamic obstacles in kinodynamic planning were

considered in an PRM approach [18]. This work plans in the state×time space of a robot by sampling to build an approximation of the search space. A vision module runs simultaneously with the planner to estimate obstacle motion.

Let S denote the state space of a robot. S can be the Configuration space of a robot, or it can include additional parameters. In the case of a car-like robot, a state $s \in S$ could be (x, y, θ) or it could also include velocity values, i.e. $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$.

Let ST denote the state×time space defined as $S \times [0, +\infty)$. Given an initial state×time pair (s_0, t_0) and a goal pair (s_g, t_g), the output of this planner is a function $u : [t_0, t_g] \rightarrow \Omega$ that leads to a collision-free trajectory $\tau : t \in [t_0, t_g] \rightarrow \tau(t) = (s(t), t)$ such that $s(t_0) = s_0$ and $s(t_g) = s_g$.

A query is solved by iteratively building a tree-shaped roadmap T whose root is the initial state×time pair through sampling until the goal is connected to the tree. Each iteration randomly selects an existing node on the roadmap, (s, t), a time $t' \leq t_g$, and a control u. Then, the control input u is applied to (s, t) for time t'. This will form a trajectory in the state×time space. If this trajectory lies in free space, then the endpoint (s', t') is added as a node in the roadmap. The trajectory itself is stored as a directed edge between (s, t) and (s', t') and u is also stored with the new edge. The algorithm exits when a trajectory leads to a new state that is in a neighborhood of the goal state. If some number of iterations N occur and a solution has not been found, then the algorithm will return a failure to find a trajectory.

This approach was applied to two robot systems. The first system was a pair of non-holonomic bases that must remain within a minimum and maximum distance of each other. The state space of this system had 6 dimensions: $(x_1, y_1, \theta_1, x_2, y_2, \theta_2)$. The second system was an air-cushioned robot that moves frictionlessly on a flat table. The state space of this system had 4 dimensions: $(x, y, \dot{x}, \dot{y})$. When adding the time dimension to the state spaces, the algorithm built a roadmap in 7 and 5 dimensions, respectively.

In simulation, obstacle motion is given to the planner before executing the algorithm. The results verify that the planner is able to find a collision-free trajectory in the precense of moving obstacles for the majority of the time over the three cases shown. In experiments with real robots, obstacle motion is captured with an overhead camera. Obstacle motion is passed to the planner before executing and the planner assumes constant velocity for each obstacle. If the obstacle velocities significantly change, the camera will alert the planner with new information so that it can replan a new trajectory with new obstacle trajectories.

Several real robot experiments show that the planner was able to navigate the robot around moving obstacles when obstacle motion is known beforehand.

IX. Real-time Adaptive Motion Planning (RAMP)

Real-time Adaptive Motion Planning (RAMP) [4] is a framework that utilizes evolutionary computation for real-time planning and execution in dynamic environments.

Evolutionary computation is a field designed to leverage sophisticated data structures with genetic algorithms to search through a state space [19]. They are particularly suited to approximate optimization problems. One of the key difficulties in robot motion planning is extremely large state spaces. Sampling-based methods are able to find a path in large state spaces, but they can be far from optimal and re-sampling is expensive for real-time planning. An evolutionary computation approach allows us to change our set of samples very quickly and incorporate optimization into our state spaces searches. Using evolutionary computation for path planning was first proposed in [3] and has since been extended to the motion planning framework RAMP.

A. Modification

A population of trajectories is maintained throughout the entire run time. Each trajectory in the population is a chromosome. At each *generation*, 1-2 randomly selected chromosomes are modified via genetic operators. The new chromosome(s) may replace other trajectories in the population.

The basic modification operators are described in [3]. They are *Insert*, *Delete*, *Swap*, *Change*, and *Crossover*. These modification operations are computationally inexpensive so it is possible to perform 100+ within a second. The operators can result in drastic changes to a path. This means that a new feasible path may be found in a very small amount of time, which facilitates real-time path planning that can react to changes in an environment. Modification operators can be added, removed, or replaced based on the application of the robot. For instance, a *Stop* operator was introduced in [4] that stops the base of a mobile manipulator for a random amount of time. This operator promoted decoupling of the base and manipulator, which was desirable to exploit redundancy.

The framework interweaves three separate procedures called planning cycles, control cycles, and sensing cycles. At each planning cycle (or *generation*), the population of trajectories is modified. In later implementations (***Cite my own paper, what if not published?***), motion error is incorporated into the population during a planning cycle. The planning cycles occur as frequently as possible. Control cycles are procedures that select a trajectory from the population that will be executed by the robot and update the starting motion state of each trajectory in the population. The frequency of control cycles is adaptive based on constraints needed to satisfy in order to switch trajectories. Sensing cycles are procedures that update information about the environment.

B. Evaluation

Evolutionary computation utilizes an evaluation function to select the best chromosome from a population. In the RAMP framework, this corresponds to evaluating the population to choose the trajectory for a robot to execute. In RAMP, feasible trajectories are evaluated differently than infeasible trajectories. A trajectory's feasibility is determined

by collision with obstacles and if all constraints can be satisfied when switching to the trajectory.

A feasible trajectory is evaluated by a weighted, normalized sum based on optimization criteria:

$$Cost = \sum_{i=1}^{N} C_i \frac{V_i}{\alpha_i} \tag{14}$$

where C_i is a weight, V_i is a value for an optimization criterion, and α_i is a normalization value. Common criteria are time costs, energy costs, orientation change, and manipulability costs.

Infeasible trajectories are trajectories that are in collision or cannot be executed due to not satisfying constraints. Rather than optimization criteria, they are evaluated based on penalties. For a mobile robot system, the following would be an evaluation function for infeasible trajectories:

$$Cost = P_T + P_\theta \tag{15}$$

$$P_T = \frac{Q_T}{T_{coll}} \tag{16}$$

$$P_\theta = Q_\theta \frac{\Delta\theta}{M} \tag{17}$$

where Q_T and Q are large constant values, T_{coll} is the time of the first collision in the trajectory, $\Delta\theta$ is the orientation change, and M is a normalization term.

The evaluation for infeasible trajectories should be designed to differentiate which infeasible trajectories are better than others. For instance, the term P_T ensures that trajectories with collision much later in time will be evaluated as better than trajectories with collision occurring very soon.

C. Sensing

Sensing cycles update the latest environment changes for the evaluation function. For dynamic obstacles, simple trajectories are predicted to use for collision detection in CT-space. An obstacle trajectory is predicted by assuming the sensed velocity is constant. Therefore, the predicted trajectory will be either a straight-line or circular arc. These are used for collision detection when evaluating the population. The obstacles may not follow what RAMP predicts, but future changes will be captured in future sensing cycles. Since sensing cycles occur frequently, using simple trajectories is sufficient for navigation, as shown through various experiments.

X. Collision Detection

Collision detection is the problem of determining whether one or more objects in a virtual environment are in contact.

Collision detection is the most computationally expensive task in robot motion planning.

XI. Conclusion

This is my Conclusion.

References

[1] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[2] S. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, C.S. Dept., Iowa State Univ., 1998.

[3] J. Xiao, Z. Michalewicz, L. Zhang, and K. Trojanowski. Adaptive evolutionary planner/navigator for mobile robots. *IEEE Transactions on Evolutionary Computation*, 1(1):18–28, 1997.

[4] J. Vannoy and J. Xiao. Real-time adaptive motion planning (ramp) of mobile manipulators in dynamic environments with unforeseen changes. *IEEE Trans. Robot.*, 24(5):1199–1212, 2008.

[5] Tomás Lozano-Pérez and Michael A Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.

[6] Tomas Lozano-Perez. Automatic planning of manipulator transfer movements. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(10):681–698, 1981.

[7] Tomas Lozano-Perez. Spatial planning: A configuration space approach. *Computers, IEEE Transactions on*, 100(2):108–120, 1983.

[8] Oliver Brock and Oussama Khatib. Elastic strips: A framework for motion generation in human environments. *The International Journal of Robotics Research*, 21(12):1031–1052, 2002.

[9] Oussama Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation*, 3(1):43–53, 1987.

[10] Steven M LaValle and James J Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[11] James Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2383–2388. IEEE, 2002.

[12] Nancy M. Amato, O. Burchan Bayazit, Lucia K. Dale, Christopher Jones, and Daniel Vallejo. Obprm: An obstacle-based prm for 3d workspaces, 1998.

[13] Steven A Wilmarth, Nancy M Amato, and Peter F Stiller. Maprm: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1024–1031. IEEE, 1999.

[14] Carole Nissoux, Thierry Siméon, and J-P Laumond. Visibility based probabilistic roadmaps. In *Intelligent Robots and Systems, 1999. IROS'99. Proceedings. 1999 IEEE/RSJ International Conference on*, volume 3, pages 1316–1321. IEEE, 1999.

[15] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[16] Bruce Donald, Patrick Xavier, John Canny, and John Reif. Kinodynamic motion planning. *Journal of the ACM (JACM)*, 40(5):1048–1066, 1993.

[17] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[18] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *The International Journal of Robotics Research*, 21(3):233–255, 2002.

[19] Z. Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media, 2013.