RICE UNIVERSITY

Replanning: A Powerful Planning Strategy for Systems with Differential Constraints

by

Konstantinos I. Tsianos

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

Houston, Texas

May 2008

Abstract

Motion planning is a fundamental problem in robotics. When the differential constraints of a real robot are also modelled, the produced motions can be more realistic and make better use of the hardware's capabilities. While it is not known if planning under differential constraints is a decidable problem and complete tractable algorithms are not available, sampling-based planners have been very successful at solving such problems. This thesis describes an online planning strategy which uses a sampling-based planner in a loop. Solutions for problems in static and known environments are computed incrementally through consecutive replanning steps. The robot is guided to the goal by a navigation function which is constantly updated to ensure the robot can explore all of its workspace. Experiments on various systems and workspaces show that the proposed approach can solve harder problems and produce paths of shorter duration compared to state-of-the art offline planners using only bounded memory.

Acknowledgements

This work has been supported by NSF IIS 0308237,NSF IIS 0713623 and NSF CNS 0615328. Experiments on equipment were supported by CNS 0454333 and CNS-0421109 ADA in partnership with Rice University, AMD and Cray.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

A fundamental problem in robotics has always been the computation of collision-free trajectories from an initial to a final robot configuration. In its most basic form, this is the *path planning* problem: a robot has to move in a static and known workspace populated with obstacles and reach a goal while avoiding all collisions. It is important to realize that even for a polygonal robot that moves in a space of polygonal obstacles, the complexity of path planning was proven to be *PSPACE-hard* [37]. Building on an already hard problem, the attention has shifted over the last two decades towards even harder planning problems by incorporating intrinsic limitations in motion that real robots have. To describe such motion constraints many terms appear in the literature. *Non-holonomic constraints* is an older term referring to non-integrable velocity constraints [27]. An example would be a car that cannot move sideways [1]. The term *kinodynamic constraints* is more recent and widely used [30, 8]. It refers to second or higher order constraints. Such constraints exist

[1]This is a standard way of modelling cars.

because a mechanical motor can only provide limited amount of torque. As a result, robots have bounded maximum acceleration (a second-order term). The most general term up to date, which is also the focus of this thesis, is *planning with differential constraints* [30]. Notice that when differential constraints are considered, solving the path planning problem for a robot is no longer sufficient. If a collision-free path is naively computed, it may happen that the motion constraints will not allow the robot to execute it. The question now becomes how to compute a collision-free trajectory that is also *implementable* or *feasible*. This is referred to as the *motion planning* problem. The benefit of considering any form of differential constraints is that the resulting motions are much closer to reality, thus making an informed and more effective use of the real hardware's capabilities. Moreover, these motions typically look more realistic and coincide with what human intuition expects to see. The latter is an increasing demand in other areas besides robotics such as the largely increasing computer gaming industry. To push this direction even further, lately there have been promising attempts to use physics engines in planning, to better capture effects such as friction and gravity [24].

Unfortunately, the gains of planning with differential constraints come at the cost of increased complexity. Complexity bounds are not known for kinodynamic planning problems and in the general case, the problem may be undecidable. Complete algorithms exist only for path planning, but they are computationally intractable and hard to implement except for some simple cases. Driven by the difficulty to solve harder planning problems, an important step was made during the last two decades with the development of *sampling-based* algorithms [30]. These algorithms were successful because they combine computational efficiency, generality of targeted systems and a noteworthy simplicity of implementation. In particular, one family of such algorithms called *tree-based* planners has yielded some very promising results for hard cases of motion planning under differential constraints.

Despite recent successes however, sampling-based planners still face a number of difficulties and are not so popular for real robotic applications. Sampling-based planners were initially developed as offline planners and even today they are often treated as such. For high dimensional problems, even state-of-the art planners such as *Rapidly exploring Random Tree* (*RRT*) [31] and *Path Directed Subdivision Tree* (*PDST*) [25] tend to require large amounts of time and memory to compute a solution. This is more pronounced in problems with differential constraints where the set of allowable motions is limited. Moreover, when a solution is found, the paths to the goal tend to be very long and contain a lot of unnecessary maneuvers. Another challenge in all sampling-based planners is the ability to handle narrow passages that the robot has to go through. For path planning problems, there is extensive research for focusing on these difficult parts of the space (see [8]). There does not seem to exist as much work on detecting narrow passages for motion planning with differential constraints.

Inspired by the progress and potential of sampling-based planners, this thesis introduces a more effective way for solving hard motion planning problems under differential constraints while addressing some of the drawbacks mentioned above. The proposed approach suggests a planning strategy which employs the idea of "divide and conquer". The planning problem is broken down into planning cycles and a tree-based planner is used in a loop. In each cycle, the tree-based planner is called for a limited time in order to replan. While the robot is moving, the algorithm has to decide what the robot will do next, after it finishes executing its current motion. Within each cycle, our proposed algorithm produces a set of possible motion options (motion generation) and selects the best motion out of that set (motion selection). This continuous replanning process introduces several new questions that must be answered:

- How is a finite set of motion options acquired?
- How are the motion options evaluated in order to select the one to be executed?
- How can we ensure that the robot will be able to backtrack and search the whole space instead of getting stuck due to the strict time limitation on the planner?
- How can the robot move safely avoiding all Inevitable Collision States (ICS) - situations from where no motion option is available - which can arise when a robot with differential constraints moves using replanning?

All the questions above have been investigated and are discussed in Chapter 3 which details the algorithm. At this point, a brief description is provided. The finite set of motions can be extracted from the paths on the tree data structure maintained by the tree-based planner. As far as evaluating these motions, a navigation function is used due to its nice way of encoding a sense of direction towards the goal. Moreover, by updating the navigation function as the robot moves, it is possible to guide the robot towards alternative routes and prevent it from getting stuck. In order to avoid ICS, the algorithm must guarantee that for every replanning cycle, the robot will have the option of a collision-free motion. As will be discussed later on, this is achieved by ensuring that a specific type of motion, called a *contingency plan*, will always be available.

Although it may be unintuitive at first, it turns out that this planning strategy can have multiple advantages over existing state-of-the art offline planners for problems with differential constraints. Despite the overhead of worrying about ICS, the proposed planning framework is shown experimentally to be more successful in harder problems and on average yields shorter solution paths in terms of execution time. In addition, since the planner is called periodically and for a fixed amount of time, this scheme only needs bounded memory. This contrasts offline planners where memory requirements can grow unbounded. It is also encouraging that the algorithm yields good results for three systems with very different characteristics and challenges. More specifically, experiments were ran on a segway which has the difficulty of being only dynamically stable and underactuated, a blimp which moves in a 3D space and heavily suffers from drift, and a 10 degree of freedom manipulator which has to move in

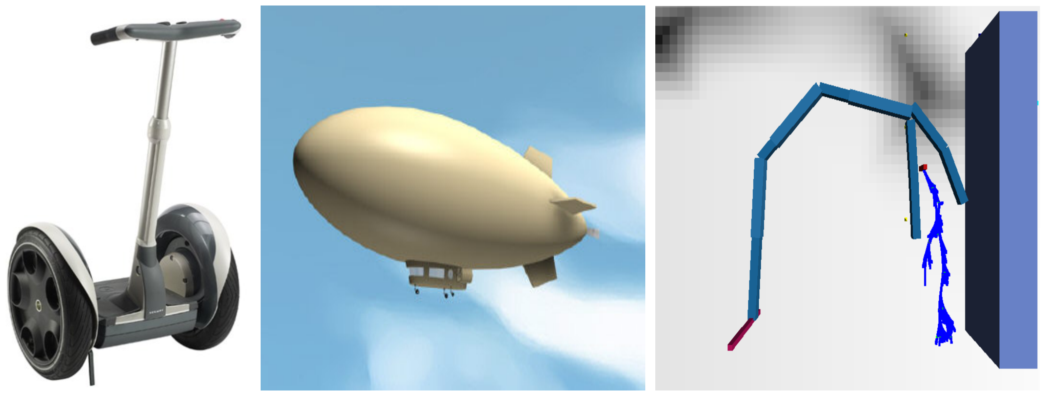


Figure 1.1: (a) A segway, (b) A blimp, (c) A 10 degree of freedom manipulator

a confined space. The robot models can be seen in Figure 1.1.

As a last comment, due to the incremental and adaptive nature of the replanning setup, this work has the potential to address a more general class of problems, involving only partially known and dynamic environments. Substantial work exists directly in areas where replanning is necessary to incorporate the latest sensor information in planning future motions. This thesis does not try to improve upon the existing work on replanning [32, 7]. Instead, it provides a study of the benefits of replanning even for problems where it is not technically necessary. At the same time, it attempts to establish a connection with problems invlolving unknown and dynamic environments which is an interesting direction for future work.

1.2 Problem Statement

Before proceeding, it is useful to introduce some basic notation and formalize the problems of interest in this thesis. Following the definition in [30] (Chapter 14), the motion planning problem for this thesis is defined as:

1. A world W populated by static obstacles O and a robot A.

2. An unbounded time interval $T = [0, \infty)$.

3. A smooth manifold X called the robot's state space.

4. The state space is described as $X = X_{free} \cup X_{invalid}$ where $X_{invalid}$ represents all robot states that are not allowed because the robot is in collision or violates some other global constraint.

5. A set of controls U that can be used to drive the robot. Given U, the robot motion is typically described by a set of motion equations $\dot{x} = f(x, u)$ defined for $x \in X$ and $u \in U$. Notice that for complicated systems, the equations of motion may not be known or may be hard to compute. In such cases, the motion of a robot under control u can still be obtained using a physics simulator(e.g, [40]). This can in fact allow more interesting motions such as slipping.

6. A state $x_{init} \in X_{free}$ is the given robot's initial state.

7. A set $X_G \subset X_{free}$ is the given goal region.

The solution to this problem is a *motion plan.* This is nothing else but a time sequence of controls $(u_1, t_1, u_2, t_2, u_3, t_3....)$. When the robot applies these controls from x_{init} it goes through states that lie only in X_{free} and ends up in a state $x_{final} \in X_G$. In addition, a slightly different version of the typical problem is of interest. On top of just finding a feasible trajectory to the goal region, it is important to measure the total time needed for the robot to actually reach the goal region. This aspect is usually ignored when formulating motion planning problems. It is however of practical importance, since a real robot will eventually need to execute the computed motion plan. In general, this notion of total time is the sum $computation + t_1 + \ldots + t_n$; the time it took to compute the plan plus the time it will take to execute it. Of course, given a specific problem, an interesting question is what a satisfactory total solution time would be. This work just compares the outcome of the proposed method against current offline algorithms.

1.3 Contributions

This thesis describes a planning strategy that uses replanning. The proposed algorithmic framework describes how to produce solutions for hard planning problems with differential constraints through the interplay of two basic modules: motion generation and motion selection. The approach is intended to be general enough and suitable for various robotic platforms. The algorithm attempts to utilize the power of modern sampling-based planners in searching high dimensional spaces, while dealing with the caveats of offline planning mentioned in Section 1.1. A full implementation is given using the Path Directed Subdivision Tree (PDST) planner [25]. However, different planners can also be plugged in. The results show that this scheme can achieve similar or better performance than state-of-the art sampling-based offline planners in a variety of different problems. The solution paths tend to be shorter than those produced by offline planners in terms of execution time. It is also important that these results are are obtained using only bounded amount of memory. Finally, the proposed algorithm is designed as an online algorithm and as such, it has the potential to deal with problems that involve dynamic and partially known environments.

In terms of implementation, this work tries to shed some light on the key components of PDST that can make this planner more effective. In particular, it is shown how to intervene in the scoring and penalization function to achieve goal biasing. Moreover, the modifications for using PDST in a loop are given. Finally, an efficient way for safety checking to avoid ICS is described, and it is shown how to incorporate a low level controller for avoiding ICS.

1.4 Summary of Remaining Chapters

The rest of the thesis is organized as follows. Chapter 2 contains a literature review on topics related to the targeted problem. Chapter 3 gives the full description of the proposed method for solving the above stated motion planning problem. Chapter 4 evaluates the performance of the method experimentally. The thesis concludes with a discussion in Chapter 5 about the limitations of the method, ideas for improvements and extensions.

Chapter 2

Previous work

This thesis describes an algorithm that addresses a number of difficulties arising in motion planning under differential constraints. A fundamental concept for planning algorithms is the abstraction of the configuration space or *C-Space*. For any robot, it is possible to define a vector q of parameters that are enough to describe the exact location of any point on the robot. The set of all possible configurations is the robot's configuration space. It is easy to see that any planning problem can be mapped into the problem of planning for a point robot in its *C-Space*. This idea can in fact be extended for robots with differential constraints. The difference is that now the robot is represented by a state, which is a point in the robot's state space X. The set of all possible robot states X, is usually a smooth manifold as was mentioned in Section 1.2. Even with this abstraction though the complexity of motion planning renders complete algorithms intractable. The reason is that it becomes very hard to explicitly construct the configuration or state space. There are two main directions that have been successful at solving planning problems: potential functions (Section 2.1) and sampling-based planners (Section 2.2). Both contain interesting ideas that are useful for this work. Potential functions can elegantly encode a general sense of direction for the robot. Simply said, if a specific type of potential function (called a navigation function) is available, it is easy to decide in what direction the robot should move from any state. Sampling-based planners on the other hand, are very efficient at searching for the solution. Especially the family of tree-based planners, can easily be used in problems with differential constrains [41]. More details and some background is given in subsequent sections.

In addition to planning methods, the approach proposed in this work uses online replanning. There is a lot of existing work on replanning algorithms. Replanning has been used so far in order to deal with problems where the environment is unknown or changing. This work on the other hand, employs replanning by choice for static and known environments even though it is not technically necessary. Finally, the combination of replanning and differential constraints gives rise to safety concerns. The robot may find itself in an inevitable collision state. Again, there exists literature on identifying such pathological states and methods for algorithmically avoiding them.

2.1 Potential Functions

Potential functions constitute an important and popular alternative for solving planning problems. They has been around since the 80's. A lot of useful references can be found in [16, 27] which cover potential functions extensively among many other motion planning topics. The interested reader can also look into [33, 42] as well as two recent textbooks [8, 30].

Although potential functions were initially proposed as a method for on-line obstacle avoidance [20], their underlying ideas can be used to yield elegant and intuitive planning algorithms. In the context of planning, the robot is guided to its goal under the influence of a forcefield that is induced by a potential function. The potential function is defined in the robot's *C-Space* and is typically composed of two terms: one attractive to the goal, and one combined repulsive term from all the obstacles. Once the potential function is defined, the goal should lie at the function's global minimum. The robot simply needs to move on the direction of the function's gradient until it converges to the global minimum at the goal. The main well-known problem of potentials is the existence of local minima. Even if the robot starts very close to the goal, it might converge to a local minimum and get trapped. One way to deal with the problem of local minima is to combine search algorithms with potential functions [2, 4]. Another idea is to define a potential function in the sense of Rimon and Koditschek [38]. Such a potential function has only one minimum at the goal by definition and is called a *navigation function*. The caveat is that in general, it is hard to construct a navigation function in a continuous space. The situation however, becomes much simpler if the space is discretized as a grid. In that case, a "wavefront propagation" algorithm [3] can be used to yield an approximate navigation function.

The important property that will be exploited in this thesis, is that fact that once a navigation function is defined, it can guide the robot to its goal from any starting configuration.

2.2 Sampling-based planning

Sampling-based algorithms by searching the robot's state space. This is a randomized search that aims at capturing the connectivity of the free part of the space by sampling. They depend on a collision checking function that can very quickly determine whether a specific point in the space , a sample, lies in free space of not. After dense enough sampling, those algorithms create a roadmap whose vertices are the samples, and edges exist when a collision-free path exists between two samples. The *Probabilistic Roadmap* (PRM) [19] is maybe the most famous and successful among the first sampling-based planners. These planners (especially the early versions), do not always consider the differential motion constraints that real robots have. However they were the first to solve many previously intractable problems [39].

The most successful and promising sampling-based planners lately are the tree-based planners [41]. They use sampling to build a tree data structure rooted at the robot's initial state and grow a tree until they hit the goal region. Among many, the two most popular examples are the Rapidly exploring Random Tree (RRT) [28, 31, 29, 21] with many variants (e.g., [18, 7]), and the Expansive State Tree (EST) [15, 14]. Tree-based planners were easily extended to problems with dynamic motion constraints mainly because a tree is a data structure that can naturally capture the notion of time; something critical when searching a state space where the ability of moving from state x_1 to x_2 does not imply that motion from x_2 to x_1 is also possible. The tree is expanded out of the robot's current state which is the root. The expansion is achieved by integrating the equations of motions forward for some amount of time using random controls to reach new parts of the state space via feasible trajectories. Taking this idea a step further, Path Directed Subdivision Tree (PDST) [25] first employed a physics-based simulator to include constraints such as friction and gravity that are hard to describe with analytic equations of motion. At the same time, a very recent paper that shares some ideas with this thesis is Discrete Search Leading Continuous eXploration (DSLX) [36]. DSLX uses a coarse grid discretization of the workspace. The grid cells are assigned weights that try to provide a global sense of direction leading the underlying sampling-based planner. This is an offline planner, but it adapts the weights of cells as the exploration progresses. Through this process, the underlying tree-based planner is guided towards the goal and significant speedups can be achieved.

Developing new and more efficient tree-based planners is still an active area of research. A starting point for the interested reader are [8, 30].

2.3 Replanning

As mentioned in Section 1.1, if a robot needs to move in an environment that is not fully known or changing, it is not possible to plan offline and execute the motion in open loop. Instead, the robot needs to compute a new plan frequently, to react to unexpected events in the environment. The idea of replanning is well established in the area of artificial intelligence [22, 32]. Algorithms like A^* and D^* Lite are able to quickly replan online and based on the latest available information, they find optimal paths for robots that navigate in unknown environments. Lately, there have been extensions for sampling-based planners. The continuous analogous to $A*$ is described in [10] (which provides a method for efficient replanning with RRT's). Their algorithm works for kinematic robots that move amidst moving obstacles. Everytime a moving obstacle invalidates part of the tree, the algorithm replans very fast to repair the solution path towards the goal. One of the first papers to talk about replanning under kinodynamic constraints is [12]. If the planner is fast enough, every time the robot encounters a moving obstacle, it can discard the current tree, and compute a new one from scratch starting at its current state. An efficient tree-based replanning algorithm is described in [7]. This planner is designed for robots in the small-sized league of Robocup: a highly dynamic adversarial environment where the positions of other robots can change very fast. Due to increased computational cost of planning in the state space, this paper describes an RRT-variant (called ERRT) which solves the path planning problem on a 2D workspace. Then, lower level control techniques are used to force the robot to follow the desired path. Another algorithm that has similarities to this thesis is GRIP [5]. It uses continuous replanning to plan for car-like vehicles that obey second order kinodynamic constraints. The paper describes a replanning algorithm for exploring unknown environments. Furthermore, it uses a two-dimensional discrete wavefront propagation to bias the underlying tree-based planner.

All these works justify the use of replanning by the fact that in unknown environments and in environments with moving obstacles, offline planning is not applicable. This is a strong motivating point, but not the goal of this thesis. Instead, this thesis contains a detailed study of the benefits that replanning can yield in hard motion planning problems, even in cases where offline planning is applicable.

2.4 Safety

When a robot has kinodynamic or differential constraints and moves using replanning, the issue of safety arises. More specifically, due to the motion constraints, the robot cannot change its direction and velocity instantaneously when in danger of an imminent collision. This means that the robot could select to move on collision-free trajectories, and still find itself in some state where no motion commands can deter the collision. For example, imagine a car that ends up in front of a wall at a very high speed and needs to replan its next motion. It is understood that for any robot to be useful in practice, it must be able to move safely avoiding these pathological situations.

All those states that are not in collision but if the robot reaches such a state it is impossible to avoid a collision in the future, are known in the literature as Inevitable Collision States (ICS) [11]. A description on how to take into account ICS states when planning in order to avoid collisions with the environment can be found in [35] and [5]. As will be explained later in the thesis, the idea is to define a set of contingency plans and then filter out as ICS all those states at which following the contingency plan leads to a collision. ICS situations can also appear between the members of a team of robots and [6] extends the notion of ICS in the case of multiple robots which utilize communication to avoid collisions. Closely related to the notion of ICS is the literature on *viability theory* [1]. A state is defined as viable if for any future time moment, a control exists that can drive the system away from an imminent collision. In this sense a non-viable state coincides with an ICS. Recent work for identifying and avoiding non-viable states when planning can be found in [17].

As explained in Chapter 3, this thesis describes a replanning framework under differential constraints and thus the robot's safety is a primary concern. The chosen approach uses the notion of ICS and Section 3.2.2 explains how to algorithmically avoid reaching such states.

Chapter 3

Algorithm

3.1 Abstract Algorithm Description

The overall algorithm is a combination of modules with distinct responsibilities. A simplified block diagram is shown in Figure 3.1. The planning process is broken down to replanning periods $P_0, P_1, P_2 \ldots$ of equal length. It can be assumed that the robot is resting in its initial state and can perform all necessary computations required for P_0. There are two major steps within each replanning period; *Motion Generation* and *Motion Selection*. Their purpose is to compute the robot's next motion. Computation takes place while the actual robot is in motion, executing the plan computed in the previous replanning period. During replanning period P_n the robot executes the motion plan that was computed in period P_{n-1} and computes the plan that will be executed in period P_{n+1}. This is reasonable under the assumption that there is no motion uncertainty and thus the state of the robot after executing a motion can be computed in advance.

The rest of this chapter is devoted to describing the modules within a replanning period in more detail. First, an abstract description is given, followed by a section that covers the implementation details.

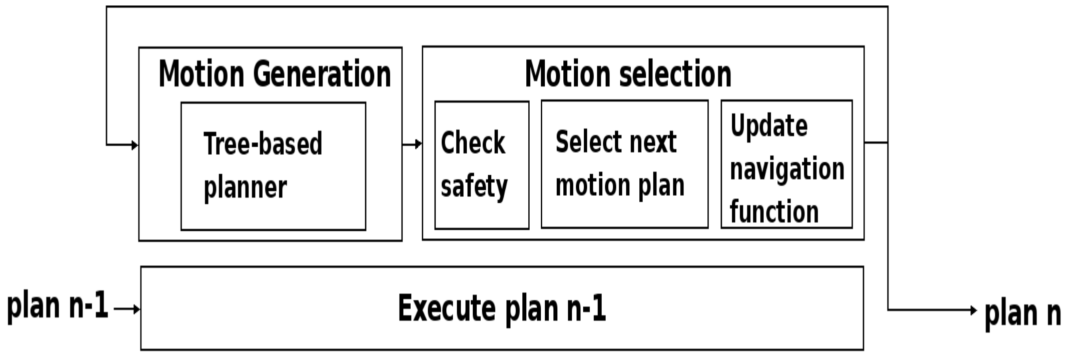


Figure 3.1: During replanning period $n-1$, the algorithm computes the next motion plan while the robot executes the plan computed in the previous replanning period.

3.1.1 Motion Generation

Given the state at which the robot will be after executing the current plan, a motion generation phase is responsible for computing a set of possible future motions. Those motions can be thought of as possible future actions for the robot. Although this thesis focuses on the use tree-based planners, note that the framework is general enough to allow other types of planners or even employing a prespecified set of maneuvers [13]. As long as a planner produces a discrete set of feasible motions, it can be plugged in here.

A typical tree-based planner searches the state space by expanding a tree data structure. This tree consists of samples which can either be states of X or whole trajectory segments in X. For the expansion step, first a sample on the tree is selected. Then, a random control vector is sampled. Finally, the control is applied on a simulated model of the robot; starting from the state indicated by the selected sample, a new sample is generated by integrating the robot model forward in time. By repeating these steps, the planner produces a tree of samples. With the information stored on the samples, each path down the tree represents a possible future motion. By extracting all the paths of the tree that describe motions of duration at least one replanning period, we obtain the desired discrete set of feasible motions that will be passed on to the Motion Selection module.

The planner is called once in every replanning period and has only limited time to run. For this reason, the planner is allowed to search only a small part of the space around the robot's state. Contrary to offline planning, here this is acceptable since each motion is not expected to have long term effects[1]. Within its time budget, it is important that the planner achieves a fairly good local coverage of the space around the robot to provide motion options in all directions. Experiments examining the effect of the varying the replanning period duration are given in Section 4.5.

3.1.2 Motion Selection

Given a set of feasible motions, this module is responsible for making the best motion selection that will take the robot closer to its goal. Before selecting the best motion, it is important to filter out all motions that are not safe. Recall from Section 2.4 that due to the motion constraints, a robot might end up in ICS states where collision is unavoidable. After safety checking, depending on the specific task

[1]One can also imagine scenarios of an unknown or unpredictable environment where it is not safe to search outside the sensing range of the robot.

at hand, any method that evaluates the motion options and selects the best one would be applicable. Here, a navigation function is used for evaluating motions. The third step of Motion Selection is to update the navigation function. This can include the discovery of new obstacles, but as explained below, it is mainly an attempt to remember the actions taken so far and monitor the progress towards the goal.

Guaranteeing Safe Motions

Safety is a critical issue that arises in systems whose motion is subject to acceleration and other dynamic constraints. In this work, safety is approached from the prism of ICS. In Section 1.2, the robot's state space was divided as $X = X_{free} \cup X_{invalid}$ into a free and an invalid set. To address safety, it is useful to refine this separation further. A system is considered to be in an *unsafe* state $x \in X_{free}$ if x is an ICS state. This means that although x itself is valid, it is impossible find a motion plan by which the system will avoid entering $X_{invalid}$ eventually. The state space is thus a disjoint union of three sets $X = X_{free} \cup X_{ICS} \cup X_{invalid}$ and the algorithm is required to guarantee the safety of the system, in the sense that the robot must move solely inside X_{free} as it was defined here.

The immediate problem that comes into to mind is how to identify X_{ICS}. In general, the question of whether a state is an ICS is undecidable in the continuous space. In the worst case, it would require the ability to check all possible time sequences of controls out of that state. Moreover, those sequences need to be checked for an infinite time horizon. The solution adopted here was based on [5, 6, 11].

Instead of asking whether there exist any plan that can prevent the collision at some state, the question becomes whether a specific control strategy can avoid the collision. Take for example a car. Let us define a specific control strategy for which the car fully deaccelerates until it comes to a complete stop while holding the steering wheel steady. This strategy can be applied at any car's state and with a simple forward integration, it is possible to see if the car will stop before hitting an obstacle or not. Of course, such a strategy can be considered to have infinite duration since the car can stay still after it has stopped. From now on, such predefined "escaping" control sequences will be referred to as *contingency plans*. Observe, that this approach is relatively conservative. States that are not true ICS will be treated as such if the specific contingency plan does not avoid invalid states.

The use of contingency plans offers an elegant way for safety checking. The next question is how to define contingency plans for an arbitrary robot. The example of a car suggests the use of stopping maneuvers as contingency plans. Unfortunately, this is not the answer for all systems. For example as explained in Section 4.1, a segway requires a sophisticated non-linear controller to compute a sequence of controls that can dynamically stabilize it. Moreover, there are systems such as Unmanned Aerial Vehicles (UAVs) for which coming to a stop is not even an option. An idea for such systems is to use repetitive cyclic motions that have infinite duration but can be checked for collisions in finite time. Given the specific type of contingency plan for the system at hand, the motion selection is responsible for choosing the best motion that is also safe in the sense that the contingency does not lead to collisions. The implementation section below provides details on how to do this algorithmically.

As a last remark at this point, it important to see that safety can be guaranteed in this framework only for static environments. The presence of moving obstacles generally breaks the guarantees. At best, if the motions of obstacles can be predicted for time τ with some accuracy, safety is replaced by $\tau - safety$ to represent the fact that the robot is only guaranteed to remain safe for this finite time τ. After that time period, the behavior of the system is unpredictable.

Navigation Function

The Motion Selection module needs to have a sense of direction and the ability to quickly answer what the best possible motion towards the goal from any state is. A navigation function with a single global minimum at the goal that captures the true distance from any state to the goal would be a perfect candidate [23, 38]. Yet, designing such navigation functions can be a hard and computationally expensive task for the problems of interest here (see Section 2.1). Instead, a navigation function on the workspace is used. First, the workspace is discretized into grid cells and then a 2D or 3D wavefront propagation algorithm is used. The discretization should be such that it does not slow down the computation. The cells that are covered by an obstacle take a value of infinity and do not participate in the computation. The rest of the cells get an integer value representing how many cells away from the goal the the current cell is. Due to the way it is computed, the navigation function encodes all the possible paths from any location to the goal. Fig. 3.2 shows an example for a robot on a plane that needs to go from the bottom left corner to the top left. In Fig. 3.2a, the darker the color, the farther that cell is from the goal. As a result, the preferred route is $R1$ through the narrow passage. There is also route $R2$ which initially appears longer. Notice that a straight line towards the goal is not even considered. These observations are more clear in Fig. 3.2b where the navigation function is drawn as a 3D surface with the height at each point being the value of the corresponding cell.

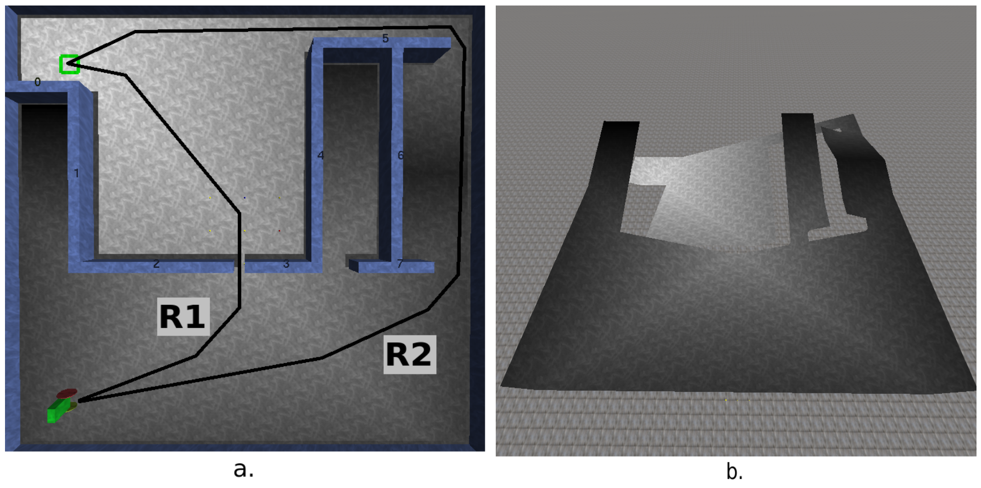


Figure 3.2: a. Navigation function on a 2D workspace. Darker means further away. b. A 3D illustration of the navigations surface.

Given the navigation function, all the candidate motion plans produced by Motion Generation are evaluated. For each motion plan, it is found inside which cell that motion ends. The motion plan gets the value that the navigation function returns for that cell. Motion Selection ends by selecting the motion with the minimum value since this will take the robot closer to its goal.

The caveat of computing the navigation function on the workspace is that it can be misleading. Route $R1$ in Fig. 3.2a leads to the goal through a narrow passage. A point robot can follow the series of cells with decreasing values indicated by $R1$ to reach the goal. A real robot can easily get stuck in front of a narrow passage through which it cannot fit, even if there is a solution via another route (route $R2$ in Fig. 3.2a). This is the price that has to be paid for computing a navigation function on the workspace instead of the state space. To resolve this issue, a learning technique is used for updating the navigation function.

Learning the Workspace

Intuitively, a robot that advances towards its goal should not go through the same area multiple times. If it happens, this is an indication that the robot is moving in circles and/or has difficulty moving in the direction suggested by the navigation function. It also means that it may be worth exploring other alternative routes to the goal before spending more time in this region - possibly a hard narrow passage. To make such a decision, one needs to detect the lack of progress. This is captured by keeping a *penalty* value for each cell, representing how much it has been visited. Initially, the penalties are zero for all cells. Every time the reference point (x, y) or (x, y, z) of the robot goes through a cell, that cell receives an additive penalty. To take into account the cell penalties, the values of the navigation function cells in the wavefront propagation are updated as : $N(c).value = c.value + N(c).penalty + 1$ where $N(c)$ is a cell that is neighboring with cell c (e.g., with 8 connectivity for 2D) and had not gotten a value so far. $c.value + 1$ is the value that $N(c)$ would

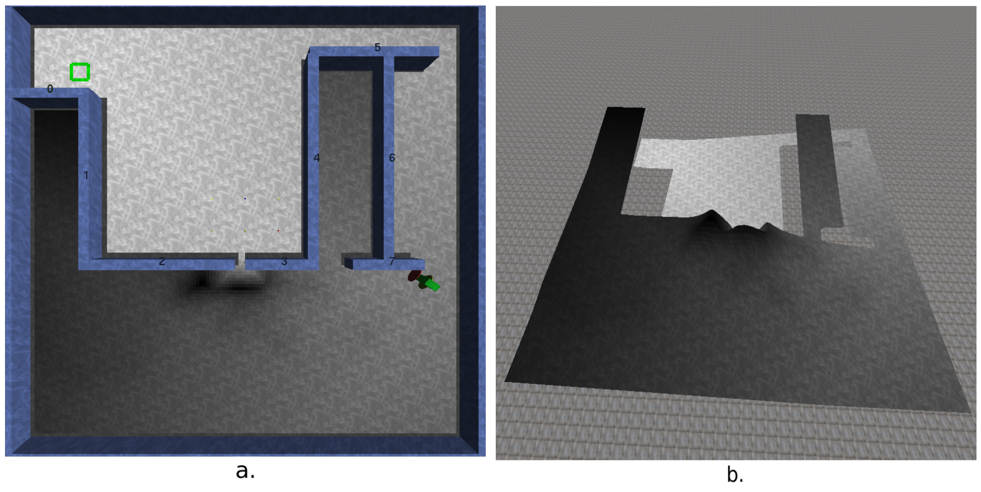


Figure 3.3: a. After some learning steps, the penalization around the narrow passage of route $R1$ makes this route less preferable b. A 3D illustration. Now route $R2$ is a better option.

get with standard wavefront propagation. $N(c)$'s own penalty is added to take into account how many times the robot has already gone through this cell. If a region of cells is penalized enough, the navigation function landscape will change and the robot will naturally try to go another way. Fig. 3.3a shows the effect of learning in the navigation function. The robot has spent some time trying to follow route $R1$ without success. After spending some time in front of the narrow passage, the navigation function changed due to penalization. This is reflected by the dark cells around the narrow passage of $R1$ (Fig. 3.3a). As a result, the navigation functions leads the robot through route $R2$. Observe in the 3D surface drawing (Fig. 3.3b), route $R2$ appears to be lower than $R1$.

3.2 Implementation Details

This section describes all the important algorithmic details and challenges that are related to the implementation of the algorithm described in the previous section. It follows the order in which the modules were given above. The tree-based planner that is used for motion generation is given first. Then details about checking for safety and the navigation function follow. Finally, the steps of the overall algorithm are presented.

3.2.1 Tree-based Motion Planner

Motion Generation uses a tree-based motion planner. The planner used here belongs to the family of Path Directed Planners [41] and is based on PDST developed by Andrew Ladd [25]. The implementation described below has certain modifications to make the algorithm usable for replanning. Moreover, the algorithm has been augmented with a biasing towards the goal which does not exist in the original algorithm. Before the algorithmic description, it is helpful to define some notation and data structure that the reader should expect to encounter. We assume that the robot is fully described by a state vector x. All states are elements of the robot's state space X. The state space contains all the configuration space components usually denoted as $q \in$ *C-Space* and in addition it can contain time derivatives $\dot{q}$ to describe the robot's velocities. All states at which the robot is in collision with an obstacle are invalid states of the robot. Moreover, a robot can have other constraints the render some states invalid. For example a segway is not allowed to tilt beyond a certain tilting angle, or a mobile robot may be required to keep all its wheels in contact with the ground at all times. Recall that $X = X_{free} \cup X_{ICS} \cup X_{invalid}$ where $X_{invalid}$ describes all the robot's forbidden states. The algorithm requires a generalized "collision checking" function that can answer whether some state is in X_{free} or not.

The robot's motion is controlled by some actuation variables u. Applying u at some state induces a motion. The algorithm requires a function that can take a state x_{init}, a control u and a time duration t and integrate forward in time to produce all the intermediate states (given some time resolution) on a feasible trajectory. From now on, a trajectory identified by (x_{init}, u, t) will be called a *motion sample* or just *sample*. Individual samples contain all the basic motion-related information, and a tree is encoded in the samples through pointers to the parent sample and the children samples. The planner stores the samples in a set called the *SampleSet*. In addition to the SampleSet, the tree-based planner maintains a subdivision of the state space into cells. Each cell has information about the cell borders and a set of the samples that are contained in the cell. The algorithm makes sure that each sample is contained in a single cell.

Algorithm 1 shows the pseudocode. It is supported by subsequent explanations of the important steps. Remember that the planner is responsible for producing valid motion options. A valid motion m is a time sequence of controls $m = \{(u_1, dt1), \ldots, (u_n, dt_n)\}$ that respects all the constraints but is not necessarily safe. The planner is given a limited amount of time to search the state space around the robot's state and build a tree of samples. The tree initially consists of one sample that starts at the robot's current state. At each iteration, a sample of the existing tree is selected and a new sample is propagated out of it.

Algorithm 1 KinodynamicPlanner(TimeLimit)

```
1: iteration = 1
2: while ($time < TimeLimit$) do
3:    $\gamma$ = SelectSample(SampleSet)
4:    Penalize($\gamma$)
5:    for ($i = 0; i < MaxPropagations; i++$) do
6:       $p$ = PropagateSample($\gamma$)
7:       if (HitGoal(p)) then
8:          return PlanToGoal
9:       end if
10:      if ($p\ != NULL$) then
11:         SetPriority(p, iteration)
12:         InsertToSampleSet(p)
13:      end if
14:   end for
15:   $c$ = GetCell($\gamma$)
16:   SubdivideCell(c)
17:   iteration++
18: end while
```

lines 3-4: The algorithm is trying to satisfy two competing requirements. On the one hand, it has to move into unexplored areas. On the other hand, for better coverage it needs the ability to backtrack. This behavior is controlled by the selection and penalization scheme. Each sample has two associated values, a **priority** and a **score**. At each iteration, the sample with the lowest score is selected out of the SampleSet. The score encodes the above two requirements. It is computed as : $score = N \cdot \frac{priority}{volume}$. The *priority* measures how many times a cell has been selected in order to allow for backtracking. The higher the priority value, the higher the score and thus the more difficult it is to select that sample again. The *volume* represents the relative size of the cell that contains the sample. As will become more clear in Section 3.2.1 below, cells with large volume correspond to areas of the space that are less explored. Larger volume results in lower score and so the sample has higher chances of getting selected. To obtain the score, in this work the priority over volume ratio is multiplied by the value of the navigation function (N) at the ending state of the sample. If that value is small, that means the sample brings the robot closer to the goal. Thus, multiplying by N yields lower scores and the samples closer to the goal have higher chances of getting selected. In *line 4*, the sample that has just been selected is penalized. In particular its priority must be increased in an exponential fashion. A standard suggestion is $priority = c * priority + 1$ where $c = 2$. Using higher values for c will result in more severe penalization, less reuse of the same samples and thus in trees that are more broad and less deep. To increase the success rate of the planner in producing feasible motions, it is possible to have an adaptive value for c. For example, c is larger if the newly propagated sample is very short. This will result in selecting the “bad” samples less frequently.

line 6: The actual search of the space is performed in this step. A new motion sample is created. It branches out of the sample that was selected in *line 3*. The details of this step are given in a separate Section 3.2.1 below.

lines 7-9 After the propagation step, the new sample may have entered the goal region X_G. If that is the case, then the algorithm follows the path on the tree of samples back to the root and keeps track of the controls used in each sample as well as the time for which those controls are executed before jumping to the next sample. The result is a time sequence of controls $\{(u_1, dt1), \ldots, (u_n, dt_n)\}$: a motion plan that drives the robot to its goal. If this is the case, the algorithm terminates successfully.

lines 11-12: If the attempt to propagate was successful, then those lines add the new sample to the SampleSet. Each new sample needs to take a priority that has to be greater than previously created samples, but still smaller than most samples that have already been used for propagations before. For this reason the current iteration number is assigned as a priority. Note that this function increases linearly while the penalization is exponential. The insertion function adds the new element to the *SampleSet*. This function is responsible for ensuring that no sample spans over multiple cells. For this reason, it is checked if the sample crosses the boundaries of the cell in which is was created (the cell in which γ lies). If this is the case, the sample has to be split into two new samples with same priorities (but potentially different scores if they lie in cells with different volumes).

lines 15-16: The final step of the algorithm is to reflect the fact that at least the area in the cell of the selected sample has been explored a bit more. This is achieved by subdividing the cell into two new cells. The process is explained in Section 3.2.1 below. When the cell is subdivided, all the samples in that cell have to be removed and reinserted to the SampleSet. Remember that the insertion process splits samples properly so that they only lie in a single cell.

Propagation

This is the procedure by which a new sample can be created to expand the tree. The new sample will have to branch out of the existing sample that was selected as described above. The important steps are given in algorithm 2.

Algorithm 2 PropagateSample(γ)

```
1: $state$ = ChoseRandomBranchState($\gamma$)
2: $control$ = SampleNewControl()
3: $newSample$ = Simulate(state, control)
4: if IsLongEnough(newSample) then
5:     ConnectToTree(newSample)
6:     Dissect(newSample)
7: else
8:     return NULL
9: end if
10: return newSample
```

line 1: This function selects a random state on the selected sample γ to be the starting state of the new sample. Since γ has a time duration t,S this step can be as simple as selecting a random number $t_{branch} \in [0, t]$. State $x(t_{branch})$ has to be computed and returned.

line 2: This is the sampling step present in some form in every sampling-based algorithm. Here, sampling consists of selecting a random control vector u that will be applied to the robot for some time. The easiest implementation that works well in practice is by selecting a uniform random number for each component of u within some specified bounds.

line 3: After the starting state and the controls have been selected, the integration function is used to simulate the system forward in time. At small time intervals the collision checking function is invoked to check if the robot has entered $X_{Invalid}$. As was mentioned earlier, a generalized collision checking function is needed here. In its simplest form, it should be able to determine whether the robot collides with an obstacle or not. This typically implies that the robot has a model of the workspace and its own geometry [2]. Over the last decade, there has been a lot of research on performing collision checking computations between polyhedra efficiently (e.g., [26, 9] and many others). Besides collisions with obstacles though, a state could be violating some other constraint. If for example the linear velocity exceeds some maximum bound, or if the wheel of a mobile robot loses contact with the ground, the collision checking function can decide that a state is invalid. If an invalid state is reached, the propagation terminates. Moreover, although it is generally a good idea to have long samples for more efficient exploration of the state space, it is good practice to impose a

[2] A common approximate representation keeps a triangulation of all the involved polygonal faces.

maximum duration to samples. Even if no invalid state was reached, at some point the propagation is terminated. The reason is that in any search process, no single direction should be trusted as being the correct way to the goal.

lines 4-7: After propagation, the sample is post-processed. If it is too short, in practice it can be rejected. The reason is that, if the SampleSet is filled with too many very short samples, performance is degraded. Line 5 connects the new sample to the tree by updating necessary bookkeeping information. Finally, in line 6 the sample is cut down into smaller samples. This is a step that is useful in the context of replanning: if the sample corresponds to a time interval that spans over several replanning periods, the sample is split into smaller samples according to the timestamps that lie exactly on an integral number of periods. Specifically, all samples that start on replanning period m are inserted in a list L_m.

Subdivision

From the above description, it is apparent that the planner does not use a primitive for computing the distance between states in the state space. Instead, it uses a subdivision scheme and the selection process described above to balance the search towards unexplored areas with backtracking.

An n-dimensional state space X with elements $x = (x_0, x_2, \ldots, x_{n-1})$, can be viewed as an n-dimensional hypercube. This means that components representing rotations are treated as Eucledian coordinates. In this work, all angles are always renormalized to be in the interval $(-\pi, \pi]$. Initially, the whole space is considered to be a hypercube of just one cell with normalized volume 1. As the algorithm proceeds, a sample γ is selected and a new sample is propagated. Then, the cell c that contains γ needs to be subdivided. In a subdivision step, a cell is split in half with respect to a certain dimension, and two new cells are created each with half the volume of the parent cell. At any given moment, the total space is represented by the cells at the leaves of a binary tree. The dimension at which a cell is subdivided depends on the depth of that cell in the subdivision binary tree. The convention is that a cell at depth k will be split on the dimension $d = k \bmod n$.

If some dimension x_i is being subdivided, this means that the algorithm keeps track of how well this dimension has been explored. So far it was assumed that the space being subdivided is the robot's state space X.This does not have to be the case though. In many cases X is high dimensional and it is not important to explore all of its dimensions. For mobile robots for example, experience shows that the Cartesian coordinates need to be covered more thoroughly than the velocity components. In general, it may prove beneficial to map the space X into a new space Y, which is projection of X. This can be as simple as an orthogonal projection on the important dimensions of X but it is possible to design even more sophisticated mapping functions to try and focus the search on certain parts of X. Notice that the mapping $Y = map(X)$ does not need to be invertible. In the case $Y \neq X$, a sample is crossing its cell's boundary if it contains a state x such that $y = map(x)$ lies outside the cell.

3.2.2 Checking for Safety

The *Motion Selection* has to select the best possible plan that is safe. For this reason, the possible options offered by *Motion Generation* have to be checked for safety. Safety checking is done for a given state by simulating a contingency plan and asking whether it results in a collision or not. The following description will also use the term safety checking for a sample. In that case, it is implied that the check is performed for the first state x_{init} of that sample.

There are different strategies for doing safety checking on the tree data structure produced by *Motion Generation*. The main limitation is that checking the contingency plan for one state is computationally expensive. Simulating one contingency plan can cost up to the order of milliseconds. This is an expensive operation in a real-time planning framework where the whole replanning cycle typically lasts fractions of a second. It is thus intractable to do safety checking out of all states on the tree. In fact, it should not be necessary; since the planner has a fixed replanning period DT, it makes sense to only check states that have a timestamp $t = n \cdot DT, n = 0, 1, 2, \ldots$. Still, it is not uncommon to have hundreds of such states on a tree. A further refinement is used in [5]. Only the states that appear on the first replanning period ($t = DT$) are checked. This is reasonable since the robot will only move for one period and then use its contingency plan if the planner fails to provide any other option. In addition, safety checking is done while the tree is being built and the

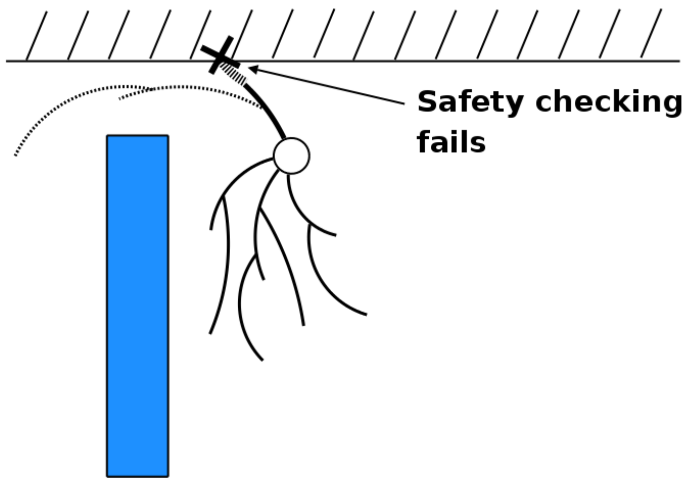


Figure 3.4: A sample leads to an unsafe state, but if more propagations are attempted out of it, it is eventually safe and also solves the narrow passage.

tree is grown only out of the safe states on the first period. Figure 3.4 shows an undesirable situation that appeared frequently in the workspaces that this thesis focuses on: many samples do not pass the safety check because they lie too close to an obstacle. Frequently though, this happens just because those samples are leading through a narrow passage. If more propagations are attempted out of those samples, it is possible that they lead to a safe sample eventually. In other words, by pruning the tree in this way, a lot of samples with good exploration potential are discarded. Such scenarios are common in systems with complicated dynamics where the valid motions are limited and the planner does not have the luxury of abandoning certain motions so quickly.

It might be beneficial to postpone safety checking until the tree is completely built. In addition, if some sample is proven safe, then all the states prior to that sample on the path back to the root of the tree are safe. Based on the previous discussion, in this thesis, safety checking is a separate step which takes place after the tree-based planner generates a tree of motion. The planner, stores samples with timestamps on the beginning of replanning period m ($t = m \cdot T, m = 0, 1, \ldots$) in a list L_m. Safety checking can be done systematically starting from the samples that are in lists far in the future. For every sample that was proven safe, all the samples prior to it in time do not need to be checked. Due to strict time limitations, even with this strategy it is not always possible to exhaustively check all samples on all lists. The algorithm ranks the samples based on how close to the goal do they bring the robot. For the ranking the navigation function is used to estimate how far each sample is from the goal as described in Section 3.1.2. Using this ranking, the few best samples in each list are checked first. Safety checking can continue as long as there is time. Notice that the effectiveness of this approach is related to tree retainment. If a significant part of the tree is retained, then a lot of the safety checking on that part has already been done in previous replanning periods.

3.2.3 Computing the Navigation Function

As was mentioned in Section 3.1.2, the navigation function is computed using a wavefront propagation algorithm on a workspace grid. Generally, if the grid is fine, the distance approximation that is represented by the navigation function will be more accurate and even motion samples that are close to each other can be distinguished. Yet, it is important that the computation of the navigation function does not take significant amounts of time. Initially none of the cells corresponding to free space has a value. A function is needed to identify the cells that are partially covered by a workspace obstacle. Given the goal cell, the navigation function is computed as shown in algorithm 3. The computation uses a queue data structure Q. Remember that each cell has associated with it a penalty value representing how many times the robot has gone through that cell.

Algorithm 3 NavigationFunction(c_{goal})

```
1: $c_{goal} = 0$
2: Q.push($c_{goal}$)
3: while ($Q \neq \emptyset$) do
4:     c = Q.pop()
5:     for all $c' \in Neighborhood(c)$ do
6:         if $c' \neq Obstacle$ and $c'$ has no value then
7:             $c'.value = c.value + c'.penalty + 1$
8:         end if
9:     end for
10: end while
```

3.2.4 Updating the Navigation Function

As the robot moves, the navigation function cells receive penalties, to indicate the fact that the robot has gone through them. The value of the penalty encodes the robot's "impatience". If is too high, then the robot will not stick to a certain path suggested by the navigation function for too long, and will quickly rush into alternatives. In the other extreme, if the penalty values are too low, the the robot may get stuck in front of an unsolvable narrow passage for a long time before the penalties accumulate and alternative routes become more preferable. One other implementation detail come from the fact that in all examples investigated in this work, the robot is large enough to take up more space that the size of the navigation function's grid cells. For such cases, it was found useful in practice to penalize a region of cells around the single cell where the projection of the robot's reference point is located in. The navigation function is defined in the workspace, where the robot is not a point. Penalizing a region of cells is in agreement with the fact that the robot has volume and takes up space. An easy way to do this is by penalizing all the cells that lie in a Gaussian distribution centered at the cell containing the robot's reference point. The cells that are farther will receive less penalty than the center, according to the distribution used.

3.2.5 Overall Replanning Algorithm

This section provides a description of the overall algorithm combining all the modules described above. Again, the pseudocode is supported by explanations below.

line 1: The first step is not necessary for the static and known environments that are mostly studied in this thesis. It is included for the few examples where the algorithm was tested in a partially known environment. At the beginning of a replanning cycle, the robot collects the latest sensor information. This step abstracts the whole mechanism for processing that information. This could include the detection on unexpected static obstacles that appeared on the map, or the prediction of the trajectories of moving obstacles that entered the robot's

Algorithm 4 Replanning Loop(Goal, CurrentPlan, NextStartSample)

```
1: ReceiveNewSensingData()
2: while (Robot Executes CurrentPlan) do
3:     PenalizeCurrentTrajectory()
4:     RecomputeNavigationFunction()
5:     RetainTree(NextStartSample)
6:     ResetSubdivisionSpace(NextStartState)
7:     RunKinodynamicPlanner(Goal, DurationOf(CurrentPlan))
8:     CheckPlanSafety()
9:     NextPlan = ChooseBestPlan(Goal)
10:    if ($NextPlan == NULL$) then
11:        NextPlan = ContingencyPlan
12:    end if
13: end while
14: return NextPlan
```

sensing region in a more general scenario.

line 2: In this replanning framework, the robot is planning its next motion while moving on a previously computed trajectory. In this way, planning and execution happen in parallel. The advantage is the real-time online planning behavior, while the disadvantage is that planning is done based only on the information obtained in the previous replanning step. Moreover, this approach will be problematic if the final state after executing a motion can not be predicted accurately in advance.

line 3: This step was discussed in Section 3.1.2. The robot keeps track of the regions it has previously visited by penalizing the corresponding cells of the navigation function.

line 4: Based on the latest available sensor information, the robot may need to revise the navigation function to include for example, some newly discovered obstacle.

line 5: This is one of the steps that connect consecutive replanning steps. The robot moves on a trajectory that corresponds to a specific path along the tree built by the tree-based planner. The subtree down that path will be valid in the future. It can be retained and used as a base for the planner in the next planning cycle.

line 6: The purpose of the tree-based planner is to search the state space around the robot. As described in Section 3.2.1, the planner uses a subdivision of a state space hypercube to do this search systematically. Since the tree-based planner has a limited amount of time within each replanning cycle, only the part of the space around the robot's current state is searched. More specifically, the algorithm generates motion options within a hypercube. The hypercube is initially assumed to have a normalized volume of 1 and the robot lies at its center. For the replanning scheme, in every cycle, the center of this hypercube is reset to the state at which the robot will be after executing its plan. All the cells created in the previous replanning cycle are erased, and the space is again represented by one cell with volume equal to 1.

line 7: After all the information has been updated, the planner can run to compute the set of possible plans for the next planning cycle. The samples that start at a timestamp in the beginning of replanning period m ($t = m \cdot T$) are inserted in a list L_m. Samples within a list are ordered according the their distance to goal. Recall that this is given from the navigation function.

line 8: Before selecting any plan for the next planning cycle, the tree produced by the planner has to be checked for safety. This step is crucial to guarantee that the robot will always have a safe plan to execute. The details were given in Section 3.2.2 above.

lines 9-12: Among the set of available safe plans, the algorithm needs to choose which one will be executed during the next planning cycle. The obvious choice is the plan that will bring the robot closer to its goal. Yet, in some cases it might be beneficial not to be entirely greedy. In particular, for complex systems (for example when stability constraints are present) it might be hard for the planner to produce a rich set of plans. This is more pronounced if the time allocated to the planner is small. To alleviate this problem, it is a good idea to choose a plan that also leaves a large enough subtree to be retained. Nevertheless, if no plan was found from the planner, the contingency plan is selected on *line 11*. The contingency plan is guaranteed to be available due to *line 8*.

3.3 Summary

This Chapter presented all the details of the proposed method for solving the class of problems described in Section 1.2. Motion plans are found via an online replanning scheme. The process is broken into replanning cycles. Within each cycle, the robot is executing a previously selected motion and at the same time it has to compute the motion to be executed next. This computation is split into a *Motion Generation* and a *Motion Selection* phase. *Motion Generation* uses a tree-based planner to search the part of the state space around the robot's state to produce as many motion options as possible. *Motion Selection* filters this set of motions and selects that best one. First, the motions have to be checked for safety to ensure the robot will not end up in an ICS. Second, a navigation function is used to evaluate how close to the goal each of the motions brings the robot, and the best motion is selected. Finally, the robot marks the parts of the workspace it has previously visited so as not to get trapped. This is achieved by penalizing the navigation function cells whenever the robots goes through them. The next chapter contains a series of experiments that try to evaluate the different aspects of the method. Notice that the overall replanning scheme is general and does not depend for example on the specific tree-based planner. The experiments actually confirm this by exhibiting similar performance when an alternative tree-based planner is plugged in. Moreover, as long as a memory mechanism is used to allow the robot to backtrack and explore all of its workspace, the use of the navigation function could be substituted with some other way for evaluating and selecting motions. To conclude, there are two aspects whose influence on the algorithm is worth investigating: the effectiveness of the penalization and the duration of the replanning period. Both topics are discussed in the following chapter.

Chapter 4

Experimental Validation

4.1 Experimental setup

Robots: This section presents a series of experiments on three robots. The bulk of the experiments were ran on a segway and a blimp, but the algorithm has also been tried on a 10 degree-of-freedom (dof) manipulator as a proof of concept. All robots present significant difficulties to planners because their state spaces are high dimensional, they have differential constraints and, except for the manipulator, are underactuated. The segway moves on a plane and its state can be fully described by seven parameters $(x, y, \theta, \alpha, \dot{\alpha}, v, \dot{\theta})$; x and y provide the location and θ the orientation, α provides the segway's tilting angle and the last three parameters are velocities. The segway is controlled only by two torques applied on its two wheels. The blimp is a model for a balloon and moves in a 3D workspace. Its states are described by parameters $(x, y, z, \theta, V_{fwdx}, V_{fwdy}, V_z, \omega)$. Its location is given by (x, y, z) and its orientation is θ. Its linear velocity has two components V_{fwdx} and V_{fwdy} that are parallel to the ground and one vertical V_z. There is also a rotational velocity ω describing the rate of change of orientation. A blimp has three controls. One force along the axis $(cos(\theta), sin(\theta), 0)$, a second vertical force along $(0, 0, 1)$ and a torque around $(0, 0, 1)$. All forces and torques for both models are bounded and the robots are affected by gravity and ground friction (for the segway). The manipulator has 10 dof due to 3 prismatic and 7 rotational joints. It is assumed to move on the plane and there is an actuator for every joint. One extra complication that further limits the set of allowable motions, comes from the fact that no self-collisions are allowed between the robot's links. To simulate the behavior of the robots, the ODE open source physics simulator was used [40]. The contingency plan used for the segway is a stopping maneuver. Unfortunately, bringing a segway to a stop is not trivial and requires the use of a controller. For the blimp, the chosen contingency is a repetitive back and forth motion at constant height. For simplicity, the manipulator was assumed to be able to stop instantaneously (large enough forces are available). Details for the segway's contingency are given in the next paragraph.

Contingency for a segway

The segway is an unstable system. If not actively controlled it cannot stay upright and falls. It moves forward by constantly compensating for a falling motion. It is a dynamically stable system. Among other complications, this property implies that a segway cannot be stopped in the traditional intuitive sense that was used in the car example above. A stopping motion can nevertheless be achieved with the use of feedback control loop. The controller implemented in this work is fully described in [34]. Here only a sketch of the important components is given.

The aforementioned paper describes a velocity controller that is able to achieve a target linear velocity v_d at some target orientation θ_d. For arbitrary v_d and θ_d, the controller provably converges but it can take several seconds depending on how the control gains are tuned. Since time is of the essence, in this work the controller is asked to achieve a (close to) zero linear velocity. To simplify the controller's task even further, the desired final orientation is the same as the orientation at the state where the controller is invoked.

The controller is derived using a partial feedback linearization technique to break-down the seven equations of motions into four linear and three non-linear equations. Then, two controllers are designed. One low level controller for the linear part that can be very fast and one high level controller for the non-linear part. The two controllers are connected through the tilting angle variable a. The idea is that the tilting angle is directly correlated to the segway's linear velocity v since the latter always has the tendency to go in the direction indicated by a. For this reason, the high level controller sets a target tilting angle a_r based on the difference between the current velocity and the desired velocity. Then, the low level controller is responsible for achieving that tilting angle a_r as fast as possible while fixing the orientation. The equations for the two controllers are given below:

Low level controller:

$$w_1 = -k_{qv}\dot{\theta} - k_q(\theta - \theta_d), \quad w_2 = -k_{av}\dot{a} - k_a(a - a_r)$$

High level controller:

$$\dot{a}_r = -k_r f_{ss} - k_v (a_m^2 - a_r^2)^2 (v - v_d) \frac{f_{ss}(a_r)}{a_r},$$

where $k_{qv}, k_q, k_{av}, k_a, k_r, k_v$ are controller gains, a_m is the maximum allowed tilting angle and $f_{ss}(a)$ is a function computed at the steady state value a_r when the low level controller has converged; w_1 and w_2 are the result output controls that can be transformed into motor torques that need to be applied in the two segway motors.

Typically, the high level controller needs to run at a low frequency (i.e., every 0.5 seconds) while the low level controller needs to run about 50 to 100 times faster. The gains need to be adjusted so that the low level controller converges to the selected a_r before the high level controller selects a new a_r.

This algorithm is able to bring the segway to an almost stopped state at which the linear, angular and tilting velocities are approximately zero (e.g., $|v| < 10^{-3}$). This usually takes up to 4 seconds of real-time segway motion and the forward simulation required for safety checking is in the order of milliseconds. It can be seen that running the controller is computationally expensive and should be used as little as possible.

Algorithms: For the experiments, two tree-based planners were used. The main tree-based planner is described in Section 3.2.1. It will be referred to as PDST in this section. For generality and comparison the RRT[31] tree-based planner was used. In the rest of this section, *PDST* and *RRT* will stand for the offline planners, while *RPDST* and *RRRT* are the replanning versions according to the method described in Chapter 3. For *RRT* , the bias used, was 7% for the offline version and 3% for the replanning version. The distance metrics were hand-tuned weighted Euclidean distances for both the segway and the blimp. For the segway the weights were $w_x = w_y = 0.35, w_\theta = 0.1, w_\alpha = 0.1, w_{\dot{\alpha}} = w_v = w_{\dot{\theta}} = 0.033$. For the blimp the weights were $w_x = w_y = w_z = 0.32, w_\theta = 0.0009, w_{velocities} = 0.009$.

The *PDST* implementation was given in Section 3.2.1. To plan for the robots described above, subdivision was done on the x, y, θ, α dimensions for the segway and on x, y, z, θ for the blimp. For the manipulator things are slightly more complicated. In particular, finding a good reference point on the robot is not as intuitive as with mobile robots. Since the robot moves on a plane, a choice was made to keep track of the (x, y)-location of a point P that is the average of the (x, y)-locations of the four links that constitute the two fingers of the robot. Using P the subdivision is done only on the x, y coordinates, obtained by mapping the 10-dimensional configuration vectors to the (x, y)-locations of P. Also, the values of the navigation function for each sample are taken at point P.

Finally, for the replanning versions the duration of the replanning period was set to 0.5 seconds in all experiments unless stated otherwise.

Environments: The environments for the experiments were designed to test the above algorithms in a variety of narrow passages and cases of misleading navigation functions. The *block* environment (Fig. 4.1a) is a structured and symmetric environment with multiple ways to reach the goal. The *door* environment (Fig. 4.1b)

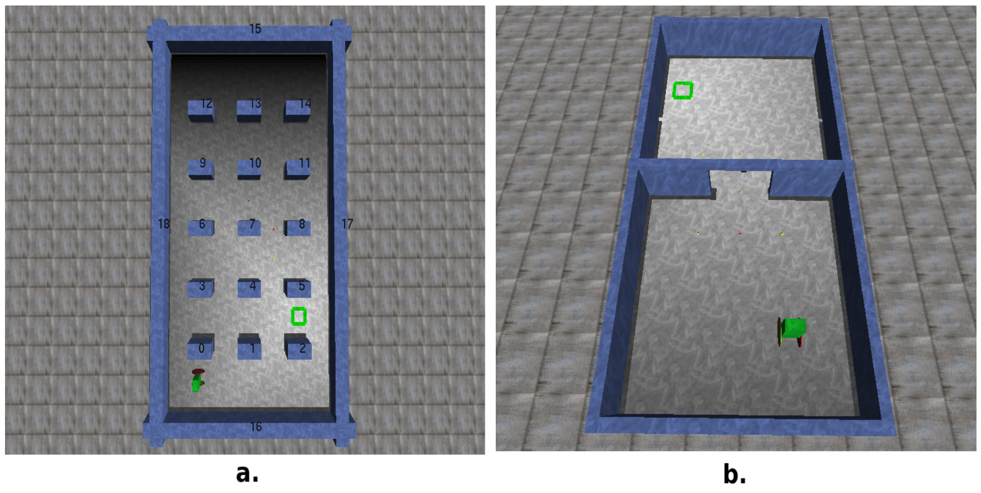


Figure 4.1: a. The *block* environment, b. The *door* environment.

contains an interesting and slightly unusual narrow passage. The segway needs to go under a door. Given the door's height, this is only possible if the segway is tilted enough, which in turn requires the segway to be moving fast enough. The third segway environment is the *building* (Fig. 4.2a). The shortest route to the goal leads through a passage that is too narrow for the segway. The goal is reachable via two longer routes. The blimp was tested on one environment (Fig. 4.2b). The blimp has to fly through a window and then hover between two parallel plates to reach its goal. For the manipulator, a classical example is used where the robot needs to squeeze itself through a narrow passage and grab a cylinder with its two fingers (Fig. 4.3).

Setup: All experiments were performed on a 2.66 Ghz processor and 4 GB of RAM. All experiments have a time limit of 10 minutes, and are averages over 50 runs unless stated otherwise.

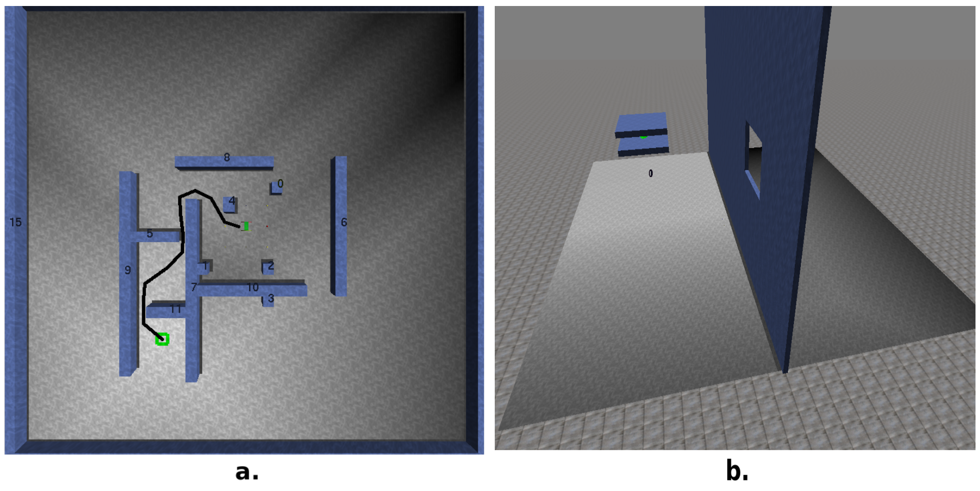


Figure 4.2: a. The indoor *building* environment, b. The environment for the blimp

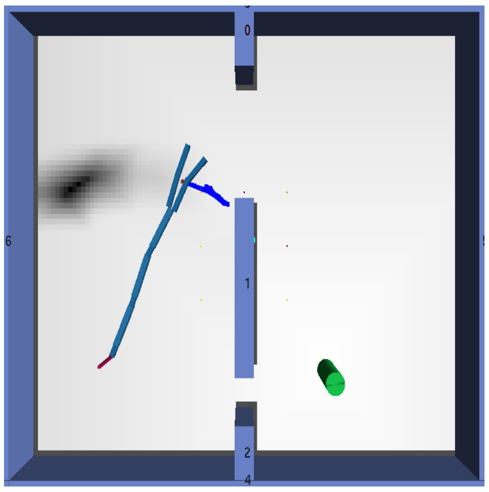


Figure 4.3: A 10 dof manipulator trying to grab a cylinder

4.2 PDST vs RRT

First, it is of interest to compare performance of the offline planners. Since RRT is a well-known standardized algorithm, this will help confirm that *PDST* is also powerful state-of-the-art planner. The comparison was performed on the *building* and *door* environments. In *building*, *PDST* solved the problem 95% of the time and took on average 40.2 seconds. *RRT* succeeded only 12% of the time and runtimes averaged to 433.1 seconds. In *door*, experiments of incremental difficulty were ran by lowering the height of the door. Table 4.1 shows the results for four height values. The left column states how tilted the segway must be to be able to go under the door. The segway's maximum tilting angle is 25 degrees. In both environments, *RRT* requires more time and fails more often, so *PDST* is indeed a powerful tree-based planner. [1]

Difficulty (deg)	PDST		RRT	
Very Easy : (8.8 – 25)	1.5	100%	68.9	99%
Easy : (12.13 – 25)	2.1	100%	87.9	98%
Medium : (16.26 – 25)	16.5	100%	171.53	19%
Hard : (21 – 25)	N/A	0%	N/A	0%
Very Hard : (23 – 25)	N/A	0%	N/A	0%

Table 4.1:

4.3 Replanning vs Offline Planning

This is an important set of experiments, that exhibit the benefits of replanning. Due to the decreased performance of *RRT*, the experiments are limited to *PDST*, *RPDST* and *RRRT*.

Segway: Of interest, is the behavior of the algorithms as the difficulty of a problem increases. In *block*, five goals that are increasingly far from the starting location are set. In *door*, five goals are defined by decreasing the height of the door as described in the previous subsection.

[1]Of course, this may not be the true difference between the algorithms, since RRT can be hand-tuned to have better performance. Note though, that most powerful versions use a bidirectional tree. Since a physics simulator is used and the segway is allowed to slip, it is not possible to use those versions since they require the ability to integrate backwards in time.

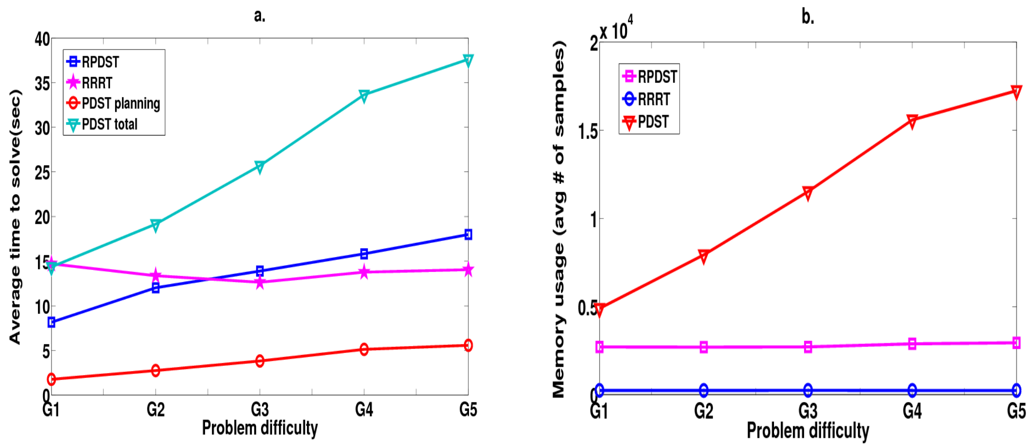


Figure 4.4: a. Time to plan and solve increasingly hard problems in *block* using *PDST*, *RPDST* and *RRRT*, b. Average memory requirements.

Figures 4.5a and 4.4a plot the time it takes to solve each problem. *PDST* uses its global view of the space to find the solution faster that the replanning algorithms. Faster planning though comes at price in memory as shown in figures 4.5b and 4.4b. As the problems get harder, especially in *door*, the memory requirements increase dramatically. Replanning finds the solution as concatenation of small partial paths. As expected, the memory usage is practically constant, with *RPDST* being more memory intensive over *RRRT* (by a constant factor). The increasing demand in memory has severe negative effects. In *door*, there is no data for *PDST* beyond G3 since the planner never solved the problem within the 10 minutes time limit. This is probably because the data structures become slow when lookups and updates are needed on large numbers of samples. Allowing for more time would eventually lead to memory exhaustion forcing the algorithm to terminate. On the other hand, the replanning algorithms only use bounded memory and can thus run indefinitely.

Looking back at the time plots in Figures 4.5a and 4.4a, an interesting observation

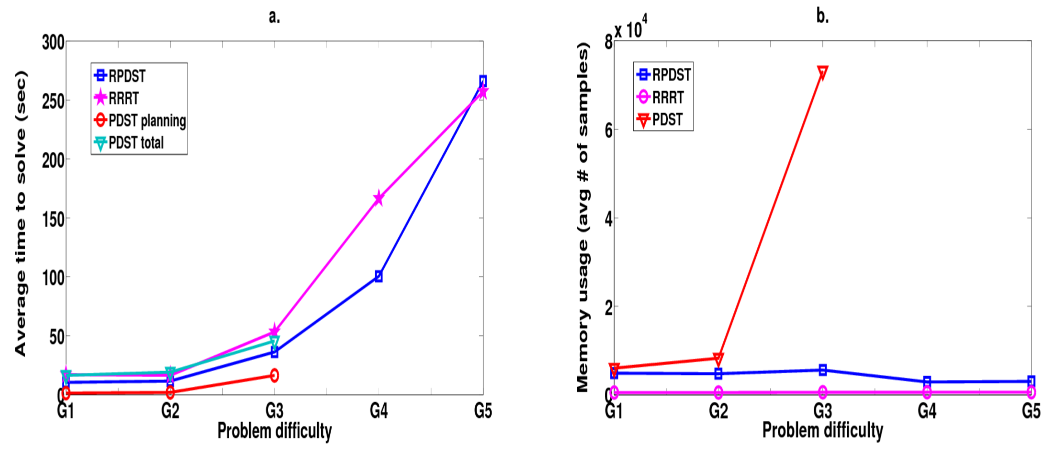


Figure 4.5: a. Time to plan and solve increasingly hard problems in *door* using PDST, R-PDST and R-RRT, b. Average memory requirements.

can be made. When solving a planning problem, the time needed for the real robot to execute the plan and move to its final location is a factor to consider. For the replanning algorithms, this time is the same as the planning time. *PDST* runs offline so the plot also contains the *total time* which is the planning time plus the time it takes to execute the plan. Notice that the replanning algorithms are doing much better in *block* and equally good in *door* until G3. This means that the paths produced by replanning tend to be shorter. This is an encouraging finding. Replanning is in a sense, a divide and conquer greedy strategy. The total problem is broken down into smaller ones, and in each step, the robot executes the best possible motion. Guided by the navigation function, this strategy can yield much shorter paths.

Notice also, that although *RRT* was shown to have significant difficulties in the problems at hand, its replanning version is very competitive, and in some cases even slightly better than *PDST*. This is explained by the fact that *RRT* and *PDST* were compared as offline planners. when *RRT* and *PDST* run just for the duration of a replanning period, the produced trees are small and the differences at exploring a small area around the robot's state become insignificant.

Blimp: The results so far, were based on experiments with the segway. The experiments for the blimp confirm the described findings. All planners solved the problem on all runs. *PDST* solved the problem at an average of 37.65 seconds just for planning and 190.26 seconds total, when the execution time is added. On the other hand, *RPDST* 's average runtime was 153.02 seconds and *RRRT* 's 125.06 seconds. Again, the offline planner find the solution much faster. Yet, *PDST* required on average 30 times more memory than *RPDST* and about 300 more than *RRRT* . Furthermore, the paths computed by *PDST* are much longer than those of *RPDST* and *RRRT* , as indicated by the total times above.

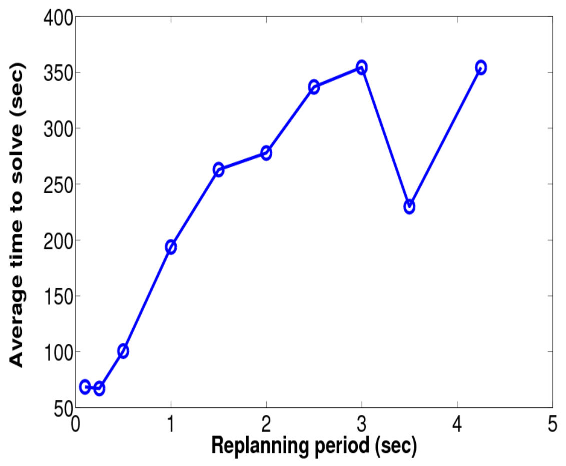
Manipulator: The manipulator is the latest piece that was added to this work as a proof of concept. The robot's structure and motion limitations are very different than those of mobile robots. Due to its high dimensional state space, it was impossible to solve the problem with the offline planners within the allowed time limit because they simply exhaust the available memory. Surprisingly it was possible to solve the problem in under 10 minutes 20% of the time with *RPDST* . This is an interesting result since a 2-dimensional navigation function seems able to guide a 10 dof robot using the reference point P described above. In the cases where *RPDST* failed, it appeared that the robot got stuck in some degenerate configuration with one or more of its joints stretched to its limits. It could be the case that the duration of the replanning period was too small for the planner to be able to find a way out, although varying the durations from 0.5 to 3 seconds did not appear to help. The other explanations could be numerical inaccuracies in the physics engine that bring the robot to a collision-free state that is too close to the obstacles and any attempt to move results in a collision.

4.4 Effect of Learning

Up to now, *RRRT* and *RPDST* were tested on problems where the navigation function was not misleading. Recall that in *building*, the navigation function leads through a passage that is too narrow. As mentioned in Section 3.1.2, replanning needs to be augmented with a learning functionality, to avoid getting stuck when the navigation function is misleading. For learning, each cell got a penalty of 0.05 every time it was used. The results are encouraging as both replanning algorithms found the solution in 100% of the attempts. *RPDST* took 47.7 seconds on average while *RRRT* took 69.95 seconds. To compare, *PDST* 's planning time was reported in 4.2 to be 40.2 seconds and the average *total time* was 98.51 seconds.

4.5 Duration of Replanning Period

The final set of experiments investigates the effect of the duration of a replanning period (RP). Notice that if the replanning period is very small, the algorithms tend to become more "reactive", while as the replanning period gets longer the algorithms get closer to offline planners. Figure 4.6 shows the time it takes to solve G4 in *door* for different replanning periods. Interestingly, as the periods get longer, performance deteriorates. The time to solve increases, and the success ratio drops. The results seem to indicate that the replanning period must be set to the shortest possible value, that is still long enough for the motion generation phase to be meaningful. An explanation for this rather unexpected result could be the aggressive nature of replanning. The robot is guided by the navigation function straight to the narrow passages and attempts to go through them multiple times. At the same time, an offline planner maintains one big tree and for the sake for completeness, it has to invest time on all branches of the tree. As the duration of the replanning period increases, the behavior of the algorithm tends to be closer to an offline planner - where the duration of a replanning period is ∞. Moreover, in the proposed scheme, having a replanning period T means that the robot has to move for T seconds before changing its plan. Due to its randomized nature, if the tree-based planner gets "unlucky" then in lack of better alternatives, the robot may need to chose a plan that drives it away from the goal for time T before it can replan and return back on track. Accumulated instances of this situation could explain the poor performance as the duration of the replanning period increases.



RP(seconds)	success
0.1	96 %
0.25	100 %
0.5	100 %
1	88 %
1.5	68 %
2	53 %
2.5	44 %
3	40 %
3.5	28 %
4.25	17 %

Figure 4.6: Time it takes *RPDST* to solve G4 in *door* for replanning periods of different duration and the success ratios for finding the solution in 10 minutes.

4.6 Partially Known and Changing Environments

The last experiments were done using the segway in an attempt to show that the proposed replanning scheme has potential to be used in harder problems where the environment is changing. The environment is shown in Figure 4.7. Although the map is given, the robot encounters unexpected obstacles such as closed doors and furniture that was not supposed to be there. The environment is also populated by moving obstacles whose motion is assumed to be fully predictable. 100 runs were averaged on this environment using *RPDST* . The average time to reach the goal was 55.88 seconds with a standard deviation of 7.23 seconds. This shows that the segway is consistent at its runtimes despite the difficulties presented by the environment. It is important to note that contingency plans were only selected 2.47% of the time, implying that the segway remains in motion almost all of the time and very rarely resorts to stopping. Finally, the presence of unexpected obstacles proves that indeed

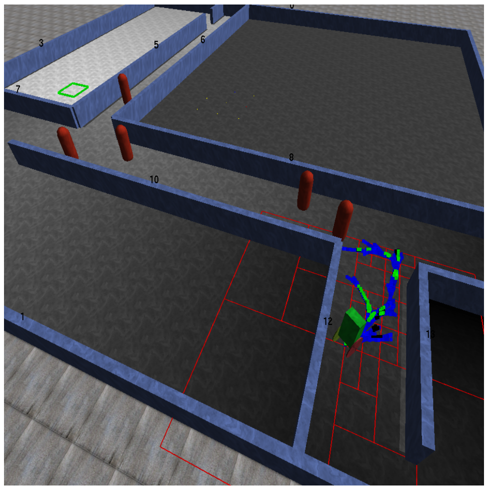


Figure 4.7: A segway that moves among unexpected obstacles and fully predictable moving obstacles.

the safety guarantees are lost. Specifically, the success rate was 73%.

Chapter 5

Discussion

This thesis investigates the benefits of continuous replanning over planning for problems with differential constraints in static and known environments. Replanning is a necessity whenever a robot has to move in a realistic setup for several reasons. It has to deal with uncertainties introduced by the motion errors, and this requires the incorporation of the latest sensor measurements that are collected and processed periodically. Even in an error-free case, replanning is necessary to take into account unexpected obstacles that might be encountered in a partially known environment and also moving obstacles. This work was motivated by these classes of problems. However, here the focus is on the interesting results for static, known environments with no uncertainty. In these cases, replanning is not enforced by the problem. However in the process of developing a powerful motion planner that deals with complex robot dynamic motion constraints, a choice was made to attempt to use replanning even though it is not technically needed.

A sampling-based motion planner is enclosed in a replanning loop and called periodically. It has limited amount of time to compute a rich set of possible motions.

Then through a selection phase, those motions are filtered to guarantee the safety of the robot (by avoiding Inevitable Collision States) and then the motion that better serves the robot's goal is executed. The result is a planning strategy that appears to have several advantages over existing offline planners. The algorithm is experimentally found to produce shorter solution paths than powerful offline sampling-based planners. Moreover, it is able to focus more on the important parts of the space (i.e., narrow passages) and manages to solve harder problems than the offline planners due to the guidance of the navigation function. Finally, the algorithm manages to do so, using only bounded amounts of memory.

It is interesting to see that the proposed algorithm uses a sampling-based planner in a somewhat different way. Instead of having it plan all the way to the goal, the planner is used as a local generator of feasible motions. In this sense, the tree-based planner is used to discretize the state space around the robot into a finite set of motion plans/actions. This potentially opens the door for higher level AI techniques for selecting the best actions from this set. The global solution comes as a result of concatenating a sequence of those small local plans and this highly resembles the idea of "divide and conquer".

One important question that arises in a replanning scheme is what the implications on the completeness properties of the algorithm are. First of all, it is clear that imposing time limitations on the tree-based planner breaks the probabilistic completeness guarantees that most offline sampling-based planners provide. This is related to the ability of an algorithm to backtrack and eventually search all the parts of the reachable space. To address this issue here, the navigation function was augmented with what was called a learning capability. This is just a simple mechanism for maintaining a rough estimate about how much each part of the workspace has been explored. The reason is that if certain areas have been visited often and the solution was not found, then the robot should try other options for approaching its goal. The penalization that the navigation function cells receive can thus be interpreted as an impatience factor that grows as the robot tries approach on a certain path without success. Whenever the penalty gets high enough, the robot backtracks. Although in all of the experiments the robot was never stuck, there is no proof that learning restores probabilistic completeness in the replanning framework.

To conclude, the benefits of replanning for static and known environments were an interesting result. Towards the direction of more realistic scenarios where there is the added difficulty of a partially known and changing environment, some initial and encouraging results were obtained with a segway. In particular, the segway did not appear to slow down due to the presence of moving obstacles whose trajectory could be predicted perfectly. The addition of unexpected obstacles had no effect the performance either. It did however show that in this case safety cannot be guaranteed overall due to the motion constraints. In terms of future work, it is important to try and refine the algorithm to accommodate uncertainties. Again, there is two dimensions, one related to the robot's intrinsic limitations (uncertainty in sensing and execution), and the other with respect to the environment, where the moving obstacles are generally unpredictable. Addressing those challenging problems could lead to more robust planning algorithms applicable to real robots.

Bibliography

[1] Jean-Pierre Aubin. *Viability Theory. Systems & Control: Foundations & Applications*. Birkhauser, 1991.

[2] Jerome Barraquand, Lydia Kavraki, Jean-Claude Latombe, Tsai-Yen Li, Rajeev Motwani, and Prabhakar Raghavan. A random sampling scheme for path planning. 1996.

[3] Jerome Barraquand, Bruno Langlois, and Jean-Claude Latombe. Numerical potential field techniques for robot path planning. *IEEE Transactions on Man and Cybernetics*, 22(2):224–241, 1992.

[4] Jerome Barraquand and Jean-Claude Latombe. Robot motion planning : A distributed representation approach. *International Journal of Robotics Research*, 10(6):628–649, 1991.

[5] Kostas E. Bekris and Lydia E. Kavraki. Greedy but safe replanning under kinodynamic constraints. In *IEEE International Conference on Robotics and Automation*, pages 704–710, Rome, Italy, April 2007. IEEE press.

[6] Kostas E. Bekris, Konstantinos I. Tsianos, and Lydia E. Kavraki. A decentralized planner that guarantees the safety of communicating vehicles with complex

dynamics that replan online. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2007.

[7] James Bruce and Manuela Veloso. Real-time multi-robot motion planning with safe dynamics. *Multi-Robot Systems: From Swarms to Intelligent Automata, Volume III*, 2006.

[8] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A. Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, June 2005. ISBN 0-262-03327-5.

[9] Stephen A. Ehmann and Ming C. Lin. Geometric algorithms: Accurate and fast proximity queries between polyhedra using convex surface decomposition. In *Computer Graphics Forum - Proc. of Eurographics*, volume 20, pages 500–510, 2001.

[10] Dave Ferguson, Nidhi Kalra, and Anthony Stentz. Replanning with RRTs. In *IEEE International Conference on Robotics and Automation*, pages 1243– 1248, 2006.

[11] Thierry Fraichard and Hajime Asama. Inevitable collision states: A step towards safer robots? In *Conference on Intelligent Robots and Systems*, 2003.

[12] Emilio Frazzoli, Munther A. Dahleh, and Eric Feron. Real-time motion planning for agile autonomous vehicles. In *American Control Conference*, volume 1, pages 43–49, 2001.

[13] Emilio Frazzoli, Munther A. Dahleh, and Eric Feron. Maneuver-based motion planning for nonlinear systems with symmetries. *IEEE Transactions on Robotics*, 21(6):1077–1091, December 2005.

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

[15] David Hsu, Jean-Claude Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. In *IEEE International Conference on Robotics and Automation*, volume 3, pages 2719–2726, April 1997.

[16] Yong K. Hwang and Narendra Ahuja. Gross motion planninga survey. *ACM Computing Surveys (CSUR)*, 24:219–291, 1992.

[17] Maciej Kalisiak. *Toward More Efficient Motion Planning with Differential Constraints*. PhD thesis, University of Toronto, 2007.

[18] Maciej Kalisiak and Michiel van de Panne. Rrt-blossom: Rrt with a local flood-fill behavior. In *International Conference on Robotics and Automation*, 2006.

[19] Lydia E. Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark H. Overmars. Probabilistic roadmaps for path planning in high dimensional configu-

ration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, August 1996.

[20] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *International Journal of Robotics Research*, volume 5, page 9098, 1986.

[21] J. Kim and J.P. Ostrowski. Motion planning of aerial robot usin rapidly-exploring random trees with dynamic constraints. In *IEEE Int. Conference on Robotics and Automation*, 2003.

[22] Sven Koenig and Maxim Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 968– 975, 2002.

[23] Prashanth Konkimalla and Steven M. LaValle. Efficient computation of optimal navigation functions for nonholonomic planning. In *Proceedings of the First Workshop on Robot Motion and Control*, pages 187–192, 1999.

[24] Andrew M. Ladd. *Direct Motion Planning over Simulation of Rigid Body Dynamics with Contact*. PhD thesis, Rice University, Houston, Texas, December 2006.

[25] Andrew M. Ladd and Lydia E. Kavraki. Motion planning in the presence of drift, underactuation and discrete system changes. In *Robotics: Science and Systems*, pages 233–241, Boston, MA, June 2005. MIT Press.

[26] Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. Fast proximity queries with swept sphere volumes. Technical Report TR99-018, Department of Computer Science, University of North Carolina at Chapel Hill, 1999.

[27] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.

[28] S. LaValle and J.J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In B.R. Donald, K. Lynch, and D. Rus, editors, *New Directions in Algorithmic and Computational Robotics*, pages 293–308. AK Peters, 2001.

[29] S.M. LaValle. From dynamic programming to rrts: Algorithmic design of feasible trajectories. In A. et al. Bicchi, editor, *Control Problems in Robotics*. Springer-Verlag, 2002.

[30] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

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

[32] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.

[33] Samer A. Masoud and Ahmad A. Masoud. Constrained motion control using vector potential fields. In *IEEE Transactions on Systems, Man, and CyberneticsPart A: Systems and Humans*, 2000.

[34] Kuaustubb Pathak, Jaume Franch, and Sunil K. Agrawal. Velocity and position control of a wheeled inverted pendulum by partial feedback linearization. In *IEEE Transactions on Robotics*, 2005.

[35] Stephane Petti and Thierry Fraichard. Safe motion planning in dynamic environments. 2005.

[36] Erion Plaku, Moshe Y. Vardi, and Lydia E. Kavraki. Discrete search leading continuous exploration for kinodynamic motion planning. In *Robotics: Science and Systems*, Atlanta, Georgia, 2007.

[37] John H. Reif. Complexity of the mover's problem and generalizations. In *IEEE Symposium on Foundations of Computer Science*, pages 421–427, 1979.

[38] Elon Rimon and Daniel E. Koditschek. Exact robot navigation using artificial potential functions. *IEEE Transactions onRobotics and Automation*, 8:501–518, 1992.

[39] Gildardo Sánchez and Jean-Claude Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. *International Journal of Robotics Research*, pages 403–407, 2003.

[40] Russell Smith. Open dynamics engine. http://www.ode.org. seen September 7, 2007.

[41] Konstantinos I. Tsianos, Ioan A. Şucan, and Lydia E. Kavraki. Sampling-based robot motion planning: Towards realistic applications. *Computer Science Review*, 1(1):2–11, August 2007.

[42] Richard Volpe and Pradeep Khosla. Manipulator control with superquadric artificial potential functions: Theory and experiments. In *IEEE Transactions on Systems, Man, and Cybernetics*, 1990.