Master's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Sampling-Based Motion Planning Using Burs of Free-Space

Bc. Jiří Hartvich

Supervisor: Ing. Vojtěch Vonásek, Ph.D.
May 2024

Acknowledgements

I would like to thank my supervisor Vojtěch Vonásek for being a guiding force in this work.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses. I also declare that ChatGPT was used as a helping hand in summarizing parts of this work.

Prague, May 23, 2024

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských prací. Rovněž prohlašuji, že bylo využito ChatGPT při shrnování částí této práce.

V Praze dne 23. května 2024

. . . .
Jiří Hartvich

Abstract

Motion planning is essential for enabling autonomous systems to navigate complex environments. This work focuses on improving and generalizing the Rapidly-exploring Bur Tree (RBT) algorithm for mobile robotic manipulators. Traditional methods like Rapidly-exploring Random Trees (RRT) and Probabilistic Road Maps (PRM) often struggle with planning in complex environments. While RBT has shown high exploratory capabilities, it was previously limited to stationary line-segment manipulators. We have adapted RBT for mobile robotic manipulators and statistically evaluated its performance against other planners, such as IK-RRT and J+RRT, in various grasping tasks. Our findings indicate that IK-RRT still outperforms other planners due to its efficient combination of inverse kinematics and bi-directional RRT. Although RBT demonstrated potential in scenarios with large free spaces, it underperformed in constrained environments due to the generalization of the algorithm and the complexity of the environments considered. This work provides insights into the conditions under which RBT and other planners are most effective, while also proposing potential improvements and solutions to the shortcomings of RBT.

Keywords: motion planning, mobile manipulators, grasping, robot kinematics

Supervisor: Ing. Vojtěch Vonásek, Ph.D.
Praha, Resslova 307/9 (vstup z Karlovo náměstí 13), místnost: E-121

Abstrakt

Plánování cest je nezbytné pro navigaci autonomních systémů ve složitých prostředích. Tato práce se zaměřuje na vylepšení a zobecnění algoritmu Rapidly-exploring Bur Tree (RBT) pro mobilní robotické manipulátory. Tradiční metody jako Rapidly-exploring Random Trees (RRT) a Probabilistic Road Maps (PRM) mají s plánováním ve složitých prostředích často potíže. Ačkoli RBT prokázalo vysokou úroveň průzkumných schopností, bylo dříve omezeno na stacionární robotické manipulátory v podobě řetězu úseček. Přizpůsobili jsme RBT pro mobilní robotické manipulátory a statisticky vyhodnotili jeho výkonnost v různých uchopovacích úlohách ve srovnání s jinými plánovači, jako jsou IK-RRT či J+RRT. Naše zjištění ukazují, že IK-RRT stále dosahuje lepších výsledků než ostatní plánovače díky efektivní kombinaci inverzní kinematiky a obousměrného RRT. Ačkoli RBT prokázalo potenciál ve scénářích s velkými prostory bez překážek, v omezených prostorech dosahovalo horších výkonů z důvodu zobecnění algoritmu a složitosti uvažovaných prostředí. Tato práce přináší poznatky o podmínkách, za kterých jsou RBT a další plánovače nejefektivnější, a zároveň navrhuje potenciální vylepšení RBT a řešení jeho nedostatků.

Klíčová slova: plánování cest, mobilní manipulátory, uchopování objektů, kinematika robotů

Překlad názvu: Rychlé plánování pohybu manipulátorů

Contents

Figures

Tables

Chapter 1

Introduction

Motion planning is fundamental to enabling machines and autonomous systems to navigate and perform tasks in complex environments. It involves finding ways to move an object or a robot from one point to another while avoiding obstacles and respecting various constraints such as speed limits, the geometry of the space and the dynamics of the moving object. This problem is often visualized through the Piano Mover's Problem, Figure 1.1a. It encapsulates the challenge of navigating through tight spaces without causing damage to the object or the environment.

Motion planning also plays a crucial role in robotics and automation, enabling mobile robots, drones and manipulators to perform a wide range of tasks. These tasks include assembling components, exploring underwater environments and interacting in social settings. In computer-aided design (CAD) systems and computer games, motion planning is essential for disassembly and assembly planning, maintainability studies and realistic character movements, enhancing both user experience and design efficiency.

In the physical world, motion planning algorithms have practical applications such as autonomous car-parking and large object transportation, an example of which is depicted in Figure 1.1b. A particularly active research area is planning for humanoid robots and robotic manipulators. Traditional planning methods often struggle with high-degree-of-freedom robots in cluttered environments, making motion planning an ongoing research challenge.

Methods like Rapidly-exploring Random Trees (RRT) and Probabilistic Road Maps (PRM) have been proposed to address motion planning tasks. While these methods perform well in high-dimensional spaces, they often fall short in terms of planning time for complex environments. To overcome these limitations, other methods such as planning for redundant manipulators and distance-information-based planners have been proposed. One such planner is Rapidly-exploring Bur Trees (RBT), which shows promise as a fast planner. RBT works well with high-degree-of-freedom manipulators and demonstrates better exploratory properties than RRT. However, it is currently only applicable to stationary line-segment manipulators. In this work, we aim to implement and generalize RBT for mobile robotic manipulators.

Motion planning algorithms typically plan a path between two joint configurations of a robot: a start and a goal configuration, each defined by a list

(a) : Piano mover. [4]

(b) : Friends moving a couch through a stairway. [5]

Figure 1.1: Large object motion planning examples.

of angles uniquely determining the robot's pose. We will address the task of planning from a starting robot configuration towards grasping objects in the environment. This task is particularly common in human environments where the exact joint configuration required to grasp an object is often unknown. Instead, the focus is on positioning the hand or gripper correctly to grasp the object. This task is more challenging than regular motion planning because the final joint angles and the reachable grasp from the starting configuration are not known in advance. We will have a set of grasping poses and a starting configuration from which we will aim to reach the goal in the shortest time possible.

Chapter 2
Related work

In this chapter, we delve into the realm of motion planning, a process that computes a sequence of permissible movements to transition from an initial state to a desired goal state. This concept is pivotal in the fields of robotics and automation. We will begin by laying out a clear definition of motion planning. Following that, we will examine two distinct paradigms within the discipline, taking a closer look at one to understand its implementation, advantages and disadvantages.

2.1 Motion planning definition

Motion planning is a computational problem to find a sequence of valid steps from a starting configuration to a goal configuration. Motion planning can either be performed in configuration space where the robot is represented as a point or in workspace where the geometry of the robot has to be taken into account. It is simpler to perform planning in configuration space because the robot is represented as a point in that is either in a collision-free configuration or in a colliding configuration. Collisions are determined in the workspace.

The workspace, denoted $\mathcal{W}$, is the the space where the robot operates. $\mathcal{W}$ is usually $\mathcal{W} \subseteq \mathbb{R}^2$ or $\mathcal{W} \subseteq \mathbb{R}^3$. A configuration $\mathbf{q}$ specifies every point of the robot such that $\mathcal{R}(\mathbf{q}) \subseteq \mathcal{W}$. $\mathbf{q}$ is usually a vector $\mathbf{q} = (q_1, q_2, \ldots, q_n)$ with n degrees of freedom, where n is the dimensionality of the configuration space. Configuration space $\mathcal{C}$ is the set of all possible configurations $\mathbf{q}$. Obstacles $\mathcal{O}$ are also a subset of the workspace $\mathcal{O} \subseteq \mathcal{W}$. A collision occurs when $\mathcal{O} \cap \mathcal{R}(\mathbf{q}) \neq \emptyset$.

Motion planning in robotics is the problem of finding a collision-free path $\tau(s)$ from a starting configuration $\mathbf{q}_{\text{init}} \in \mathcal{C}_{free}$ to a goal configuration $\mathbf{q}_{\text{goal}} \in \mathcal{C}_{free}$, or goal region $\mathcal{C}_{goal} \subseteq \mathcal{C}_{free}$, such that $\tau(s) \in \mathcal{C}_{free} \, \forall s \in [0, 1]$, where $\tau(0) = \mathbf{q}_{\text{init}}$, $\tau(1) \in \mathcal{C}_{goal}$.

Motion planning can also be formulated in terms of planning from a starting configuration $\mathbf{q}_{\text{init}}$ towards a goal region in the workspace $\mathcal{W}_{goal}$. The task is to find a collision-free path $\tau(s) \in \mathcal{C}_{free}$ from $\mathbf{q}_{\text{init}} \in \mathcal{C}_{free}$ to $\mathcal{W}_{goal} \subseteq \mathcal{W}_{free}$ such that $\tau(s) \in \mathcal{C}_{free} \, \forall s \in [0,\, 1]$, where $\tau(0) = \mathbf{q}_{\text{init}}$, $f_{\mathcal{C} \to \mathcal{W}}(\tau(1)) \in \mathcal{W}_{goal}$. The mapping from configuration space to workspace $f_{\mathcal{C} \to \mathcal{W}}$ is the robot's

Figure 2.1: Franka Emika Panda robot. [1]

forward kinematics such that

$$f_{\mathcal{C}\to\mathcal{W}}(\mathbf{q}) = \begin{bmatrix}\mathbf{R}_1 & \mathbf{t}_1\end{bmatrix}, \tag{2.1}$$

where $\mathbf{R}$ is the rotation of the end-effector and $\mathbf{t}$ is the translation of the end-effector in workspace $\mathcal{W}$ and $\mathbf{q}$ is the corresponding configuration. The end-effector is usually the gripper or last link of a robot manipulator. Motion planning for robotic manipulators is a common problem. Robotic manipulators typically involve six or more degrees of freedom; an example manipulator is depicted in Figure 2.1. It has been proven that motion planning is a P-Space complete problem which means that complexity significantly increases with the number of degrees of freedom [6], especially in environments with obstacles. Computing a feasible, let alone optimal solution, is computationally intensive.

Our task is to investigate motion planning in the context of workspace goals, similar to [7]. A type of workspace goal is for example a grasp — a 6D pose that consists of a rotation and translation. It is more comprehensively represented as a rotation matrix and translation vector pair $g = [\mathbf{R}, \mathbf{t}] \in G$. The grasp g maps to a configuration $\mathbf{q}_g \in \mathcal{C}$ in configuration space around which we can define a goal area. The goal area $\mathbf{Q}_G$ is the set of configurations that correspond to the set of goal grasps G:

$$\mathbf{Q}_G = \{\mathbf{q} \,|\, f_{\mathcal{C}\longrightarrow\mathcal{W}}(\mathbf{q}) \in G,\, \mathbf{q} \in \mathcal{C}\}, \tag{2.2}$$

where $f_{\mathcal{C}\longrightarrow\mathcal{W}}(\mathbf{q})$ is the aforementioned forward kinematics function from Equation 2.1. We assume that we are provided with the set of valid grasping poses $g \in G$, and our goal will be to construct a path to reach one of these

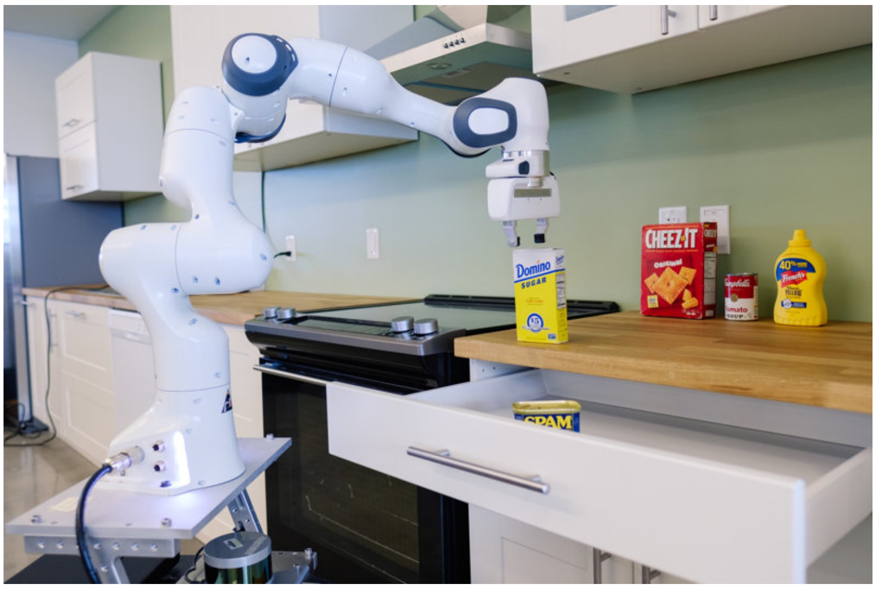


Figure 2.2: Franka Emika Panda robot arm grasping a box in a kitchen. [2]

poses with the end-effector of the manipulator. Grasping can be performed independently of planning and solutions are readily available in libraries such as the BURG toolkit [8], which builds on PyBullet [9] and has integrated visualization, physics simulation and grasp creation. The focus of this work is motion planning towards a set of grasps using a mobile robotic manipulator; the goal is for the end-effector to get very close to the target object without colliding with the environment. The manipulator we will be simulating is depicted in Figure 2.2 grasping a YCB sugar box [10].

Naive methods for motion planning with a set of grasps $g \in G$ as goals generally yield poor results [7]. In the following sections we will introduce existing paradigms for motion planning and evaluate their advantages, disadvantages and extensions for planning towards workspace goals.

2.1.1 RRT (Rapidly-exploring Random Trees)

The first approach to motion planning we introduce is Rapidly-exploring Random Trees (RRT) [11]. RRT explores configuration space efficiently by incrementally constructing a tree graph. Planning starts from the initial configuration $\mathbf{q}_{\text{init}} \in \mathcal{C}_{\text{free}}$ with the tree $\mathcal{T}$ initially containing only $\mathbf{q}_{\text{init}}$. RRT repeats the following steps: choose a random configuration $\mathbf{q}_{\text{rand}} \in \mathcal{C}$; find the configuration $\mathbf{q}_{\text{near}} \in \mathcal{T}$ closest to $\mathbf{q}_{\text{rand}}$; create a new configuration $\mathbf{q}_{\text{new}}$ that extends from $\mathbf{q}_{\text{near}}$ to $\mathbf{q}_{\text{rand}}$ by distance ε in configuration space; check if the path from $\mathbf{q}_{\text{near}}$ to $\mathbf{q}_{\text{new}}$ is collision-free; if it is collision-free, add the point $\mathbf{q}_{\text{new}}$ to the tree $\mathcal{T}$; repeat until the target configuration $\mathbf{q}_{\text{goal}} \in \mathcal{C}_{\text{free}}$ is reached withing a certain threshold; when it does find a path within the maximum amount of iterations, then it constructs a path by iterating from the goal node $\mathbf{q}_{\text{goal}}$ towards the root node $\mathbf{q}_{\text{init}}$ by taking the parent of each node and adding it to the path. The whole algorithm is described in Algorithm 1 and

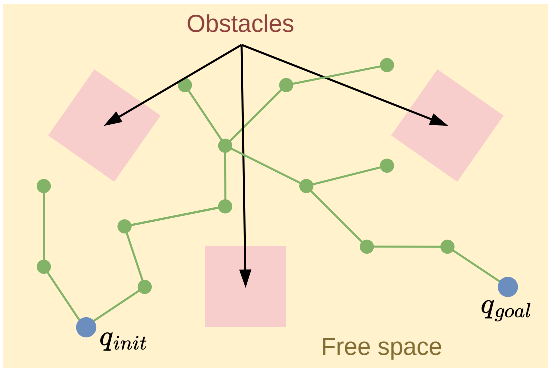


Figure 2.3: RRT example tree.

the Extend method that grows the tree to a new configuration is described in Algorithm 2. RRT has the added advantage that it is inherently biased towards exploring unexplored regions of configuration space [11]. Figure 2.3 illustrates an example RRT tree and Figure 2.4 shows how expansion works.

Algorithm 1 RRT($\mathbf{q}_{\text{init}}, \mathbf{q}_{\text{goal}}$)[12]

```
1: $\mathcal{T}_a$.Init($\mathbf{q}_{\text{init}}$)
2: $\mathcal{T}_b$.Init($\mathbf{q}_{\text{goal}}$)
3: for $k = 1$ to $K$ do
4:     $q_{\text{rand}} \leftarrow$ RandomConfig()
5:     if Extend($\mathcal{T}, \mathbf{q}_{\text{rand}}, \mathbf{q}_{\text{goal}}, \mathbf{q}_{\text{new}}$) = Reached then
6:         return Path($\mathcal{T}, \mathbf{q}_{\text{goal}}$)
7:     end if
8: end for
9: return Failure
```

The RRT Extend procedure, Algorithm 2, is the most commonly called routine in RRT, so it is often modified or called differently in various planners:

- RRTConnect uses two trees instead of one, each extending from $\mathbf{q}_{\text{init}}$ and $\mathbf{q}_{\text{goal}}$ respectively. With a small probability $p_{connect}$ it attempts to connect the two trees by repeatedly calling the Extend procedure until it collides, reaches joint limits or until some number of steps is reached [12].

- RRT* grows like regular RRT but during runtime it modifies the tree to create the shortest path [13].

- Other methods modify how the tree grows based on the environment. RRT-CT for example uses flooding — it extends orthogonally or diagonally in configuration space until it collides to cover an area as wide as possible [14].

- RBT (Rapid Bur Trees) replaces the RRT Extend procedure with a distance-based as opposed to collision-based extension function. For

Algorithm 2 EXTEND($\mathcal{T}, \mathbf{q}, \mathbf{q}_{\text{goal}}, \mathbf{q}_{\text{new}}$)[12], Figure 2.4.

```
1: $\mathbf{q}_{\text{near}} \leftarrow$ NearestNeighbour($\mathbf{q}, T$)
2: if NewConfig($\mathbf{q}, \mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}}$) then
3:     $\mathcal{T}$.Add($\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}}$)
4:     if $\mathbf{q}_{\text{new}} = \mathbf{q}_{\text{goal}}$ then
5:         return Reached
6:     else
7:         return Advanced
8:     end if
9: else
10:     return Trapped
11: end if
```

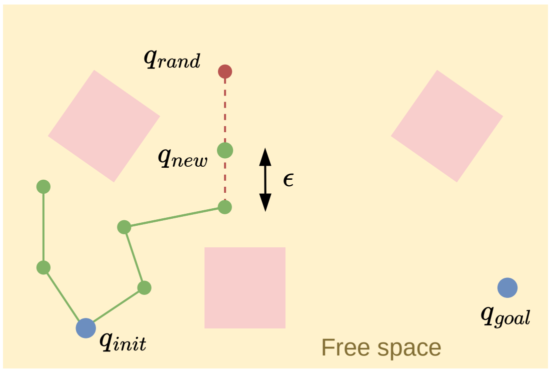


Figure 2.4: RRT extend procedure visualized in configuration space.

each distance query it is able to create several extensions of the tree at once [3].

2.1.2 PRM (Probabilistic RoadMaps)

Another popular approach to motion planning is Probabilistic RoadMaps (PRM), which proceeds in two phases. The first, learning phase consists of sampling in configuration space $\mathcal{C}$. Collision free configurations — nodes — are kept and then connected using edges that correspond to feasible paths between configurations. This graph is called the roadmap. In the second, query phase, any given start and goal configurations of the robot are connected to two nodes on the roadmap; the roadmap is then searched for a path joining these two nodes [15]. Figure 2.5 illustrates a probabilistic roadmap.

We will, however, focus solely on the RRT paradigm as it is the more popular of the two and exhibits better exploratory properties. The methods in the following sections are extensions of RRT that plan from a starting configuration $\mathbf{q}_{\text{init}}$ towards a set of grasps $g \in G$.

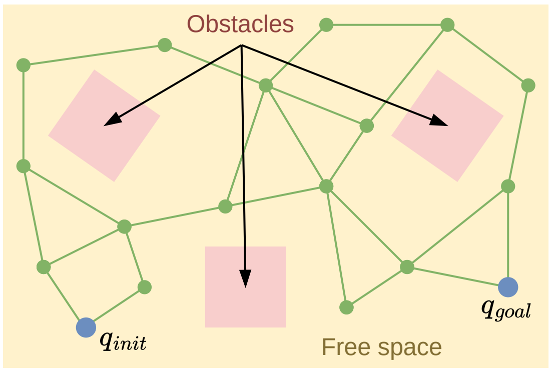


Figure 2.5: PRM example roadmap.

2.2 IK-RRT

IK-RRT [16] is a method that alternates between finding a within-joint-limits, collision-free inverse kinematics solution and bi-directional RRT. The authors propose a modified inverse kinematics method based on the reachability space of a humanoid robot. Unfortunately they do not provide an easy way to apply this to other robots. Moreover, the authors tested the planner with at most one obstacle. IK-RRT starts by trying to find a within-joint-limits, collision-free inverse kinematics solution, and only after it finds an initial solution does it start the planning process, similar to RRTConnect [12]. The first inverse kinematics solution $\mathbf{q}_{\text{goal}}$ might not be reachable from the initial configuration $\mathbf{q}_{\text{init}}$, so the planner alternates between planning and finding additional inverse kinematics solutions. The whole process is described in Algorithm 3.

2.3 J+RRT

J+RRT [16] is an extension of the basic RRT. It starts off like regular RRT, but with some probability p_{steer} it extends towards the workspace goal using ExtendToGoal. Inside ExtendToGoal, at every step, it uses the pseudo-inverse of the Jacobian to direct the end-effector towards a target workspace pose as opposed to a random configuration. The body of J+RRT is described in Algorithm 4 and ExtendToGoal is described in Algorithm 5.

2.4 Workspace RRT

Shkolnik et al. [17] explore how RRT can work in high-dimensional configuration spaces for highly redundant manipulators. Regular RRT in configuration space becomes less efficient the more redundant a manipulator becomes. In cases where the configuration space has possibly hundreds or thousands of dimensions, a step in configuration space possibly translates to an exceedingly

Algorithm 3 IK-RRT($\mathbf{q}_{\text{init}}, \mathbf{p}_{\text{obj}}, G$), $\mathbf{q}_{\text{init}}$ = start configuration, $\mathbf{p}_{\text{obj}}$ = object pose, G = set of grasp poses [16]

```
1:  $\mathcal{T}_1$.AddConfiguration($\mathbf{q}_{\text{init}}$)
2:  $\mathcal{T}_2$.Clear()
3:  while not TimeOut() do
4:      if #IKSolutions = 0 or Rand() $< p_{\text{IK}}$ then
5:          $\mathbf{p}_{\text{target}} \leftarrow$ GetRandomGrasp($G$)
6:          $\mathbf{q}_{\text{IK}} \leftarrow$ ComputeIK($\mathbf{p}_{\text{target}}$)
7:          if not Collision($\mathbf{q}_{\text{IK}}$) then
8:              $\mathcal{T}_2$.AddConfiguration($\mathbf{q}_{\text{IK}}$)
9:          end if
10:     else
11:         $\mathbf{q}_{\text{rand}} \leftarrow$ RandomConfig()
12:         if $\mathcal{T}_1$.Connect($\mathbf{q}_{\text{rand}}$) and $\mathcal{T}_2$.Connect($\mathbf{q}_{\text{rand}}$) then
13:             Solution $\leftarrow$ BuildSolutionPath($\mathbf{q}_{\text{rand}}$)
14:             return Path(Solution)
15:         end if
16:     end if
17: end while
18: return Failure
```

Algorithm 4 J+RRT($\mathbf{q}_{\text{init}}, \mathbf{p}_{\text{obj}}, G$), $\mathbf{q}_{\text{init}}$ = start configuration, $\mathbf{p}_{\text{obj}}$ = object pose, G = set of grasp poses [16]

```
1:  $\mathcal{T}$.AddConfiguration($\mathbf{q}_{\text{init}}$)
2:  while not TimeOut() do
3:      $\mathbf{q}_{\text{rand}} \leftarrow$ RandomConfig()
4:      $\mathbf{q}_{\text{near}} \leftarrow$ Nearest($\mathbf{q}_{\text{rand}}$)
5:      Extend($\mathcal{T}$, $\mathbf{q}_{\text{rand}}$, $\emptyset$, $\mathbf{q}_{\text{new}}$)
6:      if Rand() $< p_{\text{steer}}$ then
7:          Solution $\leftarrow$ ExtendToGoal($\mathcal{T}$, $\mathbf{p}_{\text{obj}}$, $G$)
8:          if Solution $\neq$ NULL then
9:              return PrunePath(Solution)
10:         end if
11:     end if
12: end while
13: return Failure
```

Algorithm 5 ExtendToGoal($\mathcal{T}$) [16]

```
1:  $\mathbf{p}_{\text{target}} \leftarrow$ GetRandomGrasp($G$)
2:  $\mathbf{q}_{\text{near}} \leftarrow$ GetNearestNeighbor($\mathcal{T}$, $\mathbf{p}_{\text{target}}$)
3:  do
4:      $\mathbf{p}_{\text{near}} \leftarrow$ ForwardKinematics($\mathbf{q}_{\text{near}}$)
5:      $\Delta_{\mathbf{p}} \leftarrow \mathbf{p}_{\text{target}} - \mathbf{p}_{\text{near}}$
6:      $\Delta_{\mathbf{q}} \leftarrow J^{+}(\mathbf{q}_{\text{near}}) \cdot \Delta_{\mathbf{p}}$
7:      $\mathbf{q}_{\text{new}} \leftarrow \mathbf{q}_{\text{near}} + \varepsilon \cdot \text{Normalize}(\Delta_{\mathbf{q}})$
8:      if Collision($\mathbf{q}_{\text{near}}$, $\mathbf{q}_{\text{new}}$) or not InJointLimits($\mathbf{q}_{\text{new}}$) then
9:          return NULL
10:     end if
11:     $\mathcal{T}$.Add($\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}}$)
12:     $\mathbf{q}_{\text{near}} \leftarrow \mathbf{q}_{\text{new}}$
13: while Length($\Delta_{\mathbf{p}}$) > ThresholdCartesian
14: return BuildSolutionPath($\mathbf{q}_{\text{near}}$)
```

small change in workspace:

$$f_{\mathcal{C}\to\mathcal{W}}\left(\mathbf{q} + \varepsilon \cdot \Delta_{\mathbf{q}}\right) \xrightarrow[n\to\infty]{} f_{\mathcal{C}\to\mathcal{W}}(\mathbf{q}), \;\; \mathbf{q}, \mathbf{q}_r \in \mathbb{R}^n, \;\; \varepsilon \in \mathbb{R}, \tag{2.3}$$

where $f_{\mathcal{C}\to\mathcal{W}}$ is the forward kinematics mapping, the configuration space is $\mathcal{C} = \mathbb{R}^n$ and $\Delta_{\mathbf{q}}$ is a random direction, $\|\Delta_{\mathbf{q}}\| = 1$. Moreover, the more redundant the manipulator, the more likely it is that joint motions cancel each other out. To tackle this, Shkolnik et al. propose a workspace-based RRT that keeps track of its nodes in workspace and for each pose in workspace it keeps a corresponding node in configuration space. The tree is grown like regular RRT, but instead of growing towards a random configuration $\mathbf{q}_r$, it is grown from $\mathbf{p}_{\text{near}} = f_{\mathcal{C}\to\mathcal{W}}(\mathbf{q}_{\text{near}})$ in a random direction $\Delta_{\mathbf{p}}$. It is directed from $\mathbf{q}_{\text{near}}$ toward $f_{\mathcal{C}\to\mathcal{W}}(\mathbf{q}_{\text{near}}) + \Delta_{\mathbf{p}}$ using the pseudo-inverse of the Jacobian at point $\mathbf{q}_{\text{near}}$. In addition, to avoid clumping, which increases the likelihood of redundant movement, null-space control is used to keep the manipulator relatively straightened out or close to a desired shape:

$$\dot{\mathbf{q}} \leftarrow \mathbf{J}^{+} \cdot \varepsilon \cdot \text{Normalize}(\Delta_{\mathbf{p}}) + \alpha(\mathbf{I} - \mathbf{J}^{+}\mathbf{J}) \cdot \dot{\mathbf{q}}_{\text{ref}}, \tag{2.4}$$

where, ε is the desired step size in the workspace, $\mathbf{J}$ is the Jacobian, $\mathbf{J}^{+}$ is its pseudo-inverse, $\dot{\mathbf{q}}_{\text{ref}}$ is the reference direction which is to be approached as a secondary goal, α is the weight parameter for the secondary goal and $\dot{\mathbf{q}}$ is the resulting configuration-space update.

2.4.1 RRT advantages and disadvantages

RRT in general is a good algorithm for planning. It quickly expands into unexplored configuration space, it is biased towards unexplored areas, and at every step it is connected by a path to the root node. However, exploration can be slow for environments with many obstacles or in constrained spaces. A configuration space with a narrow passage caused by obstacles is depicted in

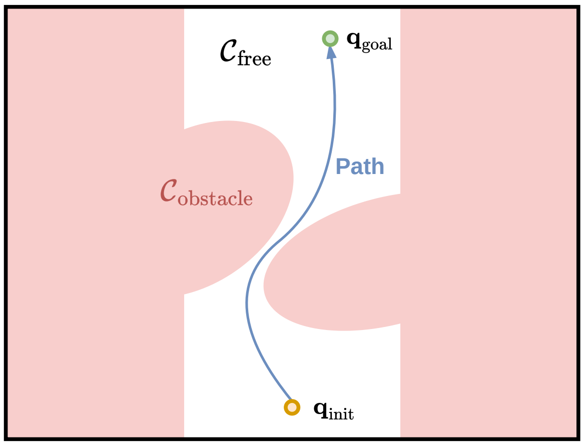

$\mathcal{C}_{\text{free}}$
$\mathbf{q}_{\text{goal}}$
Path
$\mathcal{C}_{\text{obstacle}}$
$\mathbf{q}_{\text{init}}$

Figure 2.6: Narrow passage visualized in configuration space.

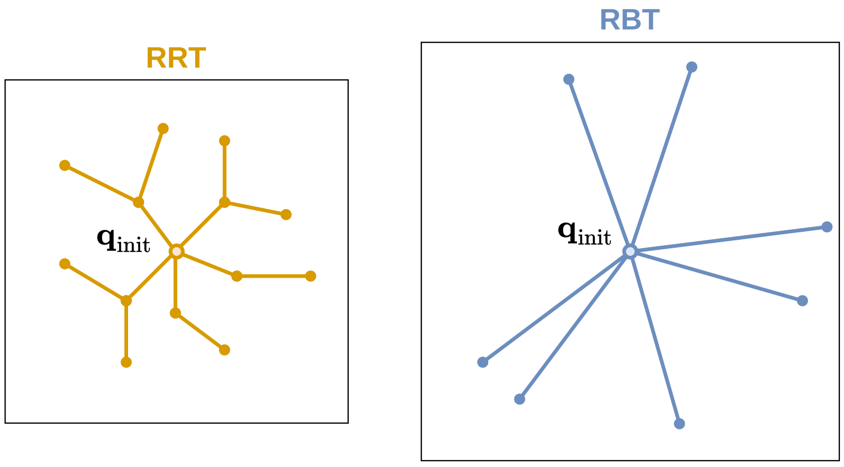
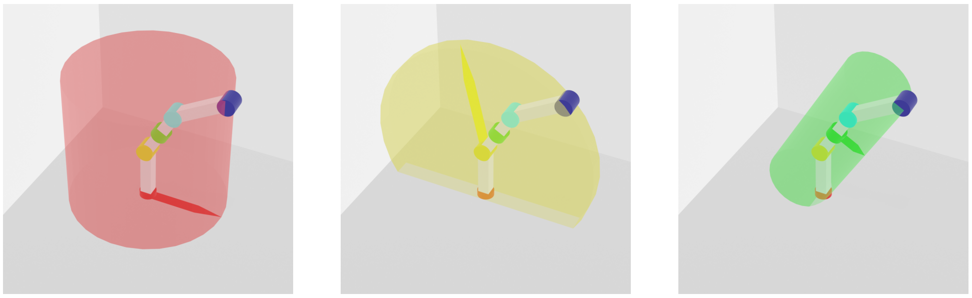


Figure 2.7: An RRT tree compared with a single step in RBT. Both are grown from an initial configuration.

Figure 2.6. To solve the issue of quick exploration, Lacevic et al. [3] propose RBT (Rapid Bur Trees), which replaces collision checking with distance queries between the environment and the robot. To gain an edge over regular RRT, the authors use the distance information to build a new object called a bur of free space which is based on complete bubbles of collision-free space that cover a significantly larger area than single RRT extension. An example of what the “bur” Lacevic et al. propose looks like next to several RRT extensions is depicted in Figure 2.7. The collision-free area defined by a distance query is usually significantly larger than one or even several RRT extensions.

2.5 Burs

A bur of free space builds on the concept of bubbles of free space from [18]. A bubble of free space at configuration $\mathbf{q}$ is a volume of configuration space that



(a) : Radius of the base joint.

(b) : Radius of the second joint.

(c) : Radius of the third joint.

Figure 2.8: Cylinders encompassing all links of the robot.

is guaranteed to be collision-free given the distance to the nearest obstacle d_c. The distance to the closest obstacle is defined as the smallest distance between every segment of the robot and every obstacle in the environment:

$$d_c = \min_{i=1\ldots n} \left(\min_{j=1\ldots m} \mathbf{d}_{ij} \right), \tag{2.5}$$

where i is a segment on the robot and j is an obstacle, n is the number of segments on the robot and m is the number of obstacles. Using the distance to the closest obstacle and the forward kinematics of the robot, we can construct a bubble of free space:

$$\mathcal{B}(\mathbf{q}, d_c) = \left\{ \mathbf{y} \;\middle|\; \sum_{i=1}^{n} \mathbf{r}_i \left| \mathbf{y}_i - \mathbf{q}_i \right| < d_c,\ \mathbf{y} \in \mathcal{C}_{free} \right\}, \tag{2.6}$$

where $\mathbf{q}$ is the starting configuration, $\mathbf{y}$ is the configuration that is collision-free inside the bubble, $\mathbf{r}_i$ is the radius of the cylinder that it collocated with joint i encompassing all subsequent links of the robot, d_c is the closest distance from the robot to any obstacle. Encompassing radii $\mathbf{r}_i$ for several joints on an example robotic manipulator are illustrated in Figure 2.8.

Extending the tree graph using bubbles of free space turns out not to provide any significant benefits to planning time [19]. That is why Lacevic et al. [3] propose a more general notion of distance between two configurations to create larger, complete bubbles. They define the distance between two configurations to be the largest displacement of any point on the robot:

$$\rho_{\mathcal{R}}(\mathbf{q}_1, \mathbf{q}_2) = \max_{\mathbf{p} \in \mathcal{R}} \left\| f_{\mathbf{p}}(\mathbf{q}_1) - f_{\mathbf{p}}(\mathbf{q}_2) \right\|, \tag{2.7}$$

where $f_{\mathbf{p}}(\mathbf{q})$ denotes the forward kinematics mapping of configuration $\mathbf{q}$ to point $\mathbf{p}$ on robot $\mathcal{R}$. In other words, the metric function $\rho_{\mathcal{R}}$ calculates the largest displacement that any point on the robot can achieve when moved from configuration $\mathbf{q}_1$ to configuration $\mathbf{q}_2$.

The complete bubble $\mathcal{CB}$, which uses this generalized notion of distance $\rho_{\mathcal{R}}$, Equation 2.7, is then defined as:

$$\mathcal{CB}(\mathbf{q}, d_c) = \left\{ \mathbf{x} \in \mathcal{C}_{free} \mid \rho_{\mathcal{R}}(\mathbf{q}, \mathbf{y}) < d_c,\ \forall \mathbf{y} \in \overline{\mathbf{qx}} \right\}, \tag{2.8}$$

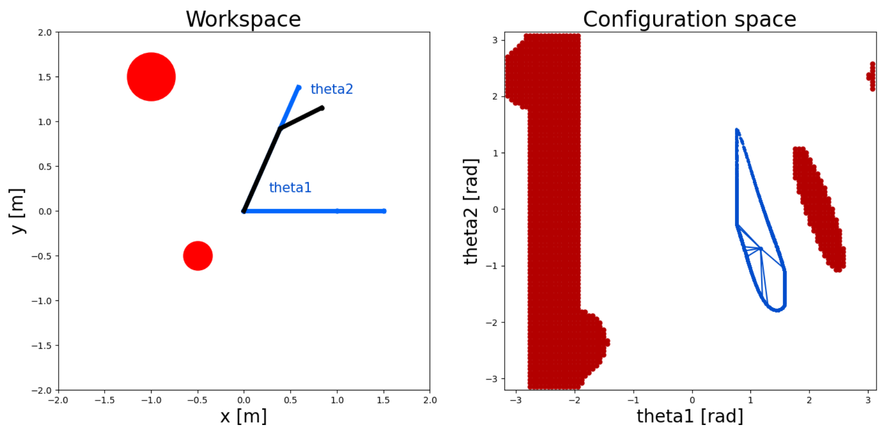


Figure 2.9: A two segment robot and a bur in the same configuration in 2D configuration space.

where d_c is the closest distance from the robot to any obstacle, $\mathbf{q}$ is the starting configuration, $\mathcal{C}_{free}$ is collision-free configuration space and $\mathbf{y}$ is any configuration inside the bubble.

Lacevic et al. 2016 [3] prove that the complete bubble is star-convex, meaning that its boundary can be reached by a straight line from its center point in configuration space. A bur is a subset of the complete bubble.

- **Definition (Bur):** *A bur is a star-convex subset of the complete bubble: $\mathcal{B}ur \subset \mathcal{CB} \subseteq \mathcal{C}$ consisting of the center-point $\boldsymbol{q}_{center}$ and a set of endpoints $\boldsymbol{Q}_e$. In other words, a $\mathcal{B}ur$ in configuration $\boldsymbol{q}_{center}$ is the center-point mapped to its child nodes $\mathcal{B}ur(\boldsymbol{q}_{center}) = \{\boldsymbol{q}_{center}, \{\boldsymbol{q}_{e_1}, \dots, \boldsymbol{q}_{e_n}\}\}$.*

- **Definition (Spine):** *A spine is a line segment that connects the center of the bur and its endpoint.*

An example bur of a two-line-segment model robot with its border highlighted in configuration space is depicted in Figure 2.9.

Lacevic et al. [3] propose a method by which to compute a bur of free space. Since a bur consists of disjunct spines — except the center point, which they all share — it means that each spine can be computed independently of others. A spine is calculated iteratively by calculating the edge-point $\mathbf{q}_{k+1}$ of the collision-free bubble $\mathcal{B}(\mathbf{q}_k)$ from Equation 2.6 like so:

$$t_{k+1}(t_k, \mathbf{q}_e, \mathbf{q}_k) = t_k + \frac{\varphi(t_k)}{\sum_{i=1}^{n} \mathbf{r}_i(\mathbf{q}_k)|\mathbf{q}_e^i - \mathbf{q}_k^i|}(1 - t_k), \tag{2.9}$$

where $t_k \in [0,\, 1]$ is the progress parameter value after step k, $\mathbf{q}_e$ is the endpoint, $\mathbf{q}_k$ is the configuration in step k: $\mathbf{q}_k = \mathbf{q} + t_k(\mathbf{q}_e - \mathbf{q})$ and $\mathbf{r}$ is the vector of radii of cylinders collocated with each joint encompassing all subsequent links of the robot, Figure 2.8. The parameter t_k starts at $t_k = 0$ which corresponds to $\mathbf{q}_{\text{near}}$; the endpoint of the bur $\mathbf{q}_{\text{endpoint}}$ ends up in some arbitrary value $t_{\text{endpoint}} \in (0,\, 1]$.

$\varphi(t)$ is the function

$$\varphi(t) = d_c - \rho_{\mathcal{R}}\left(\mathbf{q}, \mathbf{q} + t \cdot (\mathbf{q}_e - \mathbf{q})\right), \tag{2.10}$$

where $\mathbf{q}_e$ is the target configuration, $\mathbf{q}$ is the starting configuration from which the distance d_c to the closest obstacle was calculated.

A reasonable number of steps k or a distance criterion can be chosen to stop iteration. In [3], five steps are given as the threshold after which to stop iterating. From Equation 2.9 we still need a way to obtain the radius $\mathbf{r}_i$ of the

Algorithm 6 Bur($\mathbf{q}_{\text{near}}$, $\mathbf{Q}_e$, d_c), $\mathbf{Q}_e$ = target endpoints, d_c = distance to closest obstacle, Figure 2.9

```
Endpoints ← {}
for i=1 to Length($\mathbf{Q}_e$) do
    for k=1 to 5 do
        $t_k \leftarrow t_{k+1}(t_k, \mathbf{Q}_e(i), \mathbf{q}_k)$                ▷ Function from Equation 2.9
        $\mathbf{q}_k \leftarrow \mathbf{q}_{\text{near}} + t_k \cdot (\mathbf{Q}_e(i) - \mathbf{q}_{\text{near}})$    ▷ Radii $\mathbf{r}_k$ are calculated in $\mathbf{q}_k$
    end for
    Endpoints.Add($\mathbf{q}_k$)
end for
return {$\mathbf{q}_{\text{near}}$, Endpoints}
```

cylinder collocated with the joint i encompassing all further links of the robot. Radii of the cylinders encompassing all links of an example manipulator are depicted in Figure 2.8. Lacevic et al. [3] calculate the radius $\mathbf{r}_j$ for each joint j by projecting the locations $\mathbf{p}_i$ of all segments i on the plane of the joint j and then taking the maximum norm of those vectors:

$$\mathbf{r}_j = \max_{i=m\ldots n} \operatorname{proj}_{\mathbf{v}_j^{\perp}}(\mathbf{p}_i - \mathbf{p}_j), \tag{2.11}$$

where $\mathbf{v}_j$ is the axis of joint j, $\mathbf{p}_i$ is the location of segment i, $\mathbf{p}_j$ is the location of joint j, m is the next segment connected to joint j and n is the number of segments.

2.5.1 Assumptions

Lacevic et al. [3] computed burs for a robotic manipulator with two assumptions in mind:

1. the robot is represented by a chain of line segments, as can be seen for an example two-segment line-robot in the workspace part of Figure 2.9;
2. the robot is a manipulator with only revolute joints.

In reality these assumptions are a special case of capsule robots. Lacevic et al. [3] tested the algorithm on “real” robots, i.e., relatively compact, rotationally symmetric manipulators *without* end-effectors. These robots more or less satisfy the hidden assumption that the authors did *not* mention. Figure 2.10 depicts how the collision-free area of a line segment is identical to that of a capsule.

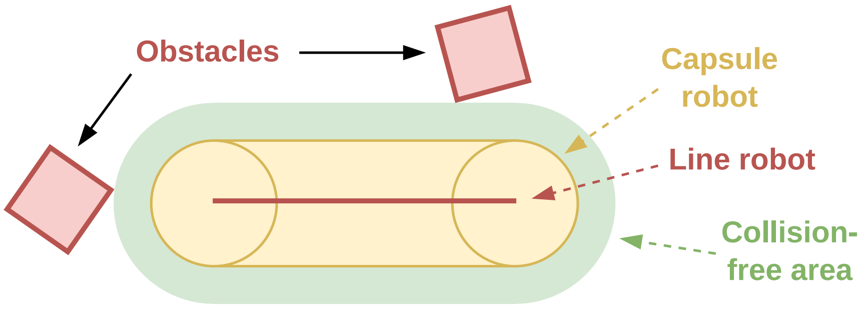


Figure 2.10: Collision-free area of a line segment and of a capsule.

2.5.2 RBT for manipulators

Rapidly-exploring Bur Tree (RBT) is a planning algorithm based on the RRT paradigm in that every new node is connected to its previous node with a collision-free edge. When the distance to the closest obstacle d_c is larger than d_{crit}, then it grows the tree using a bur of free space, Equation 2.9. In the case when the distance to the closest obstacle is smaller than d_{crit}, then the algorithm switches to regular RRT Extend, Algorithm 2. The structure of the algorithm is based on RRTConnect [12]. It uses two trees, one starting in the starting configuration $\mathbf{q}_{\text{init}}$ and the other tree starting in $\mathbf{q}_{\text{goal}}$, growing both of them using burs of free space, Algorithm 7. After every extension it attempts to connect these two trees using a custom connect algorithm that uses a bur, but with a single spine at a time, Algorithm 8. BurConnect stops when a special connection heuristic is satisfied, Algorithm 8 Line 14.

2.5.3 RBT for rigid bodies

The bur computation in the previous section is only valid under the assumption that the robot is a line-segment model. For rigid bodies, the concept of a kinematic chain does not apply. Lacevic et al. 2018 [20] propose a method how to calculate burs for rigid bodies in Lie groups SE(2) and SE(3). A rigid body is characterized by its geometry $\mathcal{G}$, translation $\mathbf{t}$ and rotation ϕ. A rigid body must be represented by all of its geometric points in addition to their translations and rotations because otherwise situations like the one in Figure 2.11 become possible. If it were represented only by its center point like in the line-segment model from the previous section then the center of the robot would be able to stay still while the rest of the robot rotates freely about its axis of rotation.

Lacevic et al. 2018 [20] introduce a generalized collision-free bubble for a rigid body. The original bubble of free space for 2D objects in SE(2), as defined in Quinlan 1995 [18] is

$$\mathcal{B}(\mathbf{t}, \phi, d_c) = \left\{ \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \;\middle|\; \left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} \mathbf{t}_x \\ \mathbf{t}_y \end{bmatrix} \right\|_2 + r_3|\theta - \phi| < d_c,\ \mathbf{y} \in \mathcal{C}_{free} \right\}, \tag{2.12}$$

Algorithm 7 RBT-CONNECT($\mathbf{q}_{\text{init}}$, $\mathbf{q}_{\text{goal}}$) [3]

```
1:  $\mathcal{T}_a \leftarrow \mathbf{q}_{\text{init}}, \mathcal{T}_b \leftarrow \mathbf{q}_{\text{goal}}$
2:  for $k = 1$ to $k_{\text{max}}$ do
3:      $\mathbf{Q}_e \leftarrow \{\}$
4:      for $i = 1$ to $N$ do
5:          $\mathbf{q}_{e_i} \leftarrow$ RandomConfig()
6:          $\mathbf{Q}_e$.Add($\mathbf{q}_{e_i}$)
7:      end for
8:      $\mathbf{q}_{\text{near}} \leftarrow$ Nearest($\mathbf{q}_{e_1}, \mathcal{T}_a$)
9:      for $i = 1$ to $N$ do                         ▷ Offset target configurations for bur
10:         $\mathbf{Q}_e(i) \leftarrow \delta \cdot$ Normalize($\mathbf{Q}_e(i) - \mathbf{q}_{\text{near}}$)          ▷ $\delta$ = joint range
11:     end for
12:     for $i = 1$ to $N$ do
13:         if $d_c(\mathbf{q}_{\text{near}}) < d_{\text{crit}}$ then
14:             if Extend($\mathcal{T}_a, \mathbf{q}_{\text{near}}, \emptyset, \mathbf{q}_{\text{new}}$) = Trapped then    ▷ Algorithm 2
15:                 continue
16:             end if
17:         else
18:             $\mathcal{T}_a$.Add(Bur($\mathbf{q}_{\text{near}}, \mathbf{Q}_e, d_c(\mathbf{q}_{\text{near}})$))
19:             $\mathbf{q}_{\text{new}} \leftarrow$ Endpoint(Bur($\mathbf{q}_{\text{near}}, \mathbf{Q}_e, d_c$), $\mathbf{q}_{e_1}$)
20:             if BurConnect($\mathcal{T}_b, \mathbf{q}_{\text{new}}$) = Reached then        ▷ Algorithm 8
21:                 return Path($\mathcal{T}_a, \mathcal{T}_b$)
22:             end if
23:             Swap($\mathcal{T}_a, \mathcal{T}_b$)
24:         end if
25:     end for
26: end for
27: return Failure
```

where $\mathbf{t}$ is the translation, ϕ is the rotation of the robot. r_3 represents the maximum distance from the origin of the robot to any other point on the robot, Figure 2.12.

The collision free bubble for SE(3) is then defined as

$$\mathcal{B}(\mathbf{q}, d_c) = \left\{ \mathbf{y} \;\middle|\; \left\| \mathbf{y}_p - \mathbf{q}_p \right\|_2 + \mathbf{r}_\phi^T |\mathbf{y}_\phi - \mathbf{q}_\phi| < d_c, \, \mathbf{y} \in \mathcal{C}_{free} \right\}, \tag{2.13}$$

where $\mathbf{r}_\phi^T = [\mathbf{r}_x \; \mathbf{r}_y \; \mathbf{r}_z]$ are the radii of enclosing cylinders of the robot with respect to its roll, pitch, and yaw axes respectively, the three cylinders are illustrated in Figure 2.13. The complete collision-free bubble $\mathcal{CB}$ from Equation 2.8 stays unchanged. Calculating $\mathbf{r}_\phi$ for each rigid body can be done before planning, as is obvious from Figure 2.13. The formula for calculating radii is simply largest magnitude of the projections of all points on the three

Algorithm 8 BUR-CONNECT($\mathcal{T}, \mathbf{q}$) [3]

```
 1: $\mathbf{q}_n \leftarrow \text{Nearest}(\mathbf{q}, \mathcal{T})$, $\mathbf{q}_0 \leftarrow \mathbf{q}_n$
 2: repeat
 3:     if $d_c(\mathbf{q}_n) > d_{\text{crit}}$ then
 4:         $\mathbf{q}_t \leftarrow \text{Endpoint}(\text{Bur}(\mathbf{q}_n, \mathbf{q}, d_c(\mathbf{q}_n)), \mathbf{q})$
 5:         $\Delta_s \leftarrow |\mathbf{q}_t - \mathbf{q}_n|$
 6:         $\mathbf{q}_n \leftarrow \mathbf{q}_t$
 7:         if $\mathbf{q}_n = \mathbf{q}$ then
 8:             return Reached
 9:         end if
10:     else
11:         if $\text{Extend}(\mathcal{T}, \mathbf{q}_n, \mathbf{q}, \mathbf{q}_{\text{new}}) = \text{Trapped}$ then
12:             return Trapped
13:         end if
14:         if $|\mathbf{q}_n - \mathbf{q}_0| \geq |\mathbf{q} - \mathbf{q}_0|$ then
15:             return Reached
16:         end if
17:     end if
18: until $\Delta_s < \text{Threshold}$
19: return Trapped
```

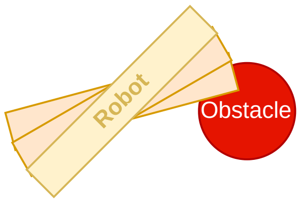


Figure 2.11: Rigid body rotational collision.

yz, xz, xy planes:

$$\mathbf{r} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \begin{bmatrix} \max_{\mathbf{p}\in\mathcal{G}} \sqrt{\mathbf{p}_y^2 + \mathbf{p}_z^2} \\ \max_{\mathbf{p}\in\mathcal{G}} \sqrt{\mathbf{p}_x^2 + \mathbf{p}_z^2} \\ \max_{\mathbf{p}\in\mathcal{G}} \sqrt{\mathbf{p}_x^2 + \mathbf{p}_y^2} \end{bmatrix}, \tag{2.14}$$

where $\mathcal{G}$ is the geometry of the rigid body, $\mathbf{p} \in \mathcal{G}$ is any point on the geometry.

Computing rigid body burs

Rigid body burs end up having the same general form as line-segment model burs. If we take it directly from Lacevic et al. [20], Equation 2.9 becomes

$$t_{k+1} = t_k + \frac{\varphi(t_k)}{g(\mathbf{q}_e, \mathbf{q}(t_k))}(1 - t_k), \tag{2.15}$$

where

$$g(\mathbf{y}, \mathbf{q}) = \|\mathbf{y}_p - \mathbf{q}_p\|_2 + \mathbf{r}_\phi^T |\mathbf{y}_\phi - \mathbf{q}_\phi|. \tag{2.16}$$

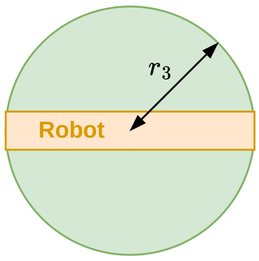
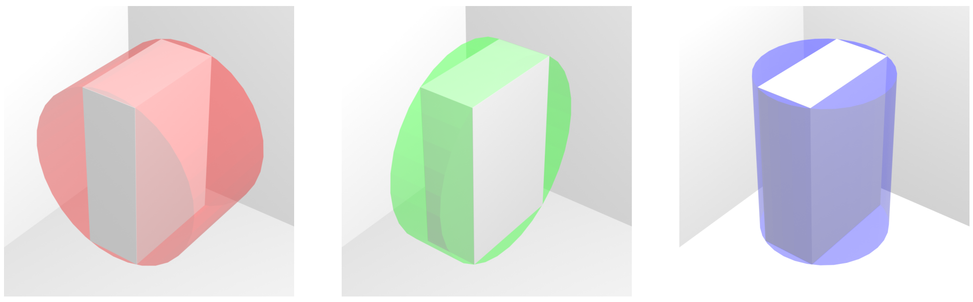


Figure 2.12: Rigid body radius in 2D.



(a) : Radius about x-axis. **(b) :** Radius about y-axis. **(c) :** Radius about z-axis.

Figure 2.13: Cylinders encompassing a rigid body along xyz-axes.

Note that in Equation 2.15, the function $\varphi(t)$ stays the same as in Equation 2.10. $\varphi(t) = d_c - \rho_{\mathcal{R}}(\mathbf{q}_1, \mathbf{q}_2)$, which means that $\rho_{\mathcal{R}}$, Equation 2.7, works even for rigid bodies. For *all* points $\mathbf{p}$ on the robot $\mathcal{R}$ and the translations of each of those points it takes the largest of those translations to be the final distance between configurations. A visualization of the calculation is depicted in Figure 2.14.

2.5.4 RBT extensions

The following two algorithms are extensions of the RBT algorithm again by Lacevic et al. The first one changes the number of spines in a bur based on the distance to the closest obstacle [21], thus improving planning efficiency; the second one assumes obstacles are convex and approximates them with planes and recalculates the estimate of the distance to the closest obstacle using these planes when it reaches the edge of the original bur [22].

Adaptive RBT

Adaptive RBT by Lacevic et al. [21] creates a bur with a number spines proportional to the distance to the nearest obstacle d_c. All formulas presented perform comparably well, so we will only present the simplest one:

$$f_1(d_c) = N_{crit} + \frac{N_s - N_{crit}}{d_s - d_{crit}}(d_c - d_{crit}), \tag{2.17}$$

where $f_1(d_c)$ is the number of spines for a given distance d_c to the nearest obstacle, N_{crit} is the minimum number of spines e.g., $N_{crit} = 2$, N_s is the

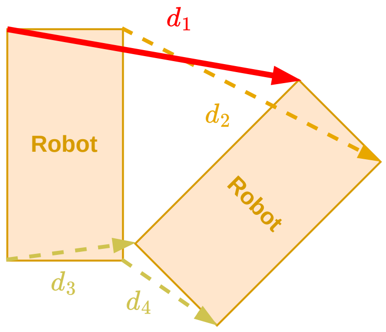


Figure 2.14: Rigid body maximum movement.

number of the optimal number of spines, e.g. $N_s \approx 7$ from [3], d_s is the distance to the closest obstacle that is to be expected most commonly. This brings a planning-time speed-up of approximately 25% compared to the plain version of RBT [21]. However, the authors tested the algorithm only for chain-of-line-segments revolute-joint manipulators.

RGBT

Rapidly-exploring Generalized Bur Tree (RGBT) [22] assumes that obstacles are convex. This means that in the worst case an obstacle is an infinite plane:

$$\mathcal{A}_{i,j} = \left\{ \mathbf{x} \middle| \left(\mathbf{p}_{i,j}^2 - \mathbf{p}_{i,j}^1\right)^T \left(\mathbf{x} - \mathbf{p}_{i,j}^2\right) = 0 \right\}, \tag{2.18}$$

where $\mathbf{p}_{i,j}^2$ is the point on obstacle j that is closest to segment i, $\mathbf{p}_{i,j}^1$ is the point on segment i that is closest to segment j. The planar approximation of a cube obstacle is illustrated in Figure 2.15.

Normally, a bur is created using the initial query of distance d_c and extended until any part of the robot moves that distance. In this case, however, when the edge of the complete bubble boundary is reached, i.e., the regular bur is created, one can switch to calculating the distance using the point-to-plane distance formula. As long as segment i stays on the inside of all planes $\mathcal{A}_{i,j}$ defined by it and all obstacles j, then this algorithm is able to extend to that free area in a single step. Now that we consider all segments i and all obstacles j, then the Equation 2.9 changes appropriately so no single segment can reach any obstacle during the creation of a spine:

$$t_{k+1} = t_k + \frac{\min_i \left(\left(\min_j d_{approx}^{j,i} \right)_i - \rho_{\mathcal{R}}(\mathbf{q}_k)_i \right)}{\sum_{i=1}^n \mathbf{r}_i(\mathbf{q}_k) |\mathbf{q}_e^i - \mathbf{q}_k^i|} (1 - t_k), \tag{2.19}$$

where i is the i-th segment, j is the j-th obstacle, $\rho_{\mathcal{R}}$ is the metric distance function from Equation 2.7, $d_{approx}^{j,i}$ is the distance between segment i and obstacle j. The authors recalculate the distance estimate $d_c \approx \min_j d_{approx}^{j,i}$ every five iterations so as not to make the general bur calculation too computationally expensive.

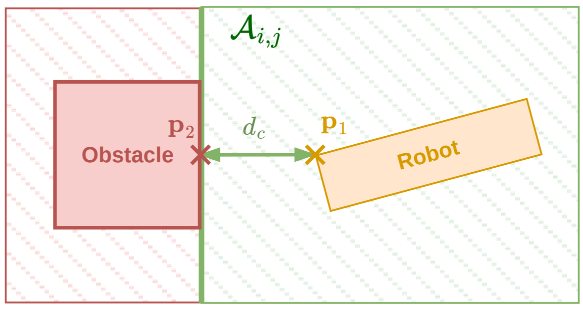


Figure 2.15: Planar approximation of an obstacle.

2.6 BURG–Toolkit

BURG–toolkit by Rudorfer et al. 2022 [8] is a set of open-source tools for Benchmarking and Understanding Robotic Grasping. It allows users to

1. create virtual scenes for generating training data and performing grasping in simulation,
2. create print-out templates to recreate virtual scenes in the physical world,
3. share the scenes with other researchers to foster comparability and reproducibility of experimental results.

The toolkit contains a library of household object models as well as models of obstacles for modelling the environment. This enables the user to benchmark not only grasping algorithms for graspable objects in the library, but also collision-avoidance in the context of grasping and motion planning. For example, an environment with a grasp target and obstacles can be created to test a motion planning algorithm, Figure 2.16.

2.6.1 Summary

Planners introduced in this section have a combination of desirable properties and also a lack of other features. IK-RRT [16], J+RRT [23] and workspace RRT [17] are all able to plan towards workspace goals but are based on RRT and may have slow exploration due to inherent properties of RRT. RBT [3] on the other hand has quick exploratory properties both in configuration and in workspace, but so far lacks any kind of adaptation for workspace goals. It would either have to rely on an existing inverse kinematics solver or use some sort of biasing like J+RRT. In the next section we will investigate how to adapt RBT to apply more generally to rigid-body manipulators and discuss possible extension for it. Testing of planners will be done in an environment created using the BURG–toolkit.

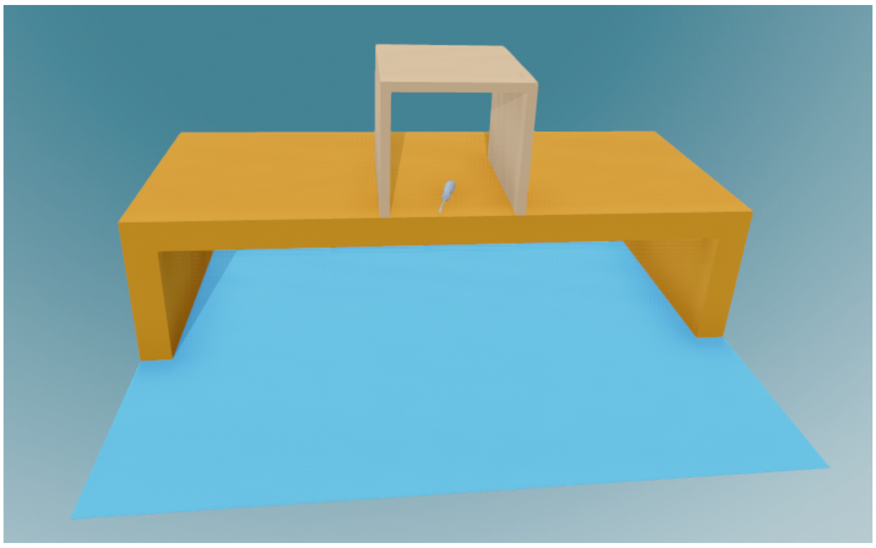


Figure 2.16: A scene with a screwdriver as grasp target and a shelf on a table as obstacles.

Chapter 3

Methods

In this section we will delve deeper into the burs of free space algorithm. In [3], we were introduced to line-segment revolute-joint burs, and in [20], the idea of a bur was adapted to rigid bodies. The question remains whether a bur can be adapted into a more general form such as a rigid-body manipulator. Lacevic et al. [3] compute burs for a revolute-joint manipulator whose forward kinematics determine world position solely by joint angles:

$$f_{\text{revolute}}(\mathbf{q}) = f_{\text{revolute}}(\theta_1, \dots, \theta_n) = \begin{bmatrix} \mathbf{p}_1(\theta_1) \\ \vdots \\ \mathbf{p}_n(\theta_1, \dots, \theta_n) \end{bmatrix}, \tag{3.1}$$

where a configuration $\mathbf{q}$ is defined as the list of joint angles: $\mathbf{q} = [\theta_1, \dots, \theta_n]$ and $f_{\text{revolute}}(\mathbf{q}) = [\mathbf{p}_1, \dots, \mathbf{p}_n]$ are the workspace positions of the segments and end-effector. To compute a rigid-body bur, on the other hand, one needs to take into account workspace translation $\mathbf{t}$ and rotation ϕ:

$$f_{\text{rigid}}(\mathbf{q}) = f_{\text{rigid}}(\mathbf{t}, \phi) = \begin{bmatrix} \mathbf{t} + \mathbf{R}(\phi)\mathbf{p}_1 \\ \vdots \\ \mathbf{t} + \mathbf{R}(\phi)\mathbf{p}_n \end{bmatrix}, \tag{3.2}$$

where a configuration $\mathbf{q}$ is defined as the world translation and rotation directly represented as vectors: $\mathbf{q} = [\mathbf{t}, \phi] = [\mathbf{t}_x, \mathbf{t}_y, \mathbf{t}_z, \phi_x, \phi_y, \phi_z]$, $\mathbf{R}(\phi)$ is the rotation matrix corresponding to ϕ and $\mathbf{p}_i$ is the position of point i on the robot geometry $\mathbf{p}_i \in \mathcal{G}$, $i = 1 \dots n$. All derivations in this chapter are configuration-space agnostic, meaning that configurations are represented as a general vector of values $\mathbf{q} = [\mathbf{q}_1, \dots, \mathbf{q}_n]$ that can correspond to rotational, translational or any other type of movement a joint can perform.

A comparison of the properties of both bur types that use the forward kinematics functions from Equations 3.1, 3.2 respectively is summarized in Table 3.1.

We can think of a rigid body as a set of points statically connected to the rigid body's origin, which itself is connected to the workspace origin through multiple translational and rotational joints, Figure 3.1.

In subsequent sections we aim to unify the framework under which burs of free space are computed: revolute-joint manipulators use projections of

Implemented property	Revolute Bur	Rigid Body Bur
Rotation	YES	YES
Translation	NO	YES
General forward kinematics	YES	NO
Kinematic chain	YES	NO
Includes geometry of robot	NO	YES

Table 3.1: Classification of the applicability of each bur algorithm.

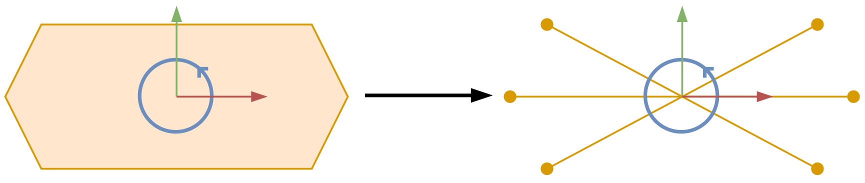


Figure 3.1: Polygon rigid body represented by a tree graph.

link positions on joint-planes to calculate burs, rigid-body burs need the rigid body's position and translation to be represented directly as configuration coordinates, and a robotic manipulator is a mixture of both.

3.1 Calculating Burs

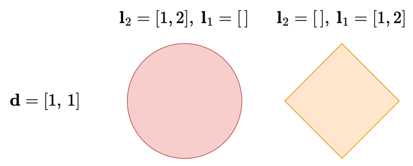
A bur's spine is computed using Equation 2.9. The equation consists of two parts:

- In the numerator there is the function $\varphi(t) = d_c - \rho_{\mathcal{R}}(\mathbf{q}, \mathbf{q} + t(\mathbf{q}_e - \mathbf{q}))$, which in turn contains $\rho_{\mathcal{R}}(\mathbf{q}_1, \mathbf{q}_2) = \max_{\mathbf{p} \in \mathcal{R}} \|f_{\mathbf{p}}(\mathbf{q}_1) - f_{\mathbf{p}}(\mathbf{q}_2)\|$. The function $\rho_{\mathcal{R}}(\mathbf{q}_1, \mathbf{q}_2)$ has already been defined generally enough for both types of burs, since $f_{\mathbf{p}}(\mathbf{q})$ is the forward kinematics of each point on the robot $\mathbf{p} \in \mathcal{R}$.
- In the denominator there is $\mathbf{r}_k^T |\mathbf{q}_e - \mathbf{q}_k|$, which later in Equation 2.15 changes to $\|\mathbf{y}_p - \mathbf{q}_p\|_2 + \mathbf{r}_\phi^T |\mathbf{y}_\phi - \mathbf{q}_\phi|$.

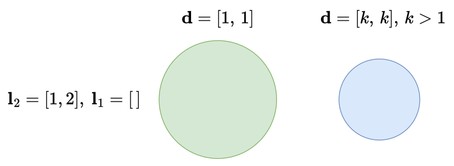
If we wish to use both types of burs then we must unify the two approaches. It is obvious that at the moment we cannot write down the collision-free bubbles from Equations 2.6, 2.12 and 2.13 in one general formula. Let us then define a more general collision-free bubble:

$$\mathcal{B}(\mathbf{q}, \mathbf{d}, \mathbf{l}_2, \mathbf{l}_1) = \\ = \left\{ \mathbf{y} \,\middle|\, \sqrt{\sum_{i \in \mathbf{l}_2} (\mathbf{d}_i |\mathbf{y}_i - \mathbf{q}_i|)^2} + \sum_{i \in \mathbf{l}_1} \mathbf{d}_i |\mathbf{y}_i - \mathbf{q}_i| < d_c,\ \mathbf{y} \in \mathcal{C}_{free} \right\} \tag{3.3}$$

where $\mathbf{d}$ is the vector signifying the lower bound of the distance when the robot changes from configuration $\mathbf{q}$ to configuration $\mathbf{y}$, $\mathbf{l}_2$ are the indices of coordinates in $\mathbf{q}$ that are measured using the L2-norm, $\mathbf{l}_1$ are the indices of


$\mathbf{l}_2 = [1,2]$, $\mathbf{l}_1 = [\,]$
$\mathbf{l}_2 = [\,]$, $\mathbf{l}_1 = [1,2]$
$\mathbf{d} = [1,\,1]$

(a) : Collision-free bubbles for norms L2 and L1 of a rigid body without rotation.


$\mathbf{d} = [1,\,1]$
$\mathbf{d} = [k,\,k]$, $k > 1$
$\mathbf{l}_2 = [1,2]$, $\mathbf{l}_1 = [\,]$

(b) : Collision-free bubbles for norm L2 with varying degrees of conservativeness of a rigid body without rotation.

Figure 3.2: Cylinders encompassing all subsequent links of the robot.

coordinates in $\mathbf{q}$ that are measured using the L1-norm. The bubbles we have already been introduced to take the following forms:

- Revolute-joint bubble, Equation 2.6: $\mathbf{l}_2 = [\,]$, $\mathbf{l}_1 = [1,\,\ldots,\,n]$, $\mathbf{d} = [r_1,\,\ldots,\,r_n]$.
- 2D rigid-body bubble, Equation 2.12: $\mathbf{l}_2 = [1,\,2]$, $\mathbf{l}_1 = [3]$, $\mathbf{d} = [1,\,1,\,r_3]$.
- 3D rigid-body bubble, Equation 2.13: $\mathbf{l}_2 = [1,\,2,\,3]$, $\mathbf{l}_1 = [4,\,5,\,6]$, $\mathbf{d} = [1,\,1,\,1,\,r_1,\,r_2,\,r_3]$.

The effect of parameters $\mathbf{l}_1$, $\mathbf{l}_2$ and distance estimates $\mathbf{d}$ is depicted in Figure 3.2. We can see that switching from L2 norm to L1 makes the bubble more conservative. Increasing the norm of vector $\mathbf{d}$ also increases the conservativeness by making the bubble more sensitive to small changes in configuration.

In the following sections we will attempt to derive a method to calculate burs that do not rely on special cases. The simplest yet at the same time most general bubble is the one which uses the 1-norm:

$$\begin{aligned}&\mathcal{B}\left(\mathbf{q}, \mathbf{d}, \mathbf{l}_2 = [\,]\,, \mathbf{l}_1 = [1, \ldots, n]\right) = \\ &= \left\{\mathbf{y} \;\middle|\; \sum_{i=1}^{n} \mathbf{d}_i |\mathbf{y}_i - \mathbf{q}_i| < d_c,\, \mathbf{y} \in \mathcal{C}_{free}\right\}\end{aligned} \tag{3.4}$$

The update that corresponds to this bubble is

$$t_{k+1} = t_k + \frac{\varphi(t_k)}{\sum_{i=1}^{n} \mathbf{d}_i(t_k)|\mathbf{q}_e^i - \mathbf{q}_k^i|}(1 - t_k), \tag{3.5}$$

where $t_k \in [0,\,1]$ is the progress parameter value after step k, starting at $t_k = 0$, $\mathbf{q}_e$ is the target endpoint, $\mathbf{q}_k$ is the configuration in step k: $\mathbf{q}_k = \mathbf{q} + t_k(\mathbf{q}_e - \mathbf{q})$ and $\mathbf{d}_i$ is the lower bound estimate of distance moved for a given change coordinate i.

3.1.1 Revolute joints

We will now see how we can generalize the computation of the bur. First, let us take a look at the vector of radii $\mathbf{r}$ encompassing all links of the robot for

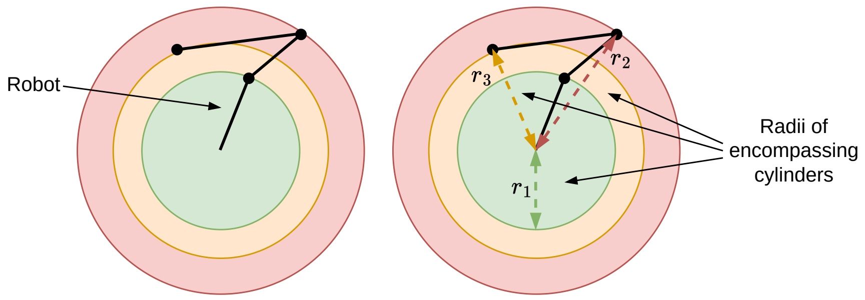


Figure 3.3: Encompassing radii from the first joint to all subsequent links.

each joint respectively. Visualizing the radius computation in Equation 2.11, we see that in Figure 3.3 the largest radius for the first joint is $\mathbf{r}_2$. We shall denote this radius as $\mathbf{r}_{1,2}$. By extension we define that $\mathbf{r}_{j,i}$ is the encompassing radius from joint j to link i.

The essential thing about the radii is that they are scalar values: we do not actually require the projected position on the plane defined by the axis of the joint. In the 2D case it is easy to see that the radius $\mathbf{r}_{j,i}$ is actually the magnitude of the derivative of the position of link i w.r.t. joint j:

$$\mathbf{r}_{j,i} = \left\| \frac{\partial \mathbf{p}_i(\mathbf{q})}{\partial \mathbf{q}_j} \right\|, \tag{3.6}$$

where $\mathbf{p}_i(\mathbf{q})$ is the i-th link's position and $\mathbf{q}_j$ is configuration $\mathbf{q}$'s j-th coordinate.

We can acquire the radius of the cylinder that encompasses the whole robot from joint j by taking the maximum of Equation 3.6 over all links i affected by joint j:

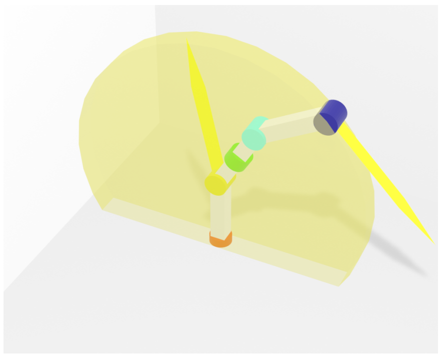
$$\mathbf{r}_j = \max_{i=j\ldots n} \left\| \frac{\partial \mathbf{p}_i(\mathbf{q})}{\partial \mathbf{q}_j} \right\| = \max_{i=j\ldots n} \left\| \mathbf{J}_{j,i} \right\|, \tag{3.7}$$

where $\mathbf{J}_{j,i}$ is the Jacobian of link $\mathbf{p}_i$ w.r.t. $\mathbf{q}_j$.

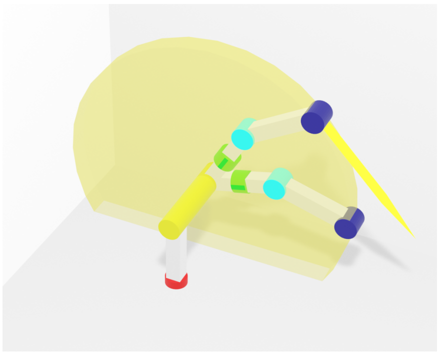
An illustration of how the Jacobian approach works in 3D is depicted in Figure 3.4. Figure 3.4a contains the vector of the Jacobian together with the projected radius in their respective world poses. In Figure 3.4b there is the rest of the robot together with the Jacobian, both offset along the axis of the joint. It is clear that in both figures the Jacobian is identical.

3.1.2 Prismatic joints

For prismatic joints there does not exist the concept of axis of rotation nor radius that encompasses every moving link of the robot. The original goal of encompassing radii was to estimate the maximum possible movement for a given change in coordinates in configuration space. The rigid-body bubble of free space, Equations 2.12 and 2.13, does indeed contain an estimate of distance for prismatic joints — the exact distance itself. As it happens, changing the coordinates in a prismatic joint directly translates to a change in workspace coordinates. As in the previous section, our task is to determine by



(a) : Rotation in the plane of the joint.



(b) : General rotation about the joint offset along the rotation axis.

Figure 3.4: Cylinders encompassing all subsequent links of the robot.

how much $\Delta\mathbf{p}$ the robot moves if we change the prismatic joint configuration coordinate q by Δq. The equation characterizing a prismatic joint is

$$\mathbf{p}_{prism}(q) = \mathbf{p}_0 + \Delta\mathbf{p} \cdot q, \tag{3.8}$$

where q is the coordinate of the prismatic joint, $\mathbf{p}_0$ is the link origin and $\Delta\mathbf{p}$ defines proportionality of q w.r.t. the translation of the link. It is obvious that an estimate of the translation of a prismatic joint is again the Jacobian:

$$\frac{\partial \mathbf{p}_{prism}(q)}{\partial q} = \mathbf{J}(q) = \Delta\mathbf{p}. \tag{3.9}$$

The distance estimate vector $\mathbf{d}$ is calculated like

$$\mathbf{d}_{j,i} = \max_{i=j\dots n} \left\| \frac{\partial \mathbf{p}_i(\mathbf{q})}{\partial \mathbf{q}_j} \right\| = \max_{i=j\dots n} \|\mathbf{J}_{j,i}\| . \tag{3.10}$$

Unlike revolute joints, however, prismatic joints affect all links equally, therefore for implementation purposes it is unnecessary to take the maxima of the partial Jacobians for each link; only the first value suffices.

3.1.3 Revolute and Prismatic Bur

We now have burs that are more general than before. They include revolute as well as prismatic joints; not only that, we can use the Jacobian the same way to calculate distance estimates for both types of joints. We can now update Table 3.1 to 3.2.

3.1.4 Added bonus for workspace goals

In the case of planning with a workspace goal, calculating and saving Jacobians becomes advantageous thanks to the fact that they can be reused when extending towards the goal. The process of biasing the end-effector towards a goal uses the pseudo-inverse of Jacobian, which itself is calculated from the Jacobian.

Implemented property	Revolute **and** Prismatic Bur	Rigid Body Bur
Rotation	YES	YES
Translation	NO$\longrightarrow$**YES**	YES
General forward kinematics	YES	NO
Kinematic chain	YES	NO
Includes geometry of robot	NO	YES

Table 3.2: Classification of the applicability of each bur algorithm.

3.1.5 Including geometry for mobile robotic manipulators

Recall that in Figure 3.1, a rigid body is represented as a tree graph. To obtain a bur for a manipulator that is composed of rigid bodies we therefore need to take into account how all these points move with each segment. We can split this task into two sub-tasks:

- recalculating encompassing radii for each segment in each configuration;
- calculating distance between configurations.

Calculating distance between configurations has been defined, but it may prove slow because Lacevic et al. [20] consider *all* points on the robot geometry $\mathbf{p} \in \mathcal{G}$. For geometries with a higher number of triangles, the cost of computation increases significantly since the complexity of $\rho_{\mathcal{R}}$ is linearly proportional to the number of points. Results from Lacevic et al. [20] indeed show that for the most complex geometry tested, planning times are comparable to other planners they benchmarked. Not only that, RbtConnect is only marginally faster than even the basic RRT [11] in that scenario.

The only missing feature in the kinematic-chain bur from Table 3.2 is geometry. Let us derive the Jacobian for the points on the geometry of the robot when linked into a chain. The Jacobian takes the form:

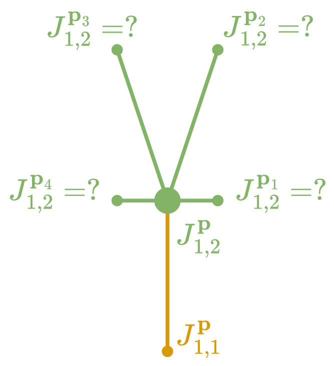
$$\mathbf{J}_{j,i} = \begin{bmatrix} \mathbf{J}^{\mathbf{p}}_{j,i} \\ \mathbf{J}^{\phi}_{j,i} \end{bmatrix} \in \mathbb{R}^{6\times n}, \ \mathbf{J}^{\mathbf{p}}_{j,i}, \mathbf{J}^{\phi}_{j,i} \in \mathbb{R}^{3\times n}. \tag{3.11}$$

Points from each segment belong to its local frame $\mathcal{F}$. A set of points belonging to local frame $\mathcal{F}$ of segment i is depicted in Figure 3.5. The positional Jacobian $\mathbf{J}^{\mathbf{p}}_{j,i}$ only approximates the movement of the local origin of the segment i. To approximate movement of the other points k on the segment, we first write down their equation:

$$\mathbf{p}^{\mathcal{W}}_k = \mathbf{p}^{\mathcal{W}}_k(\theta_1, \dots, \theta_n) + \mathbf{R}_{\mathcal{F}\longrightarrow\mathcal{W}}(\theta_1, \dots, \theta_n)\mathbf{p}^{\mathcal{F}}_k, \tag{3.12}$$

where $\mathbf{R}_{\mathcal{F}\longrightarrow\mathcal{W}}(\theta_1, \dots, \theta_n)$ rotates points from the local frame $\mathcal{F}$ to the workspace frame $\mathcal{W}$. It is clear now that taking the derivative of the above expression gives us

$$\begin{aligned} \frac{\partial \mathbf{p}^{\mathcal{W}}_k}{\partial \theta_j} &= \frac{\mathbf{p}^{\mathcal{W}}_k(\theta_1, \dots, \theta_n)}{\partial \theta_j} + \left(\mathbf{p}^{\mathcal{F}}_k\right)^T \frac{\partial \mathbf{R}_{\mathcal{F}\longrightarrow\mathcal{W}}(\theta_1, \dots, \theta_n)}{\partial \theta_j} = \\ &= \mathbf{J}^{\mathbf{p}}_{j,i} + \left(\mathbf{p}^{\mathcal{F}}_k\right)^T \mathbf{J}^{\phi}_{j,i} \end{aligned} \tag{3.13}$$



$J^{\mathbf{p}_3}_{1,2} =?$
$J^{\mathbf{p}_2}_{1,2} =?$
$J^{\mathbf{p}_4}_{1,2} =?$
$J^{\mathbf{p}_1}_{1,2} =?$
$J^{\mathbf{p}}_{1,2}$
$J^{\mathbf{p}}_{1,1}$

Figure 3.5: Geometry of a chain-link robot and the points whose movement estimations are calculated.

where $\mathbf{J}^{\mathbf{p}}_{j,i}$ is the positional Jacobian as we have already discussed, $\left(\mathbf{p}^{\mathcal{F}}_k\right)^T \mathbf{J}^{\phi}_{j,i}$ is the positional Jacobian of the nested points.

Simplifying derivation for the 2D case again, considering a case like in Figure 3.5, we denote points belonging to the geometry of a segment with zero offset in the global frame as

$$\mathbf{p}^{\mathcal{W}} = \mathbf{R}_{\mathcal{F}\longrightarrow\mathcal{W}}(\theta)\mathbf{p}^{\mathcal{F}} \tag{3.14}$$

Taking the above expression and linearizing it at point θ^k:

$$\frac{\partial \mathbf{R}(\theta)\mathbf{p}}{\partial \theta_j}|_{\theta^k} = \mathbf{p}^T\frac{\partial \mathbf{R}(\theta)}{\partial \theta_j}|_{\theta^k} = \mathbf{p}^T\frac{\partial \mathbf{R}(\theta^k)}{\partial \theta_j} \cdot \Delta_\theta. \tag{3.15}$$

Analogically for 3D, we can consider each axis of rotation independently of each other and linearize at point θ_k:

$$\begin{aligned}
&\frac{\partial \mathbf{R}(\theta)\mathbf{p}}{\partial \theta_j}|_{\theta^k} = \frac{\partial \mathbf{R}(\theta)\mathbf{p}}{\partial \phi}|_{\theta^k} = \frac{\partial \mathbf{R}(\theta)\mathbf{p}}{\partial \phi_x, \phi_y, \phi_z}|_{\theta^k} = \\
=&\mathbf{p}^T\frac{\partial \mathbf{R}(\theta)}{\partial \phi_x}|_{\theta^k} + \mathbf{p}^T\frac{\partial \mathbf{R}(\theta)}{\partial \phi_y}|_{\theta^k} + \mathbf{p}^T\frac{\partial \mathbf{R}(\theta)}{\partial \phi_z}|_{\theta^k} = \\
=&\mathbf{p}^T\frac{\partial \mathbf{R}(\theta^k)}{\partial \phi_x} \cdot \phi_x + \mathbf{p}^T\frac{\partial \mathbf{R}(\theta^k)}{\partial \phi_y} \cdot \phi_y + \mathbf{p}^T\frac{\partial \mathbf{R}(\theta^k)}{\partial \phi_z} \cdot \phi_z = \\
=&\mathbf{p}^T\left(\frac{\partial \mathbf{R}(\theta^k)}{\partial \phi_x} \cdot \phi_x + \frac{\partial \mathbf{R}(\theta^k)}{\partial \phi_y} \cdot \phi_y + \frac{\partial \mathbf{R}(\theta^k)}{\partial \phi_z} \cdot \phi_z\right),
\end{aligned} \tag{3.16}$$

where ϕ_x, ϕ_y, ϕ_z are the angles from the rotational part of the Jacobian $\mathbf{J}^\phi$. Now, for every point $\mathbf{p}_k$ in segment i we want to get an estimate of the distance it can move. We have proven in Sections 3.1.1 and 3.1.2 that taking the norm of the Jacobian of a point on the kinematic chain gives us an estimate of the distance travelled. Thus, all we have to do is calculate the norm of the Jacobian of point $\mathbf{p}_k$ with respect to joint j for which we want to calculate the minimum encompassing radius or distance estimate. Taking the norm of Equation 3.13 and substituting the rotational Jacobian from Equation 3.16:

$$\left\| \frac{\partial \mathbf{p}_i^{\mathcal{W}}(\theta^k)}{\partial \theta_j} \right\| = \left\| \mathbf{J}_{j,i}^{\mathbf{p}}(\theta^k) + \left(\mathbf{p}_i^{\mathcal{F}}\right)^T \mathbf{J}_{j,i}^{\phi}(\theta^k) \right\| =$$
$$= \left\| \mathbf{J}_{j,i}^{\mathbf{p}}(\theta^k) + \left(\mathbf{p}_i^{\mathcal{F}}\right)^T \left(\frac{\partial \mathbf{R}(\theta^k)}{\partial \phi_x}\phi_x + \frac{\partial \mathbf{R}(\theta^k)}{\partial \phi_y}\phi_y + \frac{\partial \mathbf{R}(\theta^k)}{\partial \phi_z}\phi_z \right) \right\|, \quad (3.17)$$

where θ^k is the angle at which the rotation of point $\mathbf{p}_i$ was linearized, $\mathbf{J}_{j,i}^{\phi}(\theta^k) = [\phi_x, \phi_y, \phi_z]$ are the xyz components of the rotational Jacobian. Same as in Equations 3.7 and 3.10, for every joint j and point k on segment geometry i, we would have to iterate over all points on every segment to get the maximal translation:

$$\mathbf{d}_j = \max_{i=j\ldots n} \left(\max_{k=1\ldots m} \left\| \frac{\partial \mathbf{p}_{i,k}^{\mathcal{W}}}{\partial \theta_j} \right\| \right) \quad (3.18)$$

Iterating over all points can be slow and implementation is prone to error. Instead, we can separate the translation and rotation parts of the Jacobian from Equation 3.17:

$$\left\| \frac{\partial \mathbf{p}_i^{\mathcal{W}}}{\partial \theta_j} \right\| = \left\| \mathbf{J}_{j,i}^{\mathbf{p}} + \left(\mathbf{p}_i^{\mathcal{F}}\right)^T \mathbf{J}_{j,i}^{\phi} \right\| \leq \left\| \mathbf{J}_{j,i}^{\mathbf{p}} \right\| + \left\| \left(\mathbf{p}_i^{\mathcal{F}}\right)^T \mathbf{J}_{j,i}^{\phi} \right\| \leq \left\| \mathbf{J}_{j,i}^{\mathbf{p}} \right\| + \left\| \begin{bmatrix} \mathbf{r}_x^i \phi_x^j \\ \mathbf{r}_y^i \phi_y^j \\ \mathbf{r}_z^i \phi_z^j \end{bmatrix} \right\|, \quad (3.19)$$

where $\mathbf{r}$ is the vector of encompassing radii of segment i's geometry, as has been illustrated in Figure 2.13 for rigid bodies, $\mathbf{J}_{j,i}^{\phi} = [\phi_x^j, \phi_y^j, \phi_z^j]$ are the elements of the rotational Jacobian for joint j. The above equation is easier to compute, and it incorporates the radii that are readily available from Lacevic et al. [20]. In addition, the equation above can be computed only once per segment, as opposed to Equation 3.18, which has to be calculated for every point in every segment of the robot's geometry.

Combining the maximum function from Equation 3.18 and the upper approximation of its components from Equation 3.19, we get the total upper approximation of the translation of all points connected to joint j:

$$\mathbf{d}_j = \max_{i=j\ldots n} \left(\left\| \mathbf{J}_{j,i}^{\mathbf{p}} \right\| + \left\| \begin{bmatrix} \mathbf{r}_x^i \phi_x^j \\ \mathbf{r}_y^i \phi_y^j \\ \mathbf{r}_z^i \phi_z^j \end{bmatrix} \right\| \right). \quad (3.20)$$

This final equation is of quadratic complexity, same as the original method to calculate radii from Equation 2.11, but now it also requires the Jacobian to be calculated, which adds some complexity because the Jacobian requires matrix multiplication to be calculated.

We now have burs that are once more general than before. They include revolute as well as prismatic joints and take into account the movement of points on the geometry of each segment of the robot. We can now update Table 3.2 to 3.3. The bur can now take into account forward kinematics of rigid bodies as well as kinematic chains.

Implemented property	Rigid Body Chain Bur	Rigid Body Bur
Rotation	YES	YES
Translation	**YES**	YES
General forward kinematics	YES	NO
Kinematic Chain	YES	NO
Includes geometry of robot	NO⟶**YES**	YES

Table 3.3: Classification of the applicability of each bur algorithm.

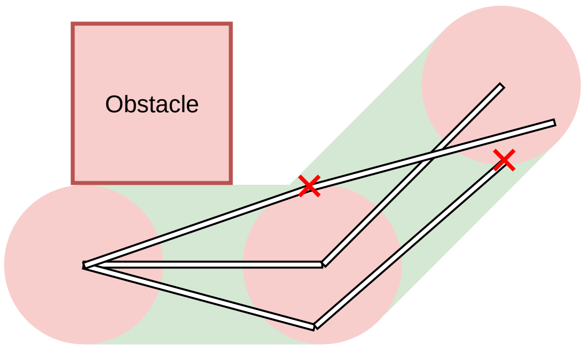


Figure 3.6: Bur distances used for calculation. Red: areas in which link endpoints can move — equivalent of the complete bubble $\mathcal{CB}$ but in workspace.

3.2 Extended RBT

There is a special case where even burs can be considered to be too restrictive. Recall in RGBT [22] that obstacles are at each distance query assumed to be infinite planes because the obstacles are convex. What happens if obstacles are non-convex? We will now use the same thought process to approximate the general case. In Figure 3.6 we can see the collision-free area that is generated by a single obstacle and how it restricts the movement of the whole robot. The red circles show areas where the endpoints of each segment are able to move to in the case of a regular bur.

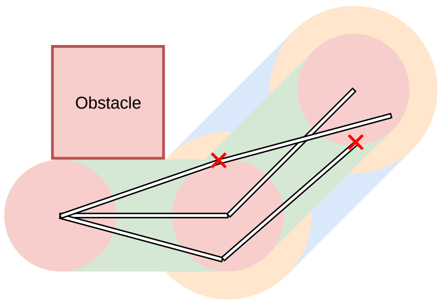
The regular bur assumes that only the single closest obstacle determines the whole collision-free space:

$$\mathcal{W}_{\text{free}} = f(\mathbf{q}, d_c), \tag{3.21}$$

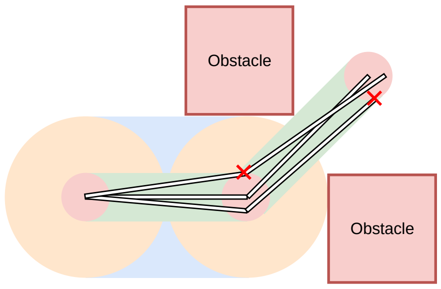
where $\mathcal{W}_{\text{free}}$ is the collision-free workspace, $\mathbf{q}$ is the configuration for which $\mathcal{W}_{\text{free}}$ is calculated and d_c is the distance to the closest obstacle. In reality, the collision-free space can more generally be defined by the closest distance between each segment and the environment:

$$\mathcal{W}_{\text{free}} = f(\mathbf{q}, \mathbf{d}_c), \tag{3.22}$$

where $\mathbf{d}_c$ is the vector of closest distances for each segment of the robot. In Figure 3.7 we can see two cases of regular bur extension superimposed on the total collision-free space as defined by the robot segments and surrounding environment $\mathcal{O}$.

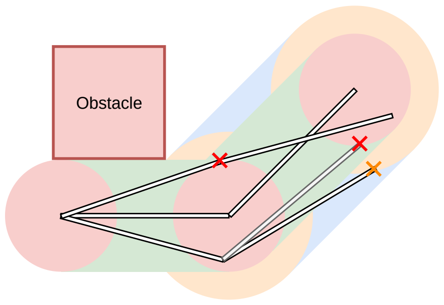


(a) : Lower links have *less* free distance than higher links.

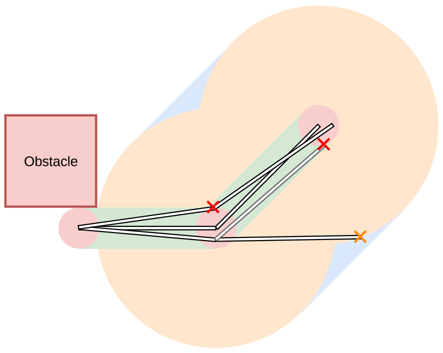


(b) : Lower links have *more* free distance than higher links.

Figure 3.7: General collision-free area.



(a) : Slightly extended bur for second segment.



(b) : Highly extended bur for second segment.

Figure 3.8: Configurations where expansion of subsequent links beyond the boundary of a bur is possible.

In Figure 3.7a, the second segment has a comparatively larger collision-free area than the first segment. In such cases where later links have more collision-free space, we can extend movement by freezing the lower joints and continuing growing a bur, but only for later links of the robot using the next smallest distances each time. Regular forward kinematics $f_{\mathcal{C}\to\mathcal{W}}$ and the vector of smallest distances to obstacles for each segment $\mathbf{d}_c$ is all we need to continue growing the now pseudo-bur.

In Figure 3.8a there is a less generous, and in Figure 3.8b a more generous case, where this special case of extended bur helps. It must be noted that this extended bur helps only when the lower links of the robot are limited — in Figure 3.7b it is of no use that the base is free but the end-effector restrained. The process for calculating an extended bur is described in Algorithm 9. It first creates a regular bur, then it starts on the lowest joint that still has a non-trivial remaining distance budget and extends the bur further, repeating this over all subsequent joints that have, after each additional step, some non-trivial distance left to move.

The extended bur does not in fact break star-convexity because the extension it performs on a regular bur is the product of a Minkowski sum of

Algorithm 9 ExtendedBur($\mathbf{q}$, $\mathbf{Q}_e$, $\mathbf{d}_c$, $\mathbf{i}_{\min}$), Figure 3.8 and 3.9

```
$i_d = \mathbf{i}_{\min}(0)$                                   ▷ $\mathbf{i}_{\min} = \text{argsort}(\mathbf{d}_c)$
$\mathbf{B} \leftarrow \text{Endpoints}(\mathbf{q}, \mathbf{Q}_e, \mathbf{d}_c(i_d))$          ▷ Generate bur endpoints.
for $j = 1$ to Length($\mathbf{d}_c$) do                       ▷ Next smallest distance
    if $\mathbf{i}_{\min}(j) < i_d$ then
        continue
    end if
    $i_d \leftarrow \mathbf{i}_{\min}(j)$
    for $k = 0$ to Length($\mathbf{B}$) do
        $d_\Delta \leftarrow \mathbf{d}_c(i_d) - \rho_{\mathcal{R}}(\mathbf{q}, \mathbf{B}(k))$         ▷ Distance left for segment $i$
        if $d_\Delta < d_{crit}$ then                          ▷ To save computation
            continue
        end if
        $\mathbf{q}_M \leftarrow \text{Mask}(\mathbf{Q}_e(k), i_d$          ▷ Zeroes out elements with index $< i$
        $\mathbf{B}(k) \leftarrow \text{Endpoints}(\mathbf{B}(k), \mathbf{q}_M, d_\Delta)$
    end for
end for
return B
```

a convex set points with the original bur. The 2D case is illustrated in Figure 3.9. It is formed by splitting the original bur horizontally, then stretching each half in the direction of the half-plane in which it is located. This is equivalent to a Minkowski sum with a line segment oriented into the vertical half-plane in which the corresponding half of the bur is located.

Caveat

In cases like the one in Figure 3.7b where lower links have more collision-free distance remaining than later links we cannot simply freeze later links and move with the base of the robot because moving earlier links affects all subsequent links. Keeping those inside their respective collision-free areas is not straightforward and would require further inquiry. Moreover, the extended bur is the opposite case which we would like to have if we want the end-effector of a robot to approach and grasp objects. Having the end-effector close to an obstacle highly restricts the robot's movement as a whole.

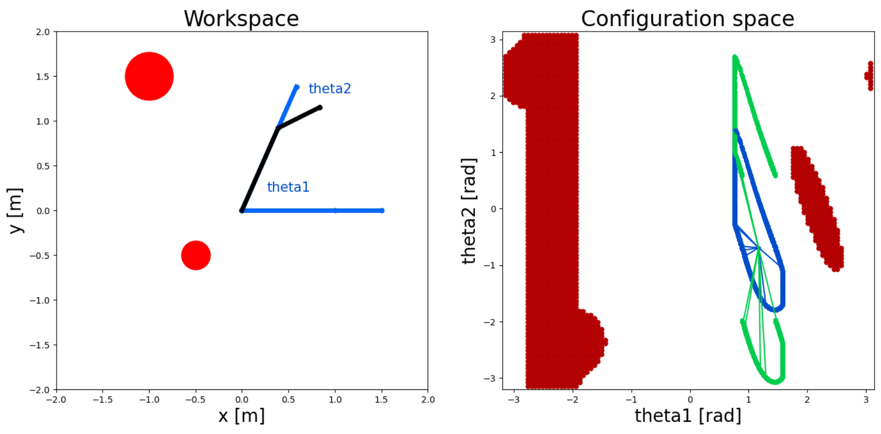


Figure 3.9: Extended bur — it is split in the middle and "stretched".

Chapter 4

Results

In this chapter, we discuss our findings about the presented algorithms and their runtime results for scenarios of varying difficulty. First, we will examine available existing implementations of RBT. In Section 3 we derived how to calculate a bur for a rigid body manipulator, combining the algorithms from [3] and [20]. The bur calculation for revolute-joint, kinematic-chain robotic manipulators from [3] unfortunately causes collisions at some endpoints because it ignores robot geometry and allows some points of the geometry to move beyond the complete bubble $\mathcal{CB}$. For that reason we implemented bur calculation with a single collision check at the end of each endpoint calculation. We have implemented the calculation of burs using Jacobians according to Equations 3.7 and 3.10 as well as the original version according to Equation 2.11. The bur that needs no collision checks and incorporates geometry uses Equation 3.20 to calculate distance estimates.

We have four types of scenarios as depicted in Figure 4.2. Each scenario comes in five difficulties, however, we will only be testing the planners on the scenarios where the greatest differences in performance among planners are visible, namely scenarios of medium difficulty. We test three motion planners: J+RRT, IK-RRT and J+RBT. For each scenario there were generated 200 target grasps using the Jogramop framework [24]. The starting configuration that is common to all scenarios is visualized in Figure 4.1a. A sample goal configuration along with the end-effector trajectory is depicted in Figure 4.1b; the goal configuration is determined by the end-effector being close enough in translation and rotation terms to a goal grasping pose. The distance score between a grasp pose g and end effector pose e is calculated as follows:

$$\begin{aligned}\text{Distance}(e, g) &= \text{Translation}(e, g)\ [\text{mm}] + \text{Rotation}(e, g)\ [\text{deg}] = \\ &= 1000\, \|\mathbf{t}_e - \mathbf{t}_g\| + \text{RadToDeg}\left(\text{acos}\left(\frac{\text{Tr}(\mathbf{R}_e^T \mathbf{R}_g) - 1}{2}\right)\right),\end{aligned} \tag{4.1}$$

where $e = [\mathbf{R}_e, \mathbf{t}_e]$ is the end-effector pose, $g = [\mathbf{R}_g, \mathbf{t}_g]$ is the grasp pose, Tr is the matrix trace function. We usually stop at Distance $= 50$ as it is within a few centimeters and a few degrees within the target grasp pose.

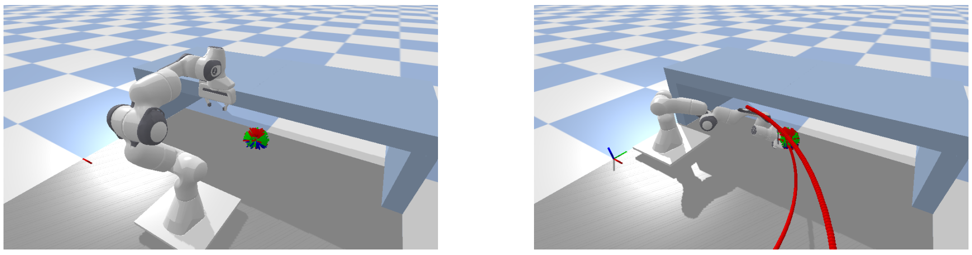
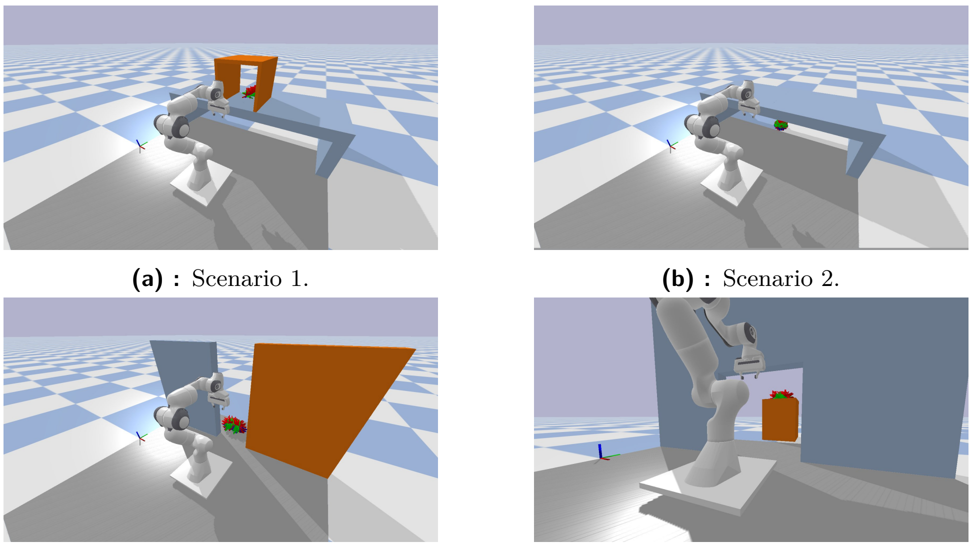

(a) : Starting configuration. **(b) :** Goal configuration of scenario 2.

Figure 4.1: Start and goal configurations of scenario 2 with a solution trajectory visualization.


(a) : Scenario 1. **(b) :** Scenario 2.

(c) : Scenario 3. **(d) :** Scenario 4.

Figure 4.2: Four tested scenarios.

4.1 RBT implementation

We use the Orocos Kinematics and Dynamics Library (KDL) [25] for loading Unified Robot Description Format (URDF) files of robots, and we use the library to perform forward kinematics, inverse kinematics and Jacobian calculations. For collision-checking and distance-checking we use the Proximity Query Package (PQP) [26]. For nearest neighbours we use the FLANN library with a kd-tree approach [27].

The algorithm that we have implemented to compare to J+RRT, Algorithm 4, and to IK-RRT 3 is described in Algorithm 10. It is analogical to J+RRT, only it replaces the regular RRT part with the RBT part. It uses bur-based extensions where possible and only reverts back to basic RRT steps when the closest obstacle is closer than a given threshold d_{crit}.

Algorithm 10 J+RBT($\mathbf{q}_{\text{init}}, \mathbf{p}_{\text{obj}}, G$), $\mathbf{q}_{\text{init}}$ = start configuration, $\mathbf{p}_{\text{obj}}$ = object pose, G = set of grasp poses

```
$\mathcal{T}$.AddConfiguration($\mathbf{q}_{\text{init}}$)
for $k = 1$ to $k_{\max}$ do
    $\mathbf{Q}_e \leftarrow \{\}$
    for $i = 1$ to $N$ do
        $\mathbf{q}_{e_i} \leftarrow$ RandomConfig()
        $\mathbf{Q}_e$.Add($\mathbf{q}_{e_i}$)
    end for
    $\mathbf{q}_{\text{near}} \leftarrow$ Nearest($\mathbf{q}_{e_1}, \mathcal{T}$)
    for $i = 1$ to $N$ do                                  ▷ Offset target configurations for bur
        $\mathbf{Q}_e(i) \leftarrow \delta \cdot$ Normalize($\mathbf{Q}_e(i) - \mathbf{q}_{\text{near}}$)          ▷ $\delta \approx$ joint range
    end for
    if $d_c(\mathbf{q}_{\text{near}}) < d_{\text{crit}}$ then                  ▷ $d_c$ = distance to closest obstacle
        if Extend($\mathcal{T}, \mathbf{q}_{\text{near}}, \emptyset, \mathbf{q}_{\text{new}}$) = Trapped then             ▷ Algorithm 2
            continue
        end if
    else
        $\mathcal{T}$.Add(Bur($\mathbf{q}_{\text{near}}, \mathbf{Q}_e, d_c(\mathbf{q}_{\text{near}})$))            ▷ Bur from Algorithm 6
    end if
    if Rand() $< p_{\text{steer}}$ then
        Solution $\leftarrow$ ExtendToGoal($\mathcal{T}$, $\mathbf{p}_{\text{obj}}$, $G$)              ▷ Algorithm 5
        if Solution $\neq$ NULL then
            return Solution
        end if
    end if
end for
return Failure
```

4.2 Hyperparameters

Lacevic et al. [3] [21] [20] [22] define some of the hyperparameters but not all of them. During testing of RBT we found out that to truly replicate RBT we needed the source code because Lacevic et al. omitted some important parameters. The missing parameter was the number of collision checks per unit of distance travelled in the workspace p_Δ. For reference, Matlab's built-in manipulator RRT planner performs one collision check every 1 cm [28]. Lacevic et al. use collision checking not in terms of collision checks for every unit of distance moved $\frac{1}{p_\Delta}$ $[m^{-1}]$ but in terms of collision checks per unit of configuration space, which in their case translates to collision checks per radian $\frac{1}{q_\Delta}$ [rad^{-1}]. This causes the distribution of their collision checks to be inconsistent in the workspace, however, the frequency of collision checking is high enough so as to avoid phasing through obstacles. Converting from configuration units to workspace units this comes out as approximately one collision check every 5 mm for the Franka Emika Panda manipulator from

Figure 2.1, double that of which Matlab uses. Hyperparameters that we used during testing are listed in Table 4.1.

Hyperparameter	Value
ε	0.1
δ	2π
d_{crit}	0.03
Collision resolution	0.005
Bur spines	7

Table 4.1: Table of hyperparameters.

4.3 Bur endpoint calculation

We have implemented three different methods to calculate burs. They are characterized in Table 4.2. The first bur is a true bur in the sense that it reaches its endpoints without colliding with the environment.

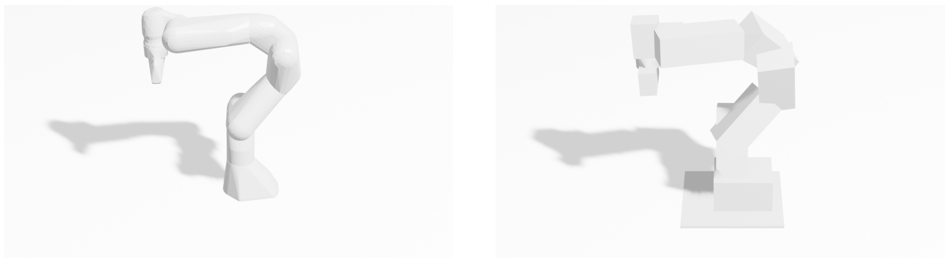
We have found that for rigid-body robotic manipulators it is unnecessary to calculate the distance between the configurations as the highest distance moved for each point on the robot's geometry as defined in Equation 2.7. Simply calculating the distances moved by each link is sufficient to fulfill the function of $\rho_{\mathcal{R}}$ from Equation 2.9. In total, we have derived three different bur spine endpoint calculations. In Table 4.2 we see the distinct properties of each endpoint calculation: the first one is the only one that uses the robot geometry, the second and third one are similar in that they use the kinematic chain representation of the robot as opposed to a series of rigid bodies. The second and third one also need a collision check at the end because they do not take into account the robot geometry.

Robot	Distance estimate	Collision check
Rigid body	Equation 3.20	No
Kinematic chain	Equations 3.7, 3.10	Yes
Kinematic chain	Equation 2.11	Yes

Table 4.2: Endpoint calculation types

4.3.1 Planning time

We shall now compare the endpoint calculations from Table 4.2 how they fare against each other. The first two are the ones we have derived in the last chapter, and the third one is the original method for bur calculation according to [3], only with an additional collision-check at the end of each spine. Collision checking and distance checking speeds depend on the number of vertices in the robot's mesh, therefore we are measuring how the computation of endpoints depends on the number of vertices. We have two meshes: a simplified, box-like mesh and the original collision mesh. The full mesh



(a) : Original robot mesh. 10^4 vertices. **(b) :** Simplified robot mesh. 10^2 vertices.

Figure 4.3: Original and simplified robot meshes.

contains on the order of 10^4 triangles with details roughly on par with the model in Figure 4.2. The simplified collision mesh is depicted in Figure 4.3b and the original full-resolution collision mesh is depicted in Figure 4.3a. We also measure the speed of calculation of each type of spine from Table 4.2.

In the ideal case, a bur should cover a significantly larger area of configuration space to offset the longer time of computation. In reality, though, the computation of a bur depends on a few factors: the distance to closest obstacle calculation, the encompassing radii calculation and the specific implementation overall. The calculation of the distance to the closest obstacle strongly depends on the number of vertices in the 3D models of the robot and the environment. Fortunately, due to the sparser nature of the bur tree it sometimes happens that we can reuse the distance check, thus saving computation time.

In Figure 4.4 we plot the time needed to compute each type of bur depending on how often the distance calculation for a given configuration is reused. On the same graph we plot out the time needed to perform a number RRT extensions equivalent to the number of spines of the bur, which Lacevic et al. state as `num spines` $= 7$ being the optimal number. We have observed that for scenarios of difficulty 4, the distance calculation is performed only 60% of the time, i.e., the distance calculation is skipped 40% of the time. We can see that distance-checking is greatly slowed down with increasing complexity of the mesh. A slow-down of 3-4 times is to be expected with a detailed mesh, as is the case in the original Franka Emika Panda collision mesh.

The times corresponding to the three types of burs from Table 4.2 are plotted out in Figure 4.4. As could somewhat be expected, the specialized projections from Equation 2.11 that give us the encompassing radii are faster than the other two calculations, although only marginally. This, however, comes at the cost of generality for prismatic joints and robot geometry.

The projections from Equation 2.11 that are necessary to compute the original bur do not natively support rigid bodies or prismatic joints. To handle robot geometry we perform a collision-check at the end of each bur endpoint, and to handle prismatic joints we add a constant estimate of p_Δ per unit of joint coordinate moved for prismatic joints. Furthermore, only bur endpoints that do not collide are kept. Algorithm 11 describes the overarching computation framework of the burs from Table 4.2.

Algorithm 11 Bur($\mathbf{q}_{\text{near}}$, $\mathbf{Q}_e$, d_c, BurType), $\mathbf{Q}_e$ = target endpoints, d_c = distance to closest obstacle, BurType $\in \{\text{JacFull}, \text{JacPos}, \text{Proj}\}$

```
Endpoints ← {}
for i=1 to Length($\mathbf{Q}_e$) do
    for k=1 to 5 do
        $t_k \leftarrow t_{k+1}(t_k, \mathbf{Q}_e(i), \mathbf{q}_k)$                ▷ Equation 3.5
        $\mathbf{q}_k \leftarrow \mathbf{q}_{\text{near}} + t_k \cdot (\mathbf{Q}_e(i) - \mathbf{q}_{\text{near}})$      ▷ Radii $\mathbf{r}_k$ are calculated in $\mathbf{q}_k$
    end for
    if BurType ≠ JacFull then
        if IsColliding($\mathbf{q}_k$) then
            continue
        end if
    end if
    Endpoints.Add($\mathbf{q}_k$)                ▷ For kinematic chain burs in Table 4.2
end for
return {$\mathbf{q}_{\text{near}}$, Endpoints}
```

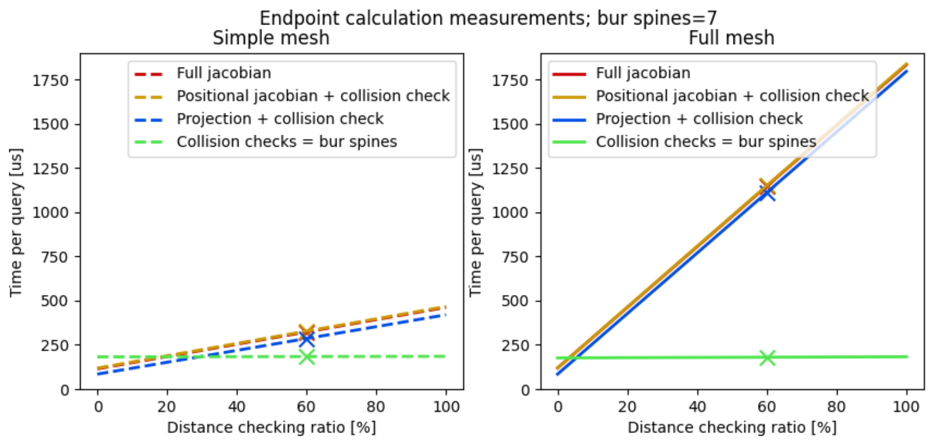


Figure 4.4: Times to calculate endpoints based on the method of calculation.

4.4 Testing on scenarios

Based on the previous results we have two candidates of burs to test: the original, projection-based bur with a collision-check at the end and the full bur calculation that takes into account robot geometry but does not require a collision-check at the end of each endpoint. We have run J+RBT with both types of burs for all scenarios, ten times each, giving us a total of 200 runs across all scenarios. In Figure 4.5 we see that the full bur solves the scenarios more slowly than the projection-based bur with an additional collision check at the end. From now on, when referring to J+RBT, we will be using the one with the projection-based bur that uses collision checks at the end — the third one in Table 4.2.

The results of planning times of planners J+RRT, Algorithm 4, IK-RRT, Algorithm 3 and J+RBT, Algorithm 10, for medium difficulties of each

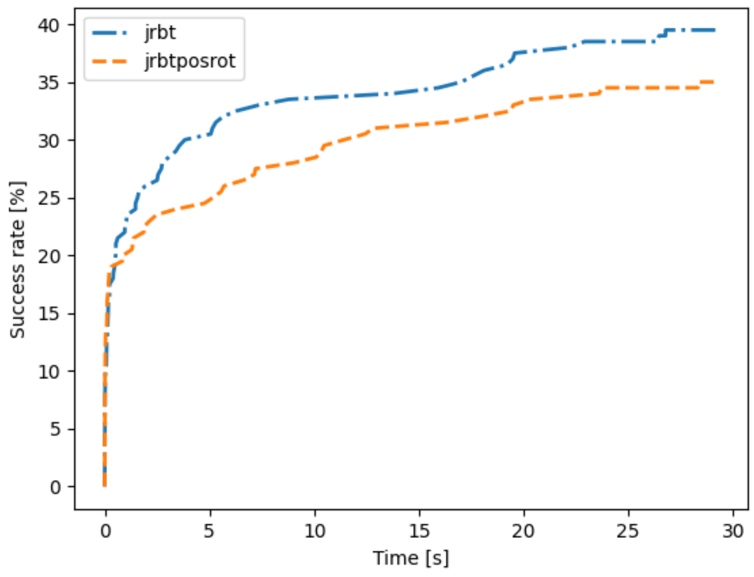


Figure 4.5: Comparison of J+RBT planners with original extended vs full bur.

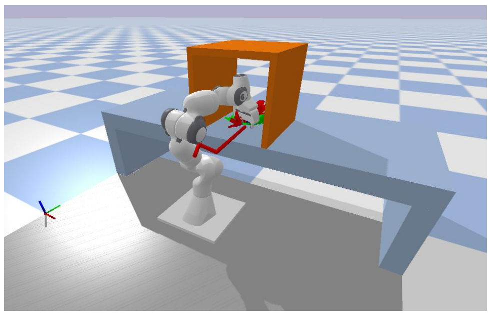


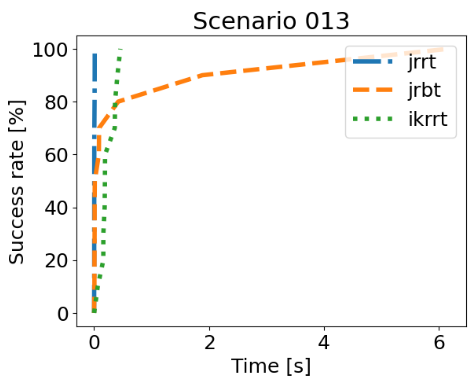
Figure 4.6: Trajectory of J+RBT in scenario 13.

scenario are depicted in Figures 4.7a, 4.7b, 4.8a and 4.8b. An example of a trajectory generated by J+RBT in scenario 13 is in Figure 4.6.

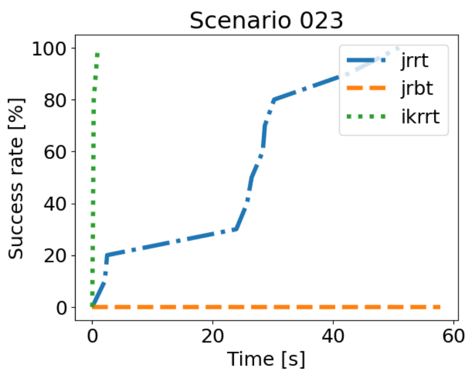
We are using simplified models of the robot with a reduced number of vertices to speed up planning times. The results show that J+RBT is significantly slower than other planners. Let us investigate why that might be the case.

4.4.1 Tree growth

The biggest change in the algorithm comes from the concept of a bur of free space. Ideally, these burs are significantly larger than an equivalent number of RRT extensions. Not only that, the number of nodes generated by RBT should be significantly larger than the number of nodes generated by RRT in the same amount of time. In Lacevic et al. [3] the authors provide data about the nodes per second speed of their implementations of RRT and RBT.

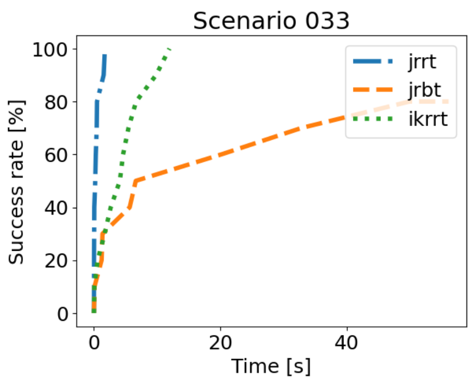


(a) : Scenario 13 planning time results.

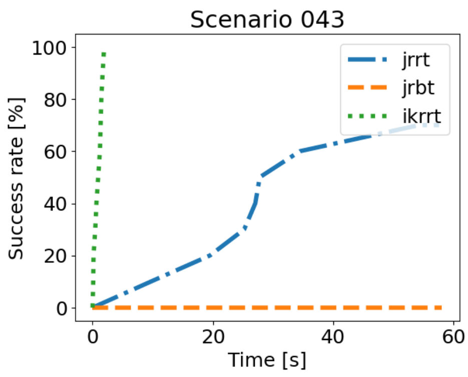


(b) : Scenario 23 planning time results.

Figure 4.7: Planning times for planners J+RRT, Algorithm 4, IK-RRT, Algorithm 3 and J+RBT, Algorithm 10, for scenarios 1 and 2.



(a) : Scenario 33 planning time results.



(b) : Scenario 43 planning time results.

Figure 4.8: Planning times for planners J+RRT, Algorithm 4, IK-RRT, Algorithm 3 and J+RBT, Algorithm 10, for scenarios 3 and 4.

In easy scenarios, their RRT produces about as many collision-free nodes per second as their RBT. In their hard scenario, RRT produces on the order of half the number of collision-free nodes that RBT does. Comparing this to our version of J+RRT and J+RBT: on our scenarios the ratio is comparable to the "hard" scenarios from [3]. J+RBT produces about 50% more collision-free nodes than RRT does. In Table 4.3 we can see the node generation rates of each algorithm respectively. What is important here is the ratios between the rate of generation of RRT nodes vs RBT nodes, not necessarily the absolute amount. The ratio between RBT and RRT nodes is consistent between our and the original methods. The inconsistency between the performance of our J+RBT planner and the original RBT is strange. The ratios of the growth of the nodes between RBT vs RRT and J+RBT vs J+RRT are comparable, yet compared to the J+RRT our algorithm is significantly slower. The algorithm we have implemented is exactly as was described in [3], yet it runs differently relative to RRT. Let us take a look at the source code to see what is different between the implementations.

	RRT (nodes/sec)	RBT (nodes/sec)
Original (easy)	4200 [3]	4300 [3]
Original (hard)	2200 [3]	3800 [3]
Ours (all scenarios)	5200	7600

Table 4.3: Comparison of node generation rates between the original and our versions of RRT and RBT algorithms. Rates vary depending on the CPU used.

4.4.2 Implementation

The RBT algorithm consists of a few main parts: distance checking, forward kinematics, collision checking, bur calculation. We will dissect each part bit by bit to see where the implementations differ. We will be referring to the source code as provided by the authors in [29].

Distance checking

We perform distance checking using PQP; Lacevic et al. perform distance checking using FCL, but fundamentally these two libraries perform almost identical computations for meshes. Lacevic et al. reuse distance calculations for given configurations, we do the same.

However, they have hardcoded that for the obstacle called `table` and for a robot that has the property called `with_table` and for its first two segments it calculates the closest distance to be infinity. We modify the distance calculation differently in that we let the user specify as a parameter the segment from which they want to include the ground-level to be included in closest obstacle calculations. For example, we can set the ground level to be $z = 0$ and only consider segments with index higher than i in the distance calculations relating to the ground plane. The way Lacevic et al. hardcoded the ground level with specific names and segment indices is useless for anyone but the authors.

Forward kinematics

Both we and Lacevic et al. use the KDL library for forward kinematics and URDF parsing. RRT performs significantly more forward kinematics computations than RBT, therefore any change to forward kinematics computations disproportionately affects RRT more. We perform forward kinematics for a kinematic chain once when creating a configuration. According to the source code of Lacevic et al. they perform forward kinematics separately for each segment, i.e., they first iterate towards the first segment, then iterate again from the beginning towards the second segment and so on until the last segment. This would slow down RRT more than it would slow down RBT. Their forward kinematics end up having a complexity $O(\frac{1}{2}N(N+1))$ for N segments. For the Franka Emika Panda robot this ends up being 6x slower even with compiler optimizations turned on.

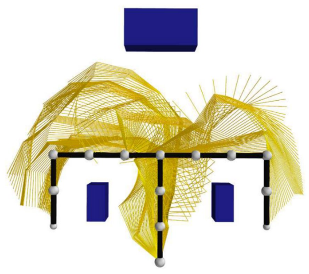


Figure 4.9: Scenario for an 8–DOF 2D manipulator by Lacevic et al. [3]. Start and goal configurations left and right.

Collision checking

Lacevic et al. perform collision checks differently in their two papers [3], [20]. In [3] they do not specify the step size ε they use in RRT, but in the source code there is $\varepsilon = 0.1$ for RRT. We have also set it to $\varepsilon = 0.1$, but in [20] they set it to $\varepsilon = 0.05$. Regardless, we assume that the authors used RRT step size $\varepsilon = 0.1$ and perform approximately 15 collision checks at every step as is written in the source code.

Bur calculation

A bur requires a distance check and calculation of its spine endpoints. We have already described how the distance check is performed and that we do it the same way as Lacevic et al. As for the spine endpoint calculations, we have found great differences in the internal implementations. The actual high-level spine calculation as defined by Algorithm 11 is identical between our two implementations. Internally, however, the radius calculations are quite different: we calculate them as has been generally defined in Equation 2.11, whereas Lacevic et al. have hardcoded the radius calculations for each robot as special functions. For each robot mentioned in their source code they have written specialized functions to calculate the radii: one function is for 2D robots and another is for the 6 degree-of-freedom xArm6 robotic manipulator. In addition, they add arbitrary constants to the encompassing radii, which they have defined in an additional `.yaml` file for each robot.

Scenarios

Another difference may stem from the type of scenario that the planners were tested on. Lacevic et al. test their RBT on scenarios where the start and goal configurations are in relatively free space away from obstacles as can be seen in Figure 4.9. The more free space there is around the manipulator the faster RBT runs, therefore in our scenarios, where we attempt to grasp in or pass through constrained spaces, RBT will run all the more slowly.

Profiling J+RRT and J+RBT

Let us investigate the implementation speeds. We have run both planners with the GNU profiler compiler option [30]. From that data we have mapped out how long each function takes to execute into Table 4.4 for J+RRT and Table 4.5 for J+RBT. We omit any functions that take up less than 5% of total runtime. Compiler optimizations were turned off.

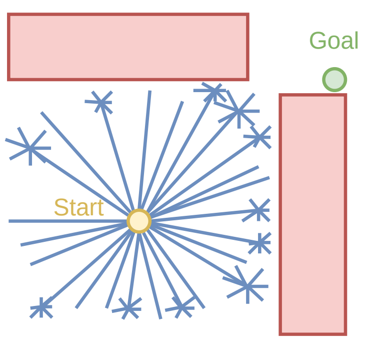
The function that we can most easily compare between the two algorithms is the SetClosestConfig function which, whenever a new configuration $\mathbf{q}_{\text{new}}$ is added to the tree, for every target grasp pose $g \in G$, it updates whether $\mathbf{q}_{\text{new}}$ is the closest configuration according to distance metric from Equation 4.1. Since the number of calls of SetClosestConfig is directly proportional to the number of nodes in the tree, and we know how many nodes per second each of the algorithms adds, then we can calculate the ratios of runtimes of the other functions of the two algorithms.

An indicator of a bad implementation of J+RBT would mean that the rate of node generation of J+RBT R_{J+RBT} would be comparable or smaller compared to R_{J+RRT} for J+RRT. SetClosestConfig takes up a constant amount of absolute time per node for both planners, therefore a larger relative node generation rate R_{alg} is better. To ease calculation we define N_{J+RRT} as the number of nodes generated per second by J+RRT and likewise for N_{J+RBT} for J+RBT; the values are given in Table 4.3. We also define S_{J+RRT} as the percentage of total runtime of the function SetClosestConfig within J+RRT and S_{J+RBT} for J+RBT; these are given in Tables 4.4 and 4.5. To fairly compare the node generation rates of the two planners we have to normalize out the time spent on SetClosestConfigs, i.e., normalize the number of nodes generated per second by the relative time spent generating them $1 - S_{alg}$. The relative node generation rate is thus

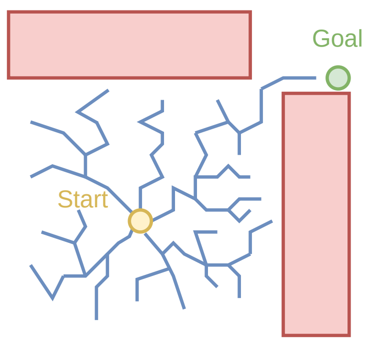
$$R_{alg} = \frac{N_{alg}}{1 - S_{alg}} \left[\frac{\text{nodes}}{\text{second} \cdot \text{relative node generation time}} \right]. \tag{4.2}$$

For J+RRT: $R_{J+RRT} = \frac{5200}{0.85} \approx 6100$; for J+RBT: $R_{J+RBT} = \frac{7600}{0.78} \approx 9700$. This result suggests that J+RBT creates many more equivalent nodes per second, which rules out more severe errors in implementation. The above equation states that every second, J+RRT spends 0.85 seconds generating nodes and 0.15 seconds checking which nodes are closest to the goal; conversely, J+RBT spends 0.78 seconds generating nodes and 0.22 seconds checking which nodes are closest to the goal while producing significantly more nodes in those 0.78 seconds.

The fact that J+RBT generates more nodes per second and on average spends less time on generating each node suggests that the concept of a bur is inefficient when implemented for the general case. Furthermore, constrained spaces significantly limit the volume of the complete collision-free bubble $\mathcal{CB}$, shrinking the available configuration space that can be explored by a single bur. During planning, J+RBT experiences plateaus more often than J+RRT, which suggests that there are cases where it is up to chance whether J+RBT finishes quickly or not. An illustration of such a case is depicted in



(a) : RBT tree growth.



(b) : RRT tree growth.

Figure 4.10: Comparison of the growth of RBT and RRT trees in the same amount of time.

Figure 4.10. As we can see, RBT generates many more nodes but often right next to obstacles, causing subsequent burs to be exceedingly small in size. Even with a 25% speed up in planning time thanks to adaptive burs [21] it would not be enough to catch up to J+RRT.

Function	Percentage of runtime [%]
IsColliding	40
NearestNeighbour	24
SetClosestConfig	15
ExtendToGoal	12

Table 4.4: J+RRT profiling result

Function	Percentage of runtime [%]
ClosestDistance	25
Bur	22
SetClosestConfig	22
NearestNeighbour	9
IsColliding	7
AddNode	6

Table 4.5: J+RBT profiling result

As mentioned in the preceding sections, Lacevic et al. save computation time by partially skipping the ClosestDistance computation for a few segments of one type of robot and some objects in the environment. Then they also simplified the bur calculation by computing the special cases of the projected encompassing radii for each robot they tested, not the general case as we have done. As it happens, their simplifications target precisely the two procedures that take up the most amount of time in RBT.

Special cases

The 2D manipulators such as the one in Figure 4.9 that Lacevic et al. have provided are represented as box primitives as opposed to meshes. The FCL library has specialized functions to speed up collision and distance queries for primitives, so having the robot represented as primitives might help speed up the operations. Not only that, all obstacles are also represented as convex box primitives which probably speeds up distance queries even further. The xArm6 robot, on the other hand, is represented as simple collision meshes. In the current library of their planners, Lacevic et al. have written specialized functions to approximate the robot and obstacles using shape primitives such as capsules, spheres and boxes. Such approximations can reduce the runtime of their algorithms even further by diminishing the runtime of ClosestDistance relative to other procedures.

Chapter 5

Conclusion

In this work we have been introduced to IK-RRT [16], J+RRT [16] and RBT [3] which has high exploratory capabilities in large patches of free space. We have derived a more general way to calculate burs for mobile robotic manipulators, making them applicable for a wide range of robot types. We then performed a statistical analysis of the planners on various grasping tasks in both easy and challenging scenarios created using Jogramop [24] and investigated why some planners work well where others fail.

We have found out that RBT generates more nodes per second than RRT, and yet its planning times are longer. This result suggests that the concept of a bur is inefficient when implemented for the general case. Moreover, constrained spaces such as the scenarios that we have tested the planners on significantly limit the volume the complete collision-free bubble $\mathcal{CB}$ can occupy. This limits the available configuration space that can be explored by a single bur. Lacevic et al. managed to adapt RBT such that it outperforms RRT in certain scenarios. As we have previously mentioned, their scenarios have fewer constrained spaces, and they reduce computation time by using solely convex obstacles comprising geometric primitives such as cubes and capsules. Finally, they simplified the bur calculation by computing special cases of the projected radii for each robot they tested, whereas we have derived bur calculation to work in the general case.

We have found that IK-RRT is the best planner overall. It alternates between finding collision-free inverse kinematics solutions to the target grasp poses and planning using bi-directional RRT. The speed of calculating inverse kinematics combined with the efficiency of bi-directional search seems to beat J+RRT and J+RBT in almost any scenario.

5.1 Future work

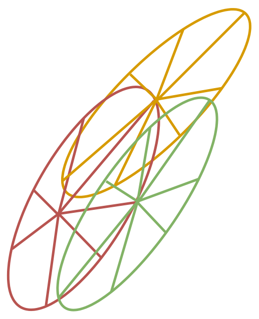
There are situations in which only one segment of the robot is in a restricted area. In that case, theoretically, the rest of the robot should be able to move freely while the one segment stays still. In Section 3.2 we mentioned the case where we can move later segments when lower segments are "stuck" between obstacles. We also mentioned the converse case where we would earlier segments should be movable when the later segments of the robot are restricted. This problem is partially addressed by expanding the definition of the complete bubble of free space $\mathcal{CB}$ from Equation 2.8. The same derivations that we have performed during the course of this work can be performed analogically using the distance profile $\mathbf{d}_c$, i.e., the distance to the closest obstacle for every segment. This set of collision-free points can be called the expanded bubble of free space $\mathcal{EB}$:

$$\mathcal{EB}(\mathbf{q}, \mathbf{d}_c) = \left\{ \mathbf{x} \in \mathcal{C}_{free} \;\middle|\; \begin{bmatrix} \|f_{\mathbf{p}_1}(\mathbf{q}) - f_{\mathbf{p}_1}(\mathbf{y})\| < \mathbf{d}_{c,1} \\ \vdots \\ \|f_{\mathbf{p}_n}(\mathbf{q}) - f_{\mathbf{p}_n}(\mathbf{y})\| < \mathbf{d}_{c,n} \end{bmatrix},\ \forall \mathbf{y} \in \overline{\mathbf{qx}} \right\}, \quad (5.1)$$

where n is the number of segments, $\mathbf{p}_i$ is the position of segment i, $\mathbf{d}_c$ is the vector of distances to the closest obstacles for each segment and $f_{\mathbf{p}_i}(\mathbf{q})$ is the forward kinematics of segment i.. The function $\varphi(t)$ from Equation 2.10 is then performed on a segment by segment basis and only the smallest result is used:

$$\varphi(t) = \min_{i=1\ldots n} \left(\mathbf{d}_{c,i} - \|f_{\mathbf{p}_i}(\mathbf{q}) - f_{\mathbf{p}_i}(\mathbf{y})\| \right), \quad (5.2)$$

Future work that combines the two approaches of extended burs, Algorithm 9, and expanded bubbles of free space, Equation 5.1, seem promising candidates for mitigating the effects constrained spaces have on the size of burs. In addition, recent work on optimal multi-trajectory motion planning using burs by Covic et al. [31] exploits the properties of burs even further. An illustration of what a multi-tree bur approach can achieve is depicted in Figure 5.1b. In Figure 5.1a we can see that when creating a single bur tree there is a lot of overlap between burs, and in extreme cases the coverage of the next bur can be over half of or even more than the area of the previous bur. A multi-tree approach would result in something more akin to Figure 5.1b. It minimizes overlap among burs by creating multiple trees starting in different collision-free configurations, thus leading to even greater utilization of the inherent advantages of burs of creating large patches of collision-free space. Or, analogically, one could grow a single tree by alternating between burs and RRT extend procedures to minimize overlap between neighbouring burs, which could also result in a structure such as the one in Figure 5.1b.


(a) : Regular bur tree.

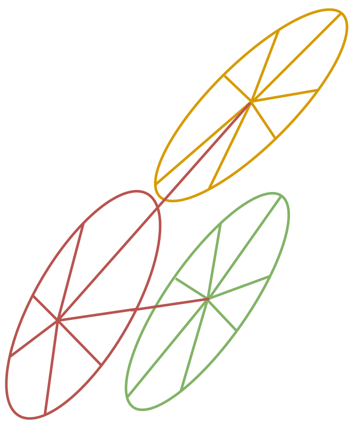

(b) : Alternative bur tree.

Figure 5.1: Comparison of bur trees.

Bibliography

[1] Chair of Production Systems, University of Leoben, “Robot Franka Emika Panda.” `https://cps.unileoben.ac.at/robot-franka-emika-panda/franka_panda_203/`, 2023. Accessed: 2024-04-17.

[2] Generation Robots, “List of criteria to look at before buying a robot arm.” `https://www.generationrobots.com/blog/en/list-of-criteria-to-look-at-before-buying-a-robot-arm/`, October 2019. Image: Panda Franka Emika care robot arm.

[3] B. Lacevic, D. Osmankovic, and A. Ademovic, “Burs of free c-space: A novel structure for path planning,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 70–76, 2016.

[4] ClipartMag, “Moving Day Clipart.” `https://clipartmag.com/moving-day-clipart`, [Date]. Accessed: 2024-04-17.

[5] The Independent, “Friends 20th anniversary: Best moments from the show as chosen by you,” *The Independent*, September 2014. Accessed: 2024-04-17.

[6] J. H. Reif, “Complexity of the mover’s problem and generalizations,” in *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pp. 421–427, 1979.

[7] E. Drumwright and V. Ng-Thow-Hing, “Toward interactive reaching in static environments for humanoid robots,” in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 846–851, 2006.

[8] M. Rudorfer, M. Suchi, M. Sridharan, M. Vincze, and A. Leonardis, “Burg-toolkit: Robot grasping experiments in simulation and the real world,” 2022.

[9] E. Coumans and Y. Bai, “Pybullet, a python module for physics simulation for games, robotics and machine learning.” `http://pybullet.org`, 2016–2021.

[10] B. Calli, A. Walsman, A. Singh, S. Srinivasa, P. Abbeel, and A. M. Dollar, "Benchmarking in manipulation research: Using the yale-cmu-berkeley object and model set," *IEEE Robotics and Automation Magazine*, vol. 22, no. 3, pp. 36–52, 2015.

[11] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," *Research Report 9811*, 1998.

[12] J. Kuffner and S. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 2, pp. 995–1001 vol.2, 2000.

[13] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller, "Anytime motion planning using the rrt*," in *2011 IEEE International Conference on Robotics and Automation*, pp. 1478–1483, 2011.

[14] M. Kalisiak and M. van de Panne, "Rrt-blossom: Rrt with a local flood-fill behavior," in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pp. 1237–1242, 2006.

[15] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[16] N. Vahrenkamp, D. Berenson, T. Asfour, J. Kuffner, and R. Dillmann, "Humanoid motion planning for dual-arm manipulation and re-grasping tasks," in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, (St. Louis, MO), pp. 2464–2470, IEEE, Oct. 2009.

[17] A. Shkolnik and R. Tedrake, "Path planning in 1000+ dimensions using a task-space voronoi bias," in *2009 IEEE International Conference on Robotics and Automation*, pp. 2061–2067, 2009.

[18] S. Quinlan, *Real-time modification of collision-free paths.* Stanford University, 1995.

[19] C. Rösmann, F. Hoffmann, and T. Bertram, "Integrated online trajectory planning and optimization in distinctive topologies," *Robotics and Autonomous Systems*, vol. 88, pp. 142–153, 2017.

[20] B. Lacevic and D. Osmankovic, "Path Planning for Rigid Bodies Using Burs of Free C-Space," *IFAC-PapersOnLine*, vol. 51, no. 22, pp. 280–285, 2018.

[21] B. Lacevic, D. Osmankovic, and A. Ademovic, "Path planning using adaptive burs of free configuration space," in *2017 XXVI International Conference on Information, Communication and Automation Technologies (ICAT)*, (Sarajevo), pp. 1–6, IEEE, Oct. 2017.

[22] B. Lacevic and D. Osmankovic, "Improved C-Space Exploration and Path Planning for Robotic Manipulators Using Distance Information," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, (Paris, France), pp. 1176–1182, IEEE, May 2020.

[23] M. Vande Weghe, D. Ferguson, and S. S. Srinivasa, "Randomized path planning for redundant manipulators without inverse kinematics," in *2007 7th IEEE-RAS International Conference on Humanoid Robots*, (Pittsburgh, PA, USA), pp. 477–482, IEEE, Nov. 2007.

[24] M. Rudorfer, J. Hartvich, and V. Vonásek, "A framework for joint grasp and motion planning in confined spaces," in *Robot Motion and Control (RoMoCo), 2024 12th International Workshop on*, July 2024. Accepted in April 2024.

[25] Orocos KDL, "The kinematics and dynamics library," 2024. Accessed: 2024-03-18.

[26] University of North Carolina at Chapel Hill, "Proximity Query Package (PQP)." `http://gamma.cs.unc.edu/SSV/`, 1999. Accessed: 2024-04-21.

[27] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International Conference on Computer Vision Theory and Applications*, 2009.

[28] MathWorks, "Plan motion for rigid body tree using bidirectional rrt." `https://www.mathworks.com/help/robotics/ref/manipulatorrrt.html`, 2024. Accessed: 2024-05-08.

[29] robotics ETF, "Rapid prototyping motion planning library v2 (rpmplv2)." `https://github.com/robotics-ETF/RPMPLv2`, 2024. Accessed: 2024-05-18.

[30] GNU Project, *GNU gprof: The GNU Profiler*. Free Software Foundation, 2024. Accessed: 2024-05-18.

[31] N. Covic, D. Osmankovic, and B. Lacevic, "Asymptotically Optimal Path Planning for Robotic Manipulators: Multi-Directional, Multi-Tree Approach," *Journal of Intelligent & Robotic Systems*, vol. 109, p. 14, Sept. 2023.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Hartvich Jiří**	Personal ID number:	**483636**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

Sampling-Based Motion Planning Using Burs of Free-Space

Master’s thesis title in Czech:

Rychlé plánování pohybu manipulátorů

Guidelines:

1. Get familiar with the motion planning problem and sampling-based planning techniques like Rapidly-exploring Random Tree and its variants [1]. Study problem of path planning for robotic manipulators [6].
2. Implement a variant of RRT for mobile manipulators using inverse kinematics, e.g., RRT-IK [4], and adapt it to mobile manipulator robots. Use the BURG toolkit as a model/simulation of the mobile manipulator and the environment [5].
3. Implement Burs-RRT [2,3] and adapt it for mobile manipulator robots.
4. Extend a selected planner from tasks 2) or 3) for grasping in constrained environments. An external library gives grasping positions. The task is to move the robot as close as possible to the grasping position. Multiple grasping positions are available for each object. Assume that final positions are given only in the task space (i.e., without inverse kinematics). Grasping is not part of this task and does not need to be solved.
5. Compare methods from tasks 2) and 3) in a set of scenarios for grasping in constrained environments. Consider „easy“ scenarios (robot motion is not blocked by obstacles), as well as „challenging “ scenarios (robot or object to be grasped is behind an obstacle). Statistically evaluate the performance of the planners.

Bibliography / sources:

[1] S. M. LaValle, Planning algorithms, 2006, Cambridge press.
[2] Lacevic, Bakir, and Dinko Osmankovic. “Path Planning for Rigid Bodies Using Burs of Free C-Space.” IFAC-PapersOnLine 51, no. 22 (2018): 280–85. https://doi.org/10.1016/j.ifacol.2018.11.555.
[3] Lacevic, Bakir, Dinko Osmankovic, and Adnan Ademovic. “Burs of Free C-Space: A Novel Structure for Path Planning.” In 2016 IEEE International Conference on Robotics and Automation (ICRA), 70–76. Stockholm, Sweden: IEEE, 2016. https://doi.org/10.1109/ICRA.2016.7487117.
[4] Vahrenkamp, Nikolaus, Dmitry Berenson, Tamim Asfour, James Kuffner, and Rudiger Dillmann. “Humanoid Motion Planning for Dual-Arm Manipulation and Re-Grasping Tasks.” In 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2464–70. St. Louis, MO: IEEE, 2009. https://doi.org/10.1109/IROS.2009.5354625.
[5] Martin Rudorfer, Markus Suchi, Mohan Sridharan, Markus Vincze, Aleš Leonardis, BURG-Toolkit: Robot Grasping Experiments in Simulation and the Real World.
[6] Kevin M. Lynch and Frank C. Park: Modern Robotics: Mechanics, Planning, and Control", Cambridge University Press, 2017, ISBN 9781107156302.

Name and workplace of master’s thesis supervisor:

Ing. Vojtěch Vonásek, Ph.D. Multi-robot Systems FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **23.01.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Vojtěch Vonásek, Ph.D.	prof. Dr. Ing. Jan Kybic	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature