Task-Trajectory Analysis Package in the Robot Operating System

by

Christina Elisabeth **Petlowany**

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2019

The Thesis Committee for Christina Elisabeth Petlowany Certifies that this is the approved version of the following thesis:

Task-Trajectory Analysis Package in the Robot Operating System

APPROVED BY

SUPERVISING COMMITTEE:

Sheldon Landsberger, Co-Supervisor

Mitch Pryor, Co-Supervisor

Dedication

To my perfect children: Penelope and Edgar.

Acknowledgments

This work would not have been possible without the unwavering support of Dr. Mitch Pryor and Dr. Sheldon Landsberger. I would also like to thank all current members and many past members of the Nuclear and Applied Robotics Group for their mentoring and friendship. Lastly, I would like to thank my family and friends. This work was supported by Los Alamos National Laboratory and a United States Department of Energy Nuclear Engineering University Program fellowship.

Abstract

Task-Trajectory Analysis Package in the Robot Operating System

Christina Elisabeth Petlowany, M.S.E.
The University of Texas at Austin, 2019

Co-Supervisor: Sheldon Landsberger
Co-Supervisor: Mitch Pryor

For many manufacturing tasks, such as welding and cutting, the task trajectory, or path, is known *a priori* in the object's reference frame. What is not known is whether or not the robot can reach the entirety of the trajectory given the relative location of the object frame to the robot's base frame and its reachable and/or dexterous workspace. The problem increases in complexity with each additional object in the robot's workspace. Some robots need to perform tasks in cluttered or confined environments, such as a glovebox, and the ability to know if and where the manipulator can perform a certain task is crucial for both design and operation.

This thesis describes the development, design, and implementation of a Task-Trajectory Analysis Package (T-TAP) within the Robot Operating

System (ROS) framework. Reachability has been extensively discussed in the literature, but current reachability visualization tools do not account for task data, and instead describe the robot's global workspace and thus take a long time to compute. Such tools may be useful for designing robotic systems, but their value diminishes when analyzing a specific task and environment. T-TAP focuses on the task space and is capable of producing real-time or near real-time feedback about the validity of a path. The results are shown in an easy-to-interpret visualization of the path points and their relative quality as measured using selected performance metrics.

T-TAP contains several capabilities. The first, and simplest, validates reachability for discrete points along the trajectory. An inverse kinematic (IK) solver is used to plan from one trajectory point to the next. The user can use standard ROS IK solvers or utilize their own IK solver. Next, T-TAP uses the Jacobian to analyze the system's performance as it completes the proposed trajectory. It ensures that joint and velocity limits are not violated, singularities are avoided, and is extensible to include additional user-define performance metrics.

T-TAP requires no prior computations, is hardware agnostic, and can be run entirely in simulation. It can reduce the time required to place and plan a trajectory by an order of magnitude. It is designed to work seamlessly with existing ROS path-planning packages. The operator needs only to send the path to T-TAP and T-TAP will analyze the trajectory. This information will allow the operator to intelligently adjust the path so that it is reachable

and viable.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The nuclear industry has been slow to incorporate automation. However, worker safety concerns like radiation exposure and ergonomic injuries make the introduction of robotics to nuclear manufacturing an ideal option. The nuclear industry may adopt robotic systems more quickly after it is easier to deliver robotic solutions for different tasks. Gloveboxes, see Figure 1.1, are used in the nuclear industry to protect workers from radiation. However, the use of gloveboxes commonly leads to ergonomic injuries for the workers. Additionally, the weakest point of the glovebox is the glove, which means the spot most likely to fail and cause a breach is where the worker handles the radioactive material. For these, and many other reasons, researchers have focused on automating glovebox activities.

One example of successful glovebox automation is the Los Alamos National Laboratory (LANL) Advanced Recovery and Integrated Extraction System (ARIES), shown in Figure 1.2 [34]. ARIES is an automated system that dismantles plutonium pits. While ARIES was successful, it can perform only one task and is not easily adaptable to other tasks.

The Nuclear and Applied Robotics Group (NRG) at The University of

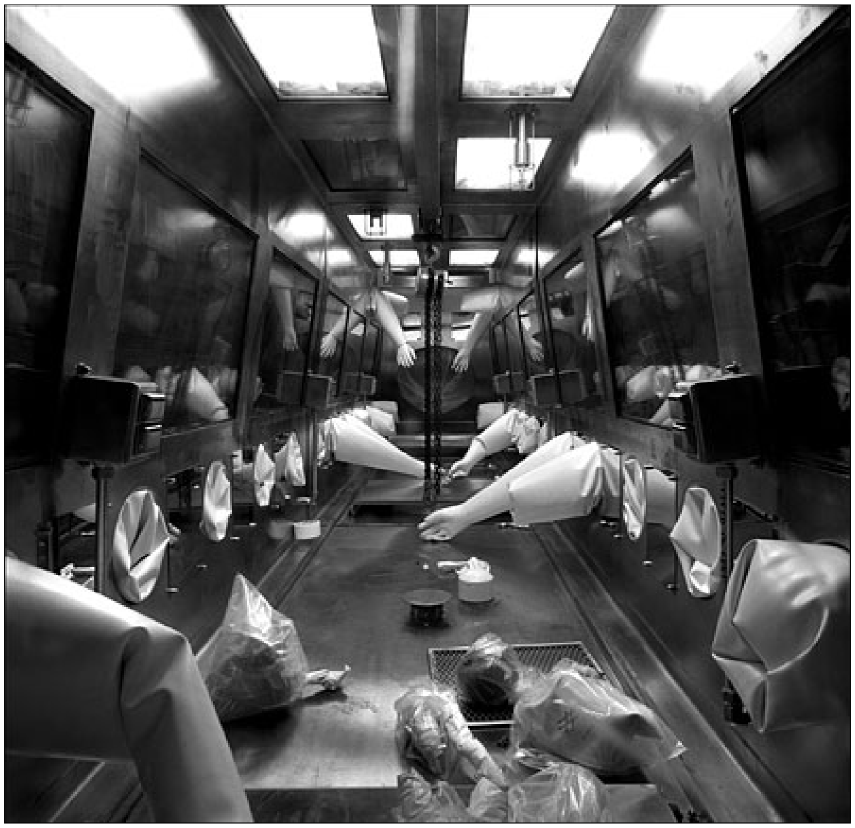


Figure 1.1: A glovebox as used in the nuclear industry at the Rocky Flats Nuclear Weapons Plant [42].

Texas at Austin has made several forays into glovebox automation. Peterson et al. [32] demonstrated material reduction inside of a glovebox. Paredes et al. [31] performed three tasks inside a multi-use glovebox: drilling, sorting, and surface finishing, shown in Figure 1.3. The last task provided the impetus for this effort. Paredes et al. used the ROS Descartes trajectory planning package [14] in order to create smooth paths over the surface of an object. However, the Descartes planning system fails if the robot fails to reach just one point and does not currently provide information to the user on which

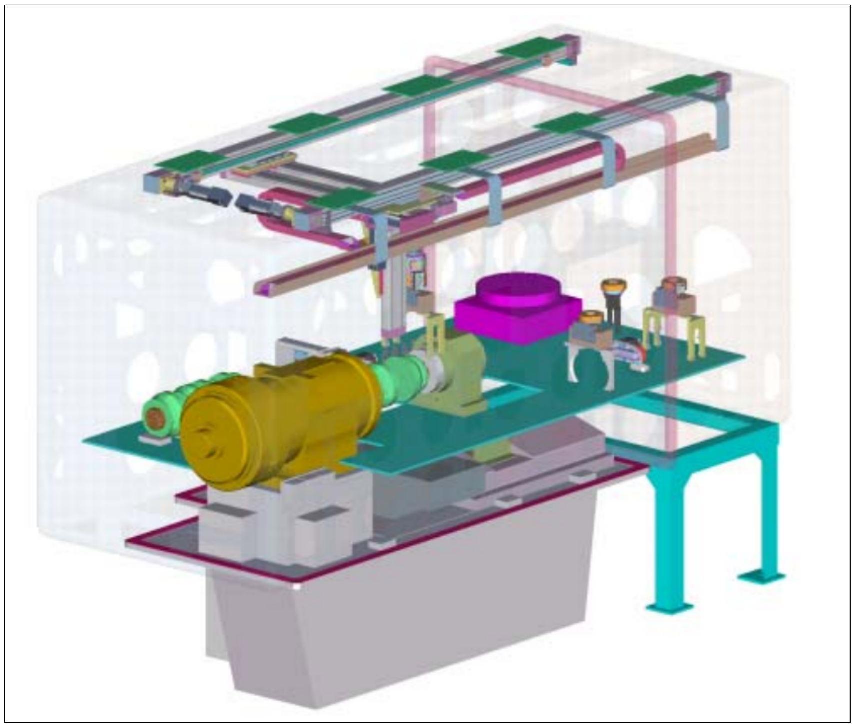


Figure 1.2: Model of the ARIES glovebox [34].

point, or points, in the trajectory failed, or why. Therefore, the user must iteratively attempt to find a location to execute the trajectory where the robot can a) reach all of the points and b) execute the trajectory without violating joint limits, velocity limits, or other constraints. A way to analyze the health of any given trajectory would greatly improve robot trajectory planning and execution.

Additionally, the glovebox is a relatively small cubicle volume, about 100 cubic feet, and, as in the case of Paredes et al. [31], quickly becomes crowded when multiple tasks need to be performed. The addition of one task may cause other tasks to be rearranged or removed. The ability to identify

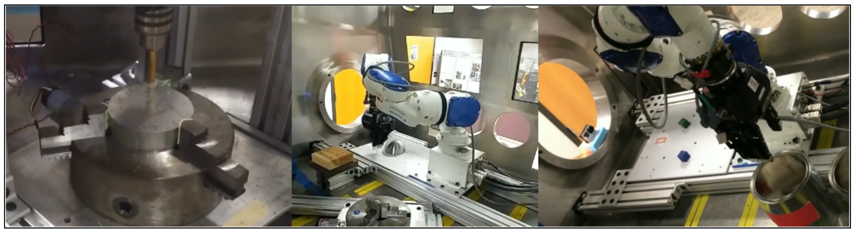


Figure 1.3: The three tasks of the multi-use workcell: drilling, sorting, and surface finishing [1].

where tasks can be performed would help the user design the workcell and easily add more tasks. Beyond glovebox automation, such a tool would be useful in other application areas and domains including small batch manufacturing, cooperative robotics, surgical robotics and assistive robots where task trajectories are often known and repeatable. There have been previous efforts related to this problem in the literature and industry which are discussed in more detail in 2. For example, there are multiple research efforts that have attempted to describe, define, and/or visualize the space the robot can reach.

1.1 Definitions

Below we briefly define several common terms in robotic kinematics that are related to this effort. This section is included in the introduction to create a shared vocabulary for the reader. A **point** in Cartesian space is defined as $\mathbf{x} = \begin{bmatrix} x & y & z & \alpha & \beta & \gamma \end{bmatrix}$.

For this paper, a **path** is defined as a set of ordered points through which the tool of the robot must pass. The definition of a **trajectory** is similar to a path but includes velocity, acceleration or other derivative information for the robot's **end-effector** (EEF) or tool. Therefore, a path informs only the curve's location whereas a trajectory is both position- and time-based. While a path is primarily restricted by the dexterous workspace and joint limits, trajectories can be further restricted by velocity limits, acceleration limits, singularities, and other peculiarities in a given rate-based inverse kinematic algorithm. Furthermore, if the task involves a payload or contact, the system may be limited by payload, actuator current limits or joint torque limits.

Joint limits are required for joints that cannot rotate indefinitely. Joints also have velocity limits to ensure that safe velocities are not exceeded. A *singularity* occurs when the configuration of the robot is such that the robot cannot move in a certain direction [28], for example if two axes of rotation are coincident, then motion in an output manifold is lost.

Intuitively, a robot arm has a limited reach much like a human arm. As an example, a two *Degree-of-Freedom* (DoF) robot will be able to reach an area in a plane as shown in Figure 1.4. Most joints are **prismatic**, or sliding joints, or **revolute**, or rotating joints. The number of directions a joint can move in determines the DoF of a particular joint. The DoF of the robot, n, is usually equal to the sum of every joint's DoF. A three DoF robot with non-coplanar joints will be able to place its EEF within a volume in space ($SE(3)$). The reachable volume or area is defined as the robot **workspace**, see Figure 1.5.

In order to reach a specific point (x, y, z) with a given orientation (x_{rot}, y_{rot}, z_{rot}), a robot must have at least six DoF. The area of a workspace where a robot can reach a point from any orientation is known as the **dexterous**

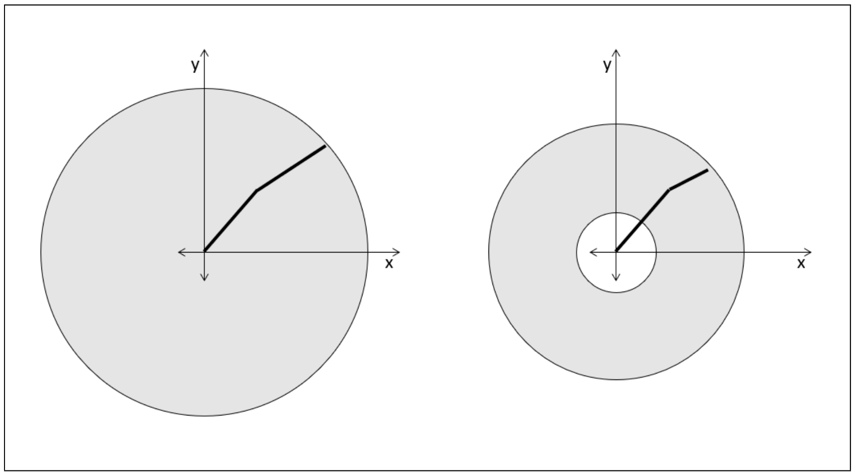


Figure 1.4: The workspace (in gray) of a two-link robot arm that rotates about the origin. Left: the link lengths are equal. Right: the second link is shorter than the first link.

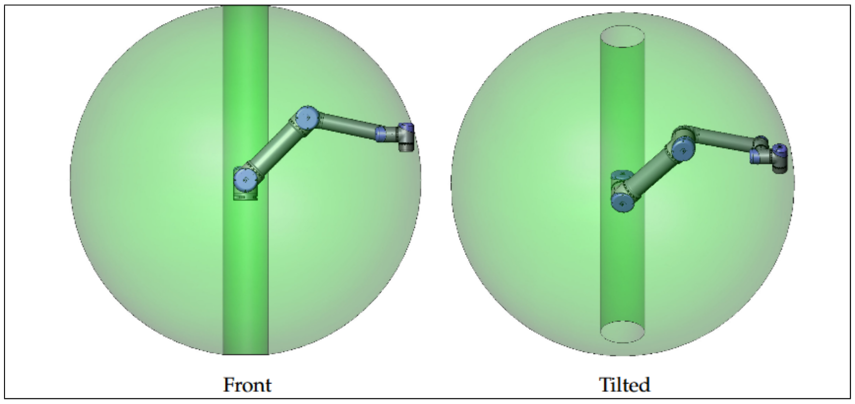


Figure 1.5: The workspace of a Universal Robotics UR3 robot arm [3].

workspace [26]. The collection of points a robot can reach is known as the **reachable workspace**. Robots with greater DoF, n, than the dimension of the task space, m, are **redundant**, $n > m$, as the robot can approach the EEF location spanning one or more continuous joint position ranges or **manifolds** [40]. The **task workspace** is different than the robot workspace.

For example, a pick and place task may require only 3 DoF ($m = 3$) if the object can be picked up from any orientation.

Frames define a point of origin for a coordinate system with respect to parts of the robot. For example, for a fixed manipulator, the **base frame** is a stationary frame located at the origin at the base of the robot. In the case of mobile robots, the base frame may move. The **tool frame** has its origin at the EEF or tool point and moves with the tool. There is a frame for each joint and the math for transferring from one frame to another can be used to determine the configuration dependent **Jacobian matrix** which is a linear mapping of the joint velocities to the EEF velocities.

The transformation matrix from reference frame i to frame j is given by the matrix ${}^{i}_{j}\mathbf{A}$:

$${}^{i}_{j}\mathbf{A} = \begin{bmatrix} {}^{i}_{j}\mathbf{R} & {}^{i}_{j}\mathbf{o} \\ 0 & 1 \end{bmatrix} \tag{1.1}$$

Where ${}^{i}_{j}\mathbf{R}$ is the relative rotation and ${}^{i}_{j}\mathbf{o}$ is the displacement between i and j. To move between successive reference frames, we multiply A matrices.

Kinematics are the study of how different robot joint positions translate to positions in space, as shown in Equation 1.2 where $\mathbf{x}$ are the EEF coordinates in space and $\boldsymbol{\Theta}$ are the joint positions. $\mathbf{J}$ is the Jacobian matrix that maps the joint velocities to Cartesian EEF velocites.

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\boldsymbol{\Theta}} \tag{1.2}$$

Inverse kinematics translate the Cartesian position of an EEF to the associated robot joint positions. Inverse kinematics are used to command the robot to desired positions. They solve for $\mathbf{\Theta}$ in the above equation. Calculating the inverse of the Jacobian is impossible when the Jacobian is not a square matrix (number of rows is equal to number of columns). When this occurs, the **Pseudoinverse** is commonly used to find a valid inverse of the Jacobian. The **null space** of a Jacobian is the set of vectors $\mathbf{v}$ where:

$$\mathbf{Jv} = \mathbf{0} \tag{1.3}$$

Performance metrics provide a way to monitor the performance of a robotic system with respect to some operational interest. For example, the **condition number** is a metric that correlates mathematically to a robot's proximity to singular positions. **Collision distance** can be used to describe the distance to collision. **Joint Range Availabilty (JRA)** characterizes robot positions by the closeness of the joint limits [18].

The **dynamics** of a robotic system model how a robot moves with respect to force inputs including joint torques, EEF forces, gravity compensation, Coriolis/centripedal effects and body contact forces. For this effort, the dynamics of the system can largely be ignored as tasks are within the industrial manipulator's payload limitations. Such payloads are typically determined by considering mass moved at both full acceleration and extension. Since glovebox motions are sufficiently slow to be considered **quasi-static** and the manipulators cannot be fully extended in the confined environments, issues related to system dynamics can be neglected. Scenarios where the system dynamics cannot be neglected are considered outside the current scope of this effort, but such efforts would also include the methodologies presented in this effort.

1.2 Problem Statement

The reachability analysis methods found in the vast majority of the literature (and reviewed in 2) do not include task-based information. Typically, the task agnosticism and generalization of these approaches has been a touted feature, but these approaches have rarely been applied to design and/or deploy systems as complex as glovebox automation dealing with confined/crowded spaces, trajectory complexity, and task variation/uncertainty. Based on our experience, existing tools, while useful for evaluating a system's capabilities, are not useful for investigating an *existing* system's ability to complete *new* tasks. The identified tools are:

- Computationally expensive, and cannot provide immediate feedback to a system developer,
- Do not include the obstacles in their analysis,
- Do not account for the task spaces where $m < 6$, and
- Typically focus on reachability which can validate paths, but not trajectories.

Thus, the Task-Trajectory Analysis Package (T-TAP) would greatly help developers more quickly and easily utilize robotic systems to complete new or modified task trajectories.

1.3 Objectives

The objective of this thesis is to develop improved visualization techniques for evaluating a robot's ability to execute complex task trajectories and then implement the techniques as a tool in ROS. The focus will be on stationary serial manipulators, but without any assumptions concerning the joint types, number of joints, or dimension of the task space. While the main focus of this effort is to aid the user in choosing a location for the trajectory in the global space, the T-TAP will also help users select functional workspace layouts in crowded environments.

The key functions of such a tool should include:

- generalized to work with any task spaces where $m \leq 6$;
- alerting the user to which designated points along the trajectory's path are reachable (or not);
- for reachable paths, informing the user if the trajectory will fail and and for what reason(s) (joint limit violation, velocity limit violation, collision, singularity, etc.);
- visually reporting these results to the user; and

- visualizing calculated performance metrics over the trajectory and simplifying the inclusion of new metrics.

Furthermore, efforts will be made to ensure the usability of the proposed system including minimizing the calculation and simplifying the inclusion of new environmental factors such as new obstacles. The list above shows that the problem is divided into two parts, the first dealing with whether a robot can reach all of the points in the path. This step is simple, but necessary to quickly and efficiently filter out poor task or robot placement. Once the robot can reach the path, the second part deals with whether the robot can reach those points given the timing constraints on the trajectory, notes if and where the trajectory fails and how, and provides performance metrics for the trajectory to monitor the "health" of the execution. These performance metrics will provide the operator with insights on how to modify the task (EEF velocity, part placement, rearranging other glovebox items, etc.) so that it can be completed.

1.4 Organization

Chapter 1 showed the need for tools that describe a robot's workspace (particularly for complex tasks in confined environments) and reviewed the relevant terms and concepts. Chapter 2 reviews related methods in the literature or industry for analyzing workspaces. Chapter 3 describe the design of the T-TAP and its capabilities. Chapter 4 discusses the implementation of the

T-TAP into ROS. Chapter 5 demonstrates and evaluates the T-TAP in use for relevant test cases using both simulation and hardware as well as compares its performance and capabilities to previously identified methodologies from the literature. Chapter 6 summarizes the developed tool(s), documents areas for future work, and concludes the thesis.

Chapter 2

Literature Review

This chapter covers efforts related to defining a system's capability to perform tasks and current trajectory planners used to complete such tasks. Most tools currently focus on reachability so they are reviewed first. Second we review existing planners. Since our goal is not to develop trajectory planners, but rather to more effectively utilize existing planners, it will be necessary to understand current practices and capabilities. Next, since the goal is to give the system designer/developer information that assists them with task placement(s) within a robot's workspace, the literature related to evaluating robotic performance in general is reviewed to identify additional performance parameters that may be of use as a part of this effort.

2.1 Existing Reachability Tools

Reachability tools have mostly focused on problems related to system design or understanding the overall capabilities of existing systems. Thus, most solutions use a pre-calculated reachability map. These are calculated with forward kinematics, inverse kinematics, or using other methods such as link sweeping. They can be roughly categorized into those that leverage forward kinematics, inverse kinematics, or algorithms that do not rely on the kinematic model at all.

2.1.1 Forward Kinematics and No Kinematics

Forward-Kinematics-based and No-Kinematics-based methods typically create reachability maps through iterative methods, such as Monte Carlo sample or joint sweeping. Müller et al. [30] created a reachability map used for identifying reachable poses for grasping. Each arm joint is rotated 0.5 degrees in between joint limits to determine the reachable workspace. The reachability map is precomputed for the 5-DoF Nao robot arm, as shown in Figure 2.1, and used to grasp coffee mugs. Additional motion planning capabilities are needed to account for obstacles. The shortest distances between the fingers and the obstacles are calculated for each point in the possible paths to ensure there are no collisions. Burget and Bennewitz [9] extend the forward kinematics reachability map to an inverse reachability map that can provide suitable robot base locations, as shown in Figure 2.2 for performing tasks. This method was implemented on the Nao humanoid robot for use with specified grasping poses. Alciatore and Ng [5] use random sampling of joint positions to describe the workspace with the Monte Carlo Method, see Figure 2.3. They do not discuss obstacle avoidance.

A few solutions exist that do not utilize inverse or forward kinematics. The sweeping method [15] starts at the last link in the robot, moves the link through its full range of motion, takes the created shape and sweeps it

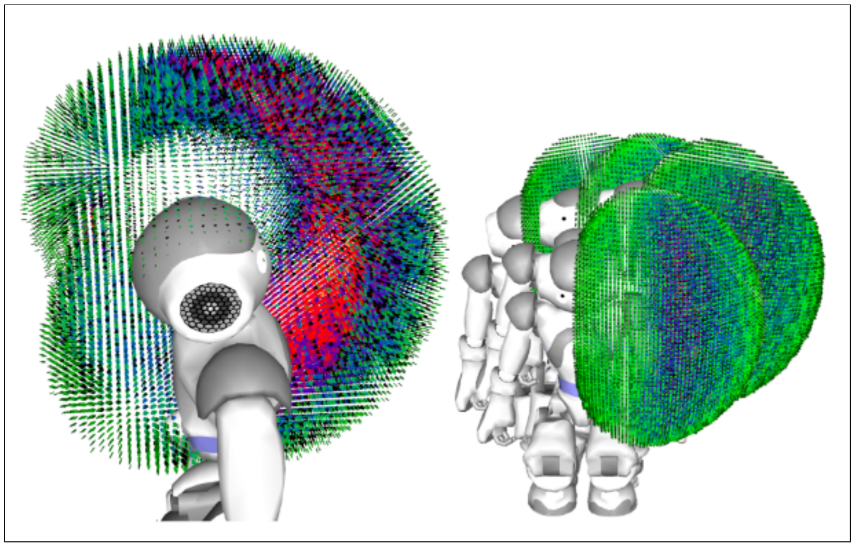


Figure 2.1: Reachability map of the Nao robot where red is most reachable and green is least reachable [30].

along the next link's range of motion. This is repeated until there a full representation of the reachable workspace, though it is hard to visualize vacancies internal to the outer boundary of the swept volume. While computationally efficient, this approach also does not account for self-collision or calculate the dexterous workspace. Kieffer et al. [21] use neural nets to learn the relationship between Cartesian EEF coordinates and joint positions for a 2 DoF robot. This method does not include collision objects. Baranes and Oudeyer [6] use active learning to obtain an inverse kinematics relationship. They use a redundant 15 DoF robot and have the robot learn its reachability in the task space while performing a task.

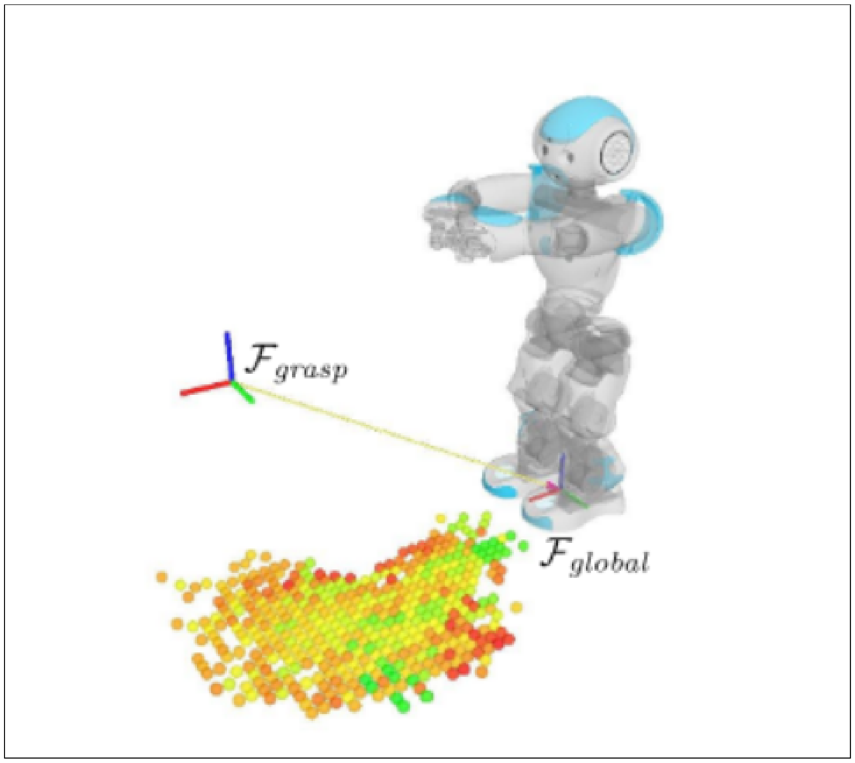


Figure 2.2: Suitable base positions of the Nao robot for a given grasp [9].

2.1.2 Inverse Kinematics

Most Inverse-Kinematics-based (IK) methods divide the possible workspace (usually estimated as a cube with sides twice as long as the robot arms and the robot centered in the cube) into discretized points. These methods use IK techniques to plan to each point from different angles of approach in order to determine reachability:

$$ReachabilityIndex = R/N * 100 \tag{2.1}$$

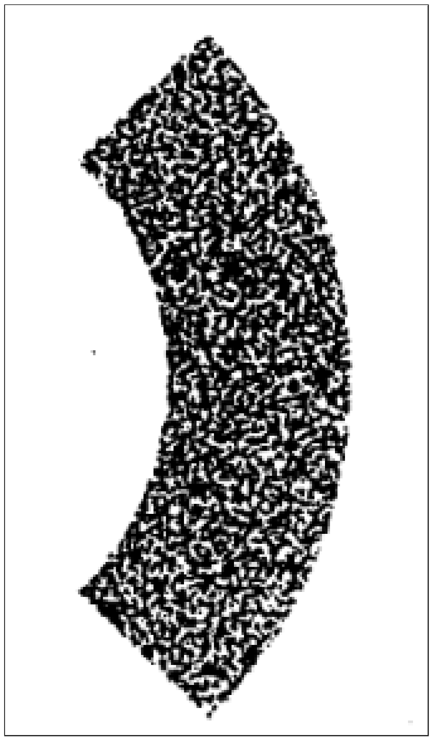


Figure 2.3: Monte Carlo points for a 2D robot workspace [5].

Where N is the number of possible approaches to the point and R is the number of reachable approaches. This reachability map can then be used to find suitable robot base positions or task trajectory locations.

2.1.2.1 Inverse Kinematics Reachability Maps

Zacharias et al. [48] provide a method for workspace discretization to create a reachability map. Zacharias et al. also created a capability map. The capability map is a reachability map featuring cones instead of spheres that give an indication of the directions from which the robot can reach each location. In the realm of trajectory reachability, Zacharias et al. use the reachability map to search for plausible linear trajectories [47] and 3D trajectories [49]. In both papers, collision checking occurs after trajectories are generated but before they are executed. In [49] they note problems sampling trajectories, namely aliasing where infrequent sampling can lead to curve smoothing, as shown in Figure 2.4.

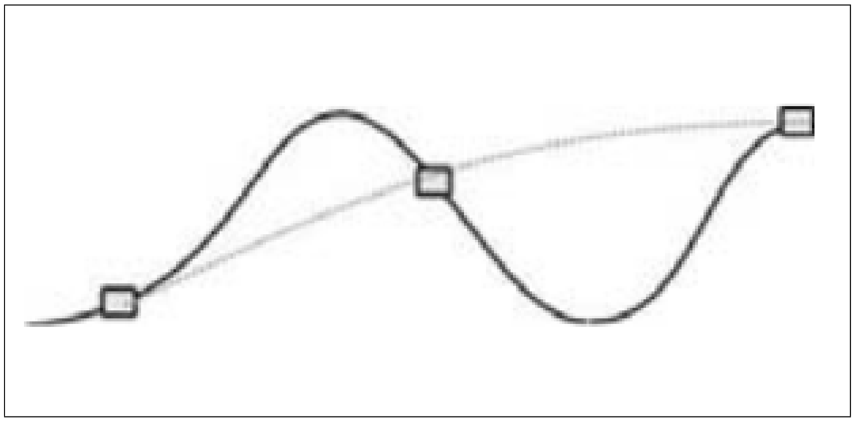


Figure 2.4: Improper sampling frequency leading to a smoother curve [49].

Vahrenkamp et al. [43] take the inverse of the reachability map. The inverted reachability map shows if an object or pose is reachable from a base position. Dong and Trinkle [13] create a map that can support switching robot EEFs. They name it the *orientation-based reachability map* and it is organized by orientation. Whereas applying transforms destroys position-based reachability maps, applying a transform to an orientation-based reachability map creates a different orientation-based reachability map. Diankov implemented the reachability and inverted reachability maps into software called Open Robotics Automation Virtual Environment (OpenRAVE) [12]. Diankov also included a powerful inverse kinematics solver called `ikfast` which can solve the IK problem for robot chains with prismatic or rotary joints. This cannot solve for timed trajectories and is difficult to implement for each robot. If the robot configuration changes, the solver must be recalculated.

In the NRG, Williams created a manipulabity-based method of determining task reachability [45]. Williams determines the manipulability of a task plane based on a robot position. In this case, manipulability differs from reachability in that reachability is calculated over a sphere and manipulability is calculated over a circle. This information is used to help plan robot locations for performing tasks in a glovebox.

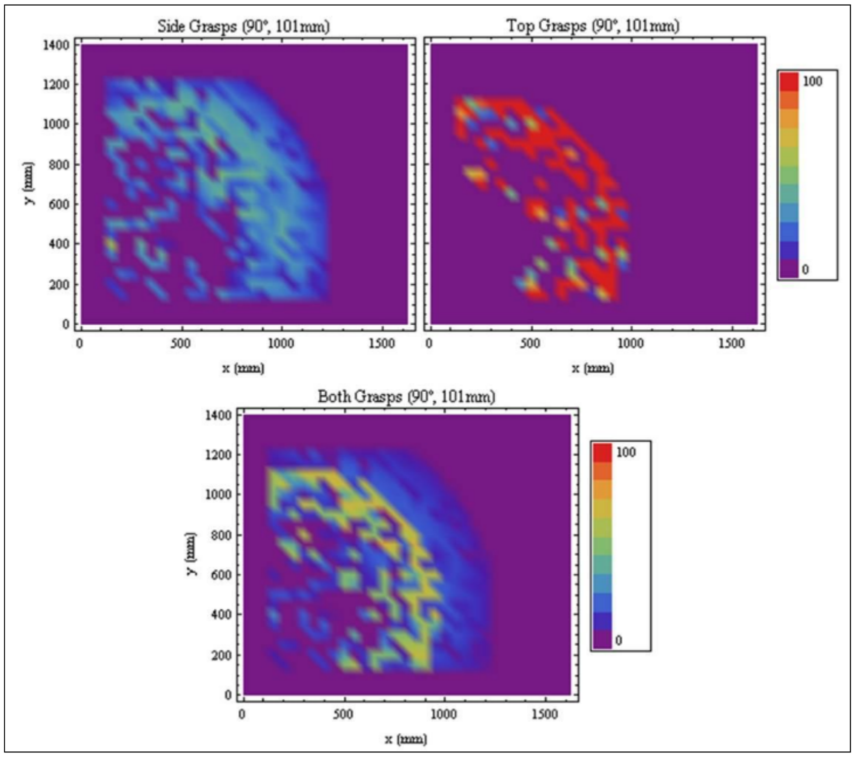


Figure 2.5: Task planes for different grasping positions for a Yaskawa Motoman SIA-10 robot [45].

One package that exists in ROS is the Reuleaux package [29], based on [48]. The Reuleaux package begins its calculations by first discretizing the robot workspace. It creates a cubic estimation of the robot's workspace which it then divides into discrete units. Each unit is filled with a sphere. Then, the program removes poses where the robot will be in collision with its self.

Each of the spheres created during the workspace discretization is surrounded by possible poses pointing towards the sphere, as shown in Figure 2.6. For each of these poses, Reuleaux uses an IK solver to determine whether the pose is reachable or not. This is translated into a reachability map, as shown in Figure 2.7.

Reuleaux can also create a capability map to show the directionality of the reachability map and, given task poses, determine suitable base locations for mobile robots to perform a task using the inverse of the reachability map. Reuleaux does not incorporate obstacle avoidance nor is it extensible to all robots. The IK method used to solve for all points in the robot workspace is suitable only for < 8 DoF robots. Additionally, computation times often exceed two hours depending on the density of the reachability map. However, it can work for all task spaces depending on the search method used by the developer to identify acceptable areas for performing tasks.

2.2 Task Trajectory Planners

The sections above outline existing tools for determining either reachability of a given robot in terms of its reachable workspace, dexterous workspace,

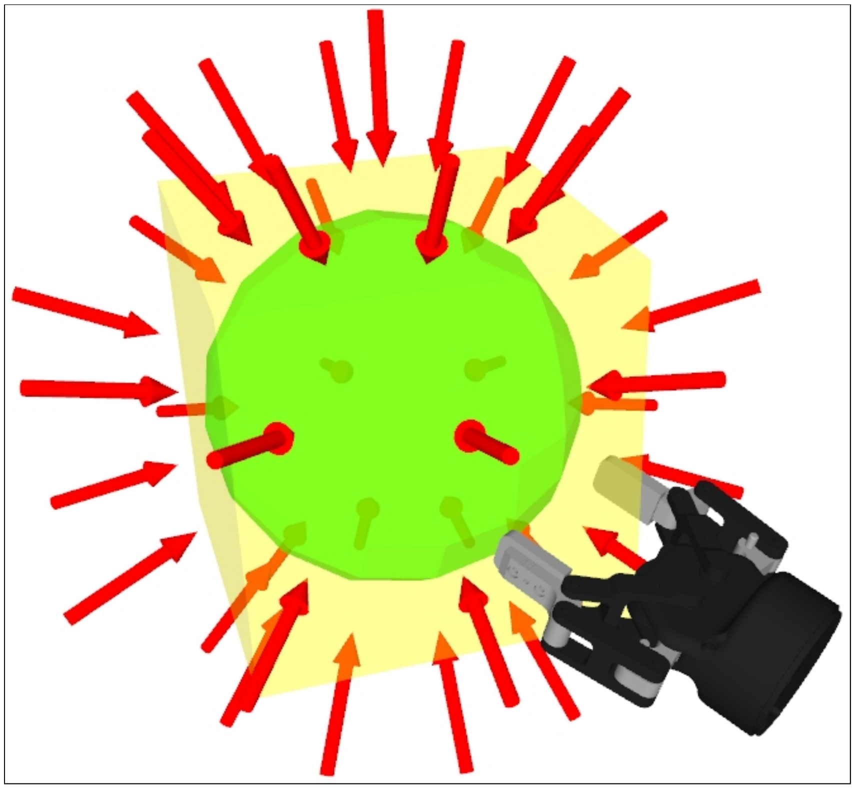


Figure 2.6: Poses surrounding a sphere for the inverse kinematics calculations [29].

or capability to complete a given trajectory. But other tools exist to define and/or execute a given point-to-point move or follow a given trajectory. Thus, it is necessary to quickly review existing planners to determine if they include planning capabilities and/or how they handle conditions where the task cannot be executed. This review of current trajectory planners suggests we can improve the usability of existing trajectory planners instead of developing yet another solution in a crowded field. The focus will also be on planners that are compatible with ROS.

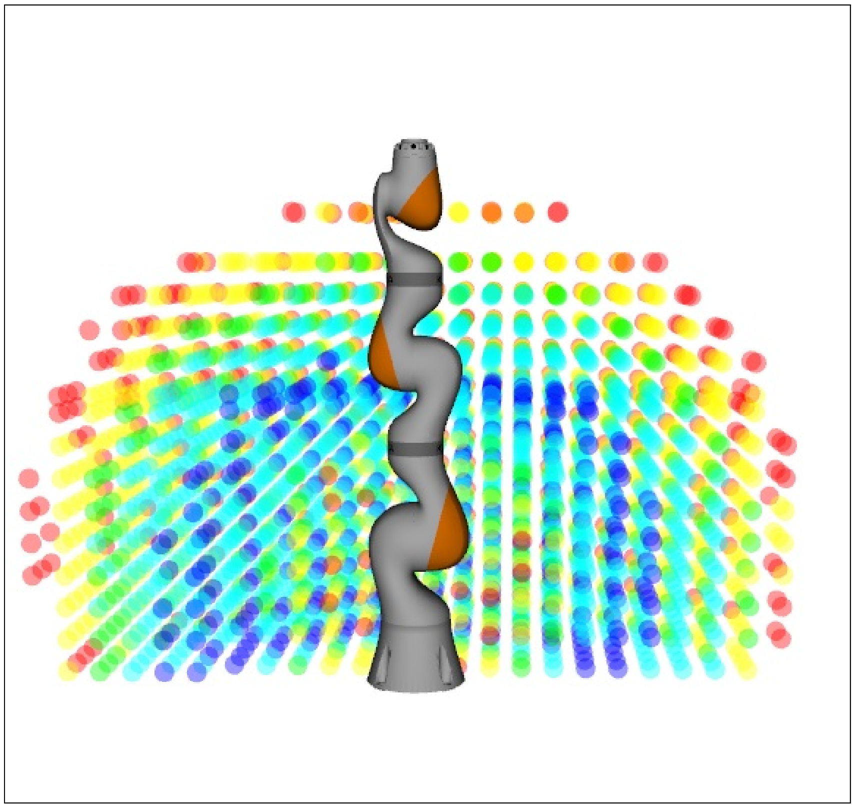
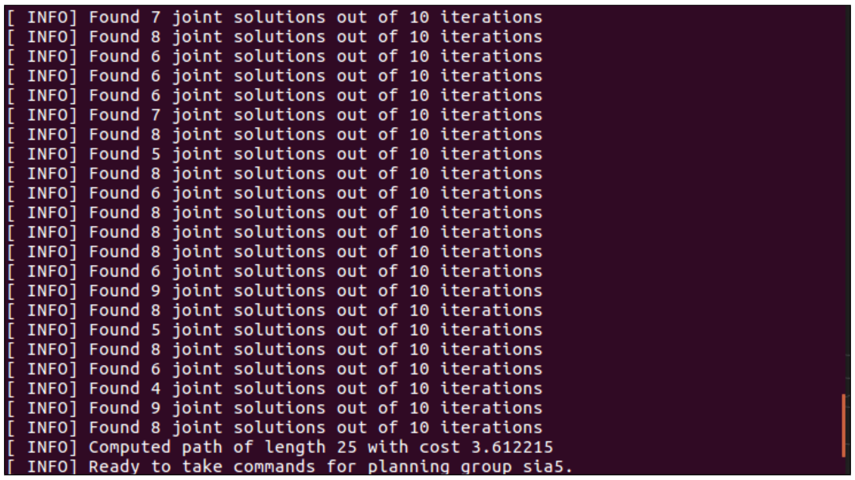


Figure 2.7: The Reuleaux reachability map for a LWR 7 DoF arm. Red indicates areas of low reachability and blue indicates areas of high reachability [29].

Existing ROS trajectory planners include Descartes [14], MoveIt! [41] and Trajopt [39]. Descartes uses a graph-based search to identify low-cost trajectories, particularly for tasks where the robot is redundant relative to the task output space. A cost-based minimization algorithm generates smooth trajectories that follow the given trajectory step time [14]. The user must input a 6 DoF task, but may reduce the number of DoF by adding freedom about an axis. A successful Descartes trajectory will show the user Figure 2.8

in the terminal and the robot will perform the trajectory.

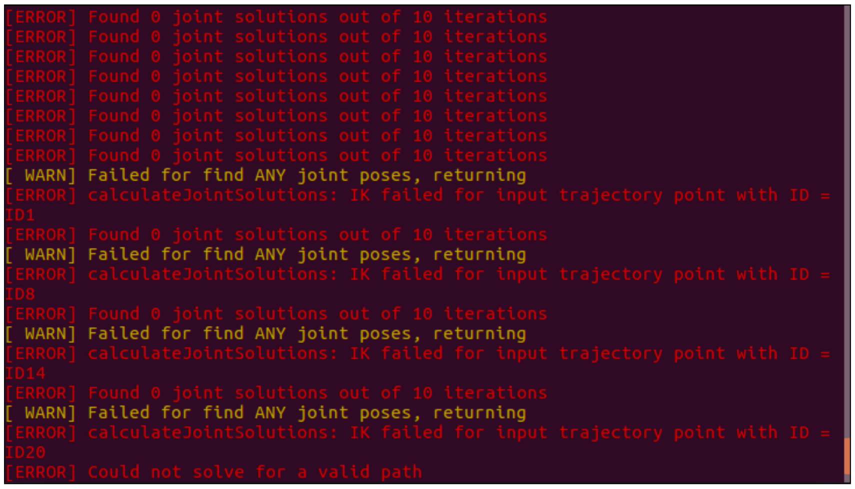


```
[ INFO] Found 7 joint solutions out of 10 iterations
[ INFO] Found 8 joint solutions out of 10 iterations
[ INFO] Found 6 joint solutions out of 10 iterations
[ INFO] Found 6 joint solutions out of 10 iterations
[ INFO] Found 6 joint solutions out of 10 iterations
[ INFO] Found 7 joint solutions out of 10 iterations
[ INFO] Found 8 joint solutions out of 10 iterations
[ INFO] Found 5 joint solutions out of 10 iterations
[ INFO] Found 8 joint solutions out of 10 iterations
[ INFO] Found 6 joint solutions out of 10 iterations
[ INFO] Found 8 joint solutions out of 10 iterations
[ INFO] Found 8 joint solutions out of 10 iterations
[ INFO] Found 8 joint solutions out of 10 iterations
[ INFO] Found 6 joint solutions out of 10 iterations
[ INFO] Found 9 joint solutions out of 10 iterations
[ INFO] Found 8 joint solutions out of 10 iterations
[ INFO] Found 5 joint solutions out of 10 iterations
[ INFO] Found 8 joint solutions out of 10 iterations
[ INFO] Found 6 joint solutions out of 10 iterations
[ INFO] Found 4 joint solutions out of 10 iterations
[ INFO] Found 9 joint solutions out of 10 iterations
[ INFO] Found 8 joint solutions out of 10 iterations
[ INFO] Computed path of length 25 with cost 3.612215
[ INFO] Ready to take commands for planning group sia5.
```

Figure 2.8: The output for a successful Descartes trajectory.

The Descartes trajectory planner was used in the NRG's multi-task glovebox efforts [31] and we observed its failures could be placed into two categories: the robot cannot reach one or more of the points, or the robot cannot move between the points with the desired velocity without exceeding defined joint velocity limits. In the event of a failure, the information reported to the developer is shown in Figure 2.9. Such failures are identified by executing the path and can only be corrected via trial-and-error adjustments to the task and/or robot's configuration.

MoveIt! plans joint and EEF positions to generate moves either to a point or along a constrained path, such as with a fixed EEF orientation, or a Cartesian path, which consists of moves in the x, y, or z directions [11]. It does not handle timed moves or trajectories except with post-processing where timestamps and velocity or acceleration values may be appended to a



```
[ERROR] Found 0 joint solutions out of 10 iterations
[ERROR] Found 0 joint solutions out of 10 iterations
[ERROR] Found 0 joint solutions out of 10 iterations
[ERROR] Found 0 joint solutions out of 10 iterations
[ERROR] Found 0 joint solutions out of 10 iterations
[ERROR] Found 0 joint solutions out of 10 iterations
[ERROR] Found 0 joint solutions out of 10 iterations
[ERROR] Found 0 joint solutions out of 10 iterations
[ WARN] Failed for find ANY joint poses, returning
[ERROR] calculateJointSolutions: IK failed for input trajectory point with ID =
ID1
[ERROR] Found 0 joint solutions out of 10 iterations
[ WARN] Failed for find ANY joint poses, returning
[ERROR] calculateJointSolutions: IK failed for input trajectory point with ID =
ID8
[ERROR] Found 0 joint solutions out of 10 iterations
[ WARN] Failed for find ANY joint poses, returning
[ERROR] calculateJointSolutions: IK failed for input trajectory point with ID =
ID14
[ERROR] Found 0 joint solutions out of 10 iterations
[ WARN] Failed for find ANY joint poses, returning
[ERROR] calculateJointSolutions: IK failed for input trajectory point with ID =
ID20
[ERROR] Could not solve for a valid path
```

Figure 2.9: The output for a failed Descartes trajectory.

previously planned path. Depending on the path, it may not be possible to achieve the desired trajectory. Moveit! fails when a move cannot be performed and does not provide any further information to the user unless a robot link is in collision.

Trajopt is a trajectory optimization method that can use sparse sampling to create collision-free paths [39]. Covariant Hamiltonian Optimization for Motion Planning (CHOMP) can improve sampled trajectories and optimize based on dynamic or task-based criteria [37]. For example, CHOMP can take a trajectory with collisions and find a collision-free trajectory while optimizing for joint velocities [11].

2.3 Performance Metrics

Researchers have previously developed a range of metrics to evaluate the performance of robotic manipulators for a variety of motivating reasons including redundancy resolution [19], system design [22], obstacle avoidance [27], as well as task optimization [25]. While not all the identified criteria would be relevant or beneficial in assisting an operator in part placement, several in the literature that could be useful are identified and reviewed in this section.

Several performance metrics can be used to evaluate a system's reachability during design or in real-time. Such metrics can also be used to define the redundancy resolution problem as an optimization problem. Metrics commonly defined in the literature include manipulability [46], dexterity, smoothness, transmissibility, system condition, etc. Manipulability, for example, defines a robot's ability to change the position or orientation of the EEF [46] given the current joint configuration. Yashikawa's manipulability metric is given in Equation 2.2.

$$w = \sqrt{det(\mathbf{J}(\mathbf{\Theta})\mathbf{J}^{\mathbf{T}}(\mathbf{\Theta}))} \tag{2.2}$$

Where $(\mathbf{J})(\mathbf{bf}(\mathbf{\Theta}))$ is the Jacobian of the robot with joint angles $bf(\Theta)$ and T symbolizes the transpose of a matrix.

Alternatively, one can use the manipulability ellipsoid [46] with principal axes $\theta_1 u_1$, $\theta_2 u_2$, ... , $\theta_m u_m$ where θ_i are the singular values and u_i are the column vectors of the singular value decomposition:

$$\mathbf{J} = \mathbf{U\Sigma V^T} \tag{2.3}$$

Dexterity can have many different physical meanings, from the manipulator's ability to reach a point from different orientations to the capability of the manipulator to perform fine, accurate motions [22]. Smoothness describes the motion of the EEF over a trajectory. Efficiency refers to minimizing some measure of efficiency such as kinetic energy [35]. Another common metric for dexterity or singularity avoidance (which are intrinsically related) is the condition number, commonly defined as:

$$c(\mathbf{J^T}) = \|\mathbf{J^T}\|\|\mathbf{J^{-T}}\| \tag{2.4}$$

It can also be used to measure the accuracy of an applied force [38]. Near singularities, the condition number approaches infinity.

There are also task-based quality measures, such as μ [25] among others, which measure the quality of grasping and thus are not relevant to this review.

2.4 Summary and Analysis

Most existing tools use a reachability map computed either with forward or inverse kinematics. These reachability maps can be extended to inverse reachability maps to search for ideal robot base placements. The sweeping method only describes the robot's reachable workspace.

The solutions listed above are not ideal for many applications. The maps can take hours to compute for a single robot configuration. In the case of Zacharias et al. [48], it took 12.4 hours to build all maps. As the maps describe the global robot workspace, they provide significant amounts of unnecessary information when looking at a single task. The maps are also robot-dependent; if the tool or robot changes, the maps must be recomputed. Dong and Trinkle attempt to solve this problem using their orientation-based reachability map [13] but can still account for only tool changes and not changes in the robot or robot configuration. This dependency is based on the need to solve the forward or inverse kinematics problem for each specific robot. Applied to the entire robot workspace, this is computationally expensive.

Furthermore, users need to use complex search algorithms to find where trajectories can be performed [47,49]. They provide too much information for the user to visually identify where the desired trajectory is possible.

Many of these solutions do not account for dynamic environments. Most perform collision checking after generating plausible trajectories [47,49], which may not be ideal for crowded environments where many or all possible trajectories may be in collision.

Thus there is a need to develop a task-specific tool for verifying and analyzing a robot's ability to complete a specific trajectory in a given environment, and provide guidance to the user in the event that the proposed trajectory is not possible. While planners such as MoveIt! provide a baseline capability to verify that selected points along a trajectory are reachable, it cannot efficiently validate the continuum of a trajectory's curves, nor can it advise on the robot's ability to complete the trajectory. While other planners including Descartes and TrajOpt, can inform the operator of a failure in executing its trajectory, none inform the user of how it fails or provide useful information on how to adjust the task space to eliminate the failure. Operators typically resort to trial and error in order to assure the task is completed. Such an approach is untenable if the environments are complex or the system must complete multiple tasks in a single space. Furthermore, these trajectory planners are unable to exploit the advantages that may exist if the task space is less than the 6-dimensional Cartesian Space. Finally, a variety of performance metrics were identified in the literature that could be calculated along the identified trajectory and communicated to the system developer to provide beneficial feedback and eliminate (or reduce) the frustrations associated with trial-and-error task placement.

The next chapter will use the lessons listed here to develop the Task-Trajectory Analysis Package (T-TAP) with three key components: reachability verification, trajectory verification, and presentation of relevant performance metrics calculated during the task execution.

Chapter 3

Development

Chapter 2 showed the need for a task-based trajectory analysis tool. Current reachability tools usually describe the robot's entire reachable workspace using a global reachability metric, not the accessibility of the task workspace. This chapter describes the mathematical constructs and algorithms of the Task-Trajectory Analysis Package (T-TAP) which include:

- reduction of task space (when $m < 6$) for both paths and trajectories,
- general reachability of the path,
- reachability of the trajectory based on:
  - adherence to joint position limits,
  - compliance with joint velocity limits based on time parameterization,
  - overall health of the trajectory given by the condition number or other selected metrics.

3.1 Requirements

In order to provide practical feedback to the user, T-TAP must accomplish certain requirements. T-TAP must be "quick," where quick is defined as taking minutes to run, not hours. Less than ten minutes for most paths is desirable. Additionally, as most current solutions do not account for obstacles, T-TAP must be able to account for obstacles including environments where obstacle placement may change.

Since many tasks exist with $m < 6$, the T-TAP should be able to work with any task where $m < 6$.

Finally, the T-TAP must provide visual feedback to the user. The T-TAP must identify path reachability and trajectory feasibilty with respect to robot design or performance constraints. For trajectories, T-TAP must show whether joint limits were met. Selected metrics must be visualized and should be extensible to include additional metrics.

These requirements are organized in Table 3.1.

Requirement	**Definition**
Quick Run Time	Run time of less than 10 minutes for most tasks
Obstacle Awareness	Accounts for obstacles and movement of obstacles
Task Generality	Works for tasks with $m \leq 6$
Path Reachability Visualization	Identifies where in the task the robot can and cannot reach and displays this to the user
Joint Limit Check	Ensures all joint limits are met
Metric Visualization	Visualizes the selected metrics in a way that is intuitive to the user
Metric Extensibility	Can include additional metrics

Table 3.1: Table of requirements and their definitions.

3.2 Path Reachability

The first step is to quickly verify whether the points along a trajectory are reachable by the robot. This step occurs first since all generated trajectories are guaranteed to fail if the robot cannot reach each individual point without any other constraints. While relatively simple to check using common methods, there are several secondary requirements that should be considered in the trajectory reachability step, including:

- speed of the trajectory with respect to joint velocity limits,
- warnings for attainable but poorly conditioned points,
- and visualization of failed point(s) to assist the user in re-positioning the task.

Zacharias et al. [47, 49] search the reachability map for areas where defined trajectories can be completed. This requires the prior calculation of the reachability map, which can take 4 hours or more, depending on the discretization size of the map. For a simple check of a trajectory, this time frame is not ideal. Additionally, fitting a discrete trajectory to a discretized map brings up other difficulties, namely the problem of fitting path points to the discretized workspace. This can lead to the aliasing issue described in Chapter 2.

With any changes to the robot configuration or environment, the reachability map must be recalculated. This limits hardware agnosticism. Using this knowledge, a desirable solution would check each pre-calculated path point instead of computing an entire map. Therefore, we use the robot model and an inverse kinematics solver to plan to each point from a starting position and determine whether that point is reachable from the starting position.

Alternatively, we could plan to one point, then plan to the next point from the previous point. We selected the current solution in favor of the alternate solution due to speed considerations. If we plan to one point and then want to use that point to plan to the next, we need the IK solver to find a plan, then to wait for the robot simulation to execute that plan and update its position. Whereas if we plan from the same starting point, we only need to wait for the IK solver to find a plan (or time out, as is discussed in Chapter 4) before we can immediately start planning to the next point without needing to wait for an updated robot state.

Paths are given in the form of waypoints: specified points along the path. These waypoints include x,y,z positions and quaternion orientations; the reasoning for this is discussed in Chapter 4. The specific IK solver utilized is left to the user's discretion. ROS-related options will be discussed in Chapter 4. To update the trajectory reachability with a change in obstacles or robot configuration, one must simply alter the robot model and rerun the code. The IK solver will account for the differences in the robot configuration and/or environment. Collision checking is performed inherently in the IK solver and is discussed in further detail in Chapter 4.

The code begins by marking every point in the path as blue to indicate which points have not been tested. This was done to allow the user to imme-

```
mark all points blue
for i=1:number of points,
    from starting position, plan to point i
        if successful,
            mark point as green
        else
            mark point as red
```

Code 3.1: Pseudocode for path reachability

diately visualize the path they sent to the robot and check to make sure that it is correct. It also allows the user to see the code progress. As each point is tested, the point is marked either red or green, red showing a failed point and green showing a successful point.

Code 3.1 demonstrates the algorithms used to determine trajectory reachability:

A* is one existing algorithm that plans to a point [16]. It searches along nodes to find the minimum cost path. Moreover, it searches the fewest amount of nodes needed to guarantee a lowest cost solution. Rapidly-exploring Random Trees (RRT) provide a way to quickly search the workspace for viable paths, without putting an emphasis on minimum cost paths [24]. These are extended for use in redundant systems [8]. Some planning methods use artificial potential fields to steer the planned path away from obstacles and towards a goal [44]. ROS has several built-in options, which will be discussed in the next chapter.

To account for tasks with space $m < 6$, we search near the value of the undefined variable, stopping when we can reach a point or when we exceed a set time limit. For example, if the rotation about z of the end effector is irrelevant, we randomize the value of that variable within ten degrees of the set value. For the x,y, and z variables, we allow ten centimeters of wiggle room. We search near the given value to try to find paths that are as continuous and smooth as possible. We use the randomized method as opposed to iterating through possible values as the iterative method quickly blows up when as m decreases below 6. Note that the user must provide a starting value for every variable.

One computationally expensive part of the path analysis code is the planning visualization, which can be toggled on and off. The code run time also increases dramatically when $m << 6$. Different, more intelligent methods for searching the task space are possible and do exist, but are outside the scope of this work. Potential solutions are described in 6 under Future Work. The code is tested for its run time in Chapter 5.

The path reachability portion of the T-TAP does not support velocity limits or singularities as they relate to traversing a trajectory, but does ensure joint limits are met for each waypoint.

3.3 Trajectory Analysis

Once all of the points on a proposed trajectory are reachable, we can analyze the trajectory. While higher order kinematic and dynamic analysis is possible, this work focuses on the first order kinematics governed by the following equation.

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{\Theta}} \tag{3.1}$$

Where $\mathbf{x}$ is a vector of Cartesian velocities, $\mathbf{J}$ is the Jacobian matrix, and $\dot{\mathbf{\Theta}}$ is a vector of joint velocities. The Jacobian is a function of the robot's current joint position. For small $\mathbf{\Delta\Theta}$, this equation can be written:

$$\mathbf{\Delta x} = \mathbf{J\Delta\Theta} \tag{3.2}$$

Equation 3.2 shows that a change in $\mathbf{\Theta}$ results in a change in $\mathbf{x}$ that is proportional to $\mathbf{J}$. If $\mathbf{J}$ is known, we can convert changes in $\mathbf{x}$ to changes in $\mathbf{\Theta}$. I.e. we can convert changes in the end-effector position to changes in the joint positions of the robot:

$$\mathbf{\Delta\Theta} = \mathbf{J^{-1}\Delta x} \tag{3.3}$$

Where $\mathbf{J^{-1}}$ is the inverse of the Jacobian matrix. We assume that $\mathbf{J}$ is known, as many programs can solve for $\mathbf{J}$ given the robot configuration. However, if $\mathbf{J}$ is not a square matrix (i.e. $m = n$), no inverse exists for $\mathbf{J}$. Therefore, we must use the pseudoinverse of $\mathbf{J}$. There are several different ways to calculate a pseudoinverse, the most common being the Moore-Penrose pseudoinverse:

$$\mathbf{J}^{+} = \mathbf{J}^{\mathbf{T}}(\mathbf{J}\mathbf{J}^{\mathbf{T}})^{\mathbf{-1}} \tag{3.4}$$

Equation 3.4 holds only for $m < n$. For $m > n$:

$$\mathbf{J}^{+} = (\mathbf{J}^{\mathbf{T}}\mathbf{J})^{\mathbf{-1}}\mathbf{J}^{\mathbf{T}} \tag{3.5}$$

In Equations 3.4 and 3.5, $\mathbf{J}^{\mathbf{T}}$ is the transpose of $\mathbf{J}$ and $\mathbf{J}^{+}$ is the pseudoinverse. These equations require that $\mathbf{J}$ is full rank [23].

Another method of obtaining the pseudoinverse is through Singular Value Decomposition (SVD) [4, 10]:

$$\mathbf{J}^{+} = \mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{\mathbf{T}} \tag{3.6}$$

Where the matrices $\mathbf{U}$ (a matrix of left singular vectors), $\mathbf{V}$ (a matrix of right singular vectors), and $\mathbf{\Sigma}$ (a diagonal matrix containing the singular values of $\mathbf{J}$) are derived from the decomposition of $\mathbf{J}$, given by:

$$\mathbf{J} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathbf{T}} \tag{3.7}$$

$\mathbf{\Sigma}^{+}$ is the pseudoinverse of $\mathbf{\Sigma}$. This pseudoinverse is created by taking the reciprocal of the nonzero values of $\mathbf{\Sigma}$.

We use the SVD formulation of the pseudoinverse, as it does not require a full-rank matrix $\mathbf{J}$ nor does it require changing the formula based on $m > n$ or

$m < n$. Given a robot starting position on the trajectory, we can use Equation 3.8 to calculate the joint positions for the next point on the trajectory. The trajectory point format is the same as the path point format described above, with the addition of a time step to describe the time it takes the robot to travel between waypoints.

$$\boldsymbol{\Delta\Theta} = \mathbf{J}^{+}\boldsymbol{\Delta}\mathbf{x} \tag{3.8}$$

For tasks with $m \leq 6$, we can remove the corresponding rows of the Jacobian matrix and $\mathbf{x}$ vector to account for reduced output space of the task.

Once we calculate the joint positions, we check to make sure that the newly calculated joint positions are within the joint position limits. Given a time step that dictates the time spent moving from point to point in the trajectory, we calculate the joint velocities and ensure that these are within the joint velocity limits. If a trajectory exceeds only velocity limits, the time step can be increased to create a successful trajectory. Once a robot exceeds joint limits, we exit the code as the rest of the trajectory is impossible.

Next, we ensure that the new joint position is not in collision with the environment or in self-collision. First, we obtain the collision scene from the robot environment. We check that each trajectory point is not in collision. When the robot collides with an object, we exit the code and identify the collision with a marker showing the collision point. This way, the user knows where the collision occurred and can adjust the trajectory as needed.

```
start with robot at trajectory point 1
for i=2:number of points,
    get Jacobian
    determine change in x from point i-1 to point i
    remove rows of the Jacobian/x as needed
    calculate pseudoinverse of Jacobian
    calculate change in joint angles
    if in joint position limits
        continue
    else
        break
    calculate joint velocities
    if in joint velocity limits
        continue
    else
        break
    if not in collision
        continue
    else
        show collision point
        break
    calculate metrics
create visualization of metrics
```

Code 3.2: Pseudocode for trajectory analysis

For the new trajectory point, we calculate the desired metrics and display them for the user.

This process is shown in the pseudocode below:

While other metrics can be added to the code, the selected metric for this effort was the condition number. We selected this metric to identify when the robot is running into a singularity, as the existence of singularities along a trajectory can cause the trajectory to fail or otherwise behave in an undesirable manner. Undesirable behavior includes erratic, jerky movements that occur as the robot can no longer move in the desired direction. To check for these singularities, we calculate the condition number.

There are many ways to calculate a condition number. One common method, identified in Chapter 2, is shown in Equation 3.9.

$$c(\mathbf{J^T}) = \|\mathbf{J^T}\|\|\mathbf{J^{-T}}\| \tag{3.9}$$

We use the SVD version of the condition number, which is the ratio of largest to smallest singular value [17]. The singular values are found in the matrix $\mathbf{\Sigma}$, as described above. The SVD condition number avoids the use of complicated matrix norm calculations and circumvents the choice of which norm to use, which may vary based on the application.

A large condition number implies that the system is close to a singularity. After a large condition number occurs, the rest of the trajectory is not viable and the program exits the loop of trajectory calculations. The selection of the cutoff for condition numbers is described in Chapter 4. Condition numbers are displayed according to three binned categories. Low condition numbers are green, condition numbers approaching the cutoff are yellow, and condition numbers exceeding the cutoff are red.

Once the calculations are complete for every waypoint in the trajectory, we show the user the final trajectory. We do not update the position of the robot during calculations as this was proven to be computationally expensive and increased the run time of the code.

With this framework for the T-TAP, we can now implement the T-TAP in ROS.

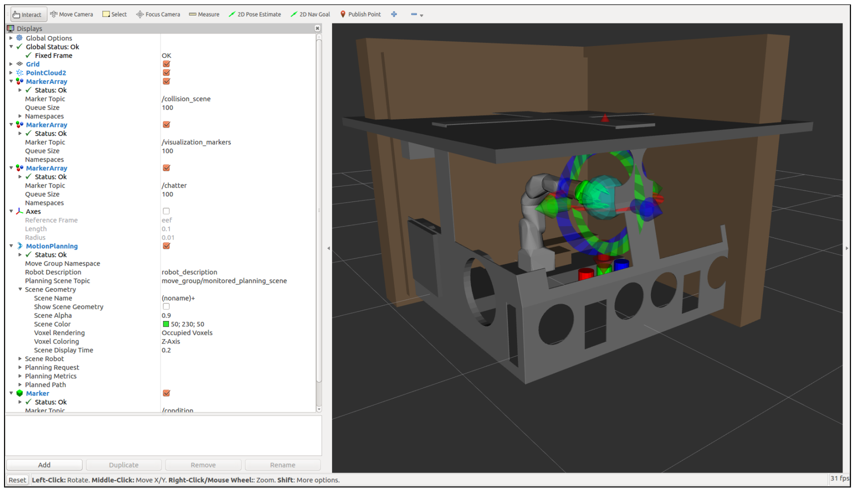
Chapter 4

Implementation

The chosen platform for the implementation of this project is the ROS [36]. ROS is an open-source platform for robot development. Many motion planners and trajectory planners, such as MoveIt! [41] and Descartes [14] already exist in ROS. T-TAP was coded in C++, one of the languages supported by ROS.

The system was designed to be hardware agnostic for all serial manipulators, although it was tested on the SIA5 (a redundant robot) and UR3 (a non-redundant robot). To use T-TAP, the user needs a visualization of the robot and collision scene in RViz, as shown in Figure 4.1, to fill out the config file, and to give the needed trajectory to either the path reachability or trajectory reachability program. T-TAP uses the same trajectory input file format as the Descartes tutorials, that is, an `Eigen::Affine3d` vector of x,y,z positions and quaternion orientations. This type of vector was selected because it can be converted into different message types as needed for different applications. For example, it can be converted to a `geometry_msgs::Pose` using the `tf::poseEigenToMsg` method.

The config file includes information such as the name of the robot move



Global Options	
Global Status: Ok	
Fixed Frame	OK
Grid	✓
PointCloud2	✓
MarkerArray	✓
Status: Ok	
Marker Topic	/collision_scene
Queue Size	100
Namespaces	
MarkerArray	✓
Status: Ok	
Marker Topic	/visualization_markers
Queue Size	100
Namespaces	
MarkerArray	✓
Status: Ok	
Marker Topic	/chatter
Queue Size	100
Namespaces	
Axes	☐
Reference Frame	eef
Length	0.1
Radius	0.01
MotionPlanning	✓
Status: Ok	
Move Group Namespace	
Robot Description	robot_description
Planning Scene Topic	move_group/monitored_planning_scene
Scene Geometry	
Scene Name	(noname)+
Show Scene Geometry	☐
Scene Alpha	0.9
Scene Color	🟩 50; 230; 50
Voxel Rendering	Occupied Voxels
Voxel Coloring	Z-Axis
Scene Display Time	0.2
Scene Robot	
Planning Request	
Planning Metrics	
Planned Path	
Marker	✓
Status: Ok	
Marker Topic	/condition

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options.

Figure 4.1: The SIA5 and glovebox in the RViz environment.

group, the fixed frame of the robot, the topic name for the joint trajectory command (needed for trajectory reachability only), the joint state topic, the time step (needed for trajectory reachability only), the task free variables, and the reduced task space time out time (needed for path reachability only).

4.1 Path Reachability

T-TAP uses the robot visualization and collision scene as the robot environment for the path reachability test. The robot visualization also informs the code as to which robot is being used for motion planning. First, T-TAP displays each untested point as blue in RViz using visualization markers. Then, it tests the reachability of each point using the built-in inverse kinematics of MoveIt!. IK algorithms are discussed in more detail below. T-TAP determines whether the robot can reach the given point from the current state of the robot. The inverse kinematics of MoveIt! ensures that the final pose is not in collision with any environment objects. Points that are reachable are marked as green and points that are not are marked red.

When $m < 6$, T-TAP reduces the task space by allowing for wiggle room around the reduced output space. T-TAP does this by randomizing the reduced variables to values near the starting value, stopping when a reachable point is found or when a set time limit is reached. The user must still send a starting guess for that variable. For example, if the initial guess for the EEF rotation about z is 0 degrees, but the output space does not include rotation about z, random values from -10 to 10 degrees will be tested.

Note that it is possible that the robot cannot reach a point from its current state but could reach that point from another pose. Therefore the selection of the starting pose of the robot is critical, but currently left to the user to decide. Selection of a "smart" starting configuration is outside the scope of this work but is discussed in the Future Work section of Chapter 6. As discussed in Chapter 5, the need for a better starting position occurred only once during testing.

Once the calculation is complete, the reachability of the trajectory is displayed. However, this algorithm is not perfect. Built-in planners have a timeout feature. If a solution to the inverse kinematics problem cannot be found in a certain set time limit, no more attempts to find a solution are made. In T-TAP, this is marked as an unreachable point. We leave the time limit in order to keep the runtime of T-TAP as quick as possible, though this means that sometimes reachable points will be marked as unreachable. The user may edit this time as they see necessary, although there is a standard time limit set in MoveIt! that is suitable for most trajectories. Unless the robot environment is very complex, such as the glovebox environment where there is an enclosed space with many collision objects, it will generally not be necessary for the user to edit the time limit. The user may use any kinematics solver they desire, although the KDL Kinematics plugin is native to ROS. If a closed-form solver exists for your robot, as it does for the UR3, the timeout feature will likely not be a problem.

A closed-form solution occurs when an equation (or equations) exist that fully describe the behavior of the robot and create a solution to the IK problem. Closed-form solutions exist only for robots with certain geometries [33]. For example, if a 6-DoF arm has three primsatic joints and three revolute joints, a closed-form solution exists. Robot arms with 7 DoF or more do not have a closed-form solution and require a numerical solution.

The KDL Kinematics plugin is a numerical solution that uses the linear least squares problem to solve for joint angles [20]. Given the forward kinematics equation as shown in Equation 4.1, the least squares problem minimizes Equation 4.2 to obtain a solution to Equation 4.1.

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{\Theta}} \tag{4.1}$$

$$\|\mathbf{\dot{x}} - \mathbf{J}\dot{\mathbf{\Theta}}\| \tag{4.2}$$

The value in Equation 4.2 is minimized using gradient descent, such as Newton-Raphson, which produces a initial guess for the next iteration using the IK equation shown in Equation 4.3. A pseudoinverse is used if necessary. With an initial guess for $\dot{\mathbf{\Theta}}$, this process is iterated until a solution is found within a specified tolerance or until a maximum number of iterations is reached.

$$\mathbf{\Delta\Theta} = \mathbf{J^{-1}\Delta x} \tag{4.3}$$

Another commonly used ROS kinematics solver is TRAC-IK [7]. TRAC-IK utilizes two different solvers: an updated KDL solver that improves the handling of local minima and SQP-SS, a sequential quadratic programming solver that better handles joint limits and minimizes sum of squares error. These solvers are run on different threads and the most quickly computed solution is used. To switch between the KDL Kinematics plugin and TRAC-IK, one downloads TRAC-IK and edits the kinematics.yaml file in the robot's MoveIt! package.

IKFast creates an analytical, closed-form solution for most robot arms ($DoF < 8$) [2]. However, creating this analytical solution is a lengthy and complex process. The user must download OpenRAVE, create a collada file, create the IKFast solution, and then create an IKFast plugin. Additionally, IKFast must be updated for changes in the robot configuration.

Overall, the native KDL Kinematics plugin is suitable for most applications, though TRAC-IK, IKFast, or a closed-form solution will likely perform better. If a closed-form solution exists, the user should utilize that solution. TRAC-IK is recommended over the use of the KDL Kinematics plugin, but not required. IKFast is not recommended due to its complex implementation unless an IKFast solution is already in use.

4.2 Trajectory Verification

Once the path reachability has been confirmed, the user can analyze trajectory reachability. The trajectory reachability portion of T-TAP pulls robot and collision scene information from RViz and MoveIt! T-TAP then operates in a series of steps listed in Figure 4.2 and described below.

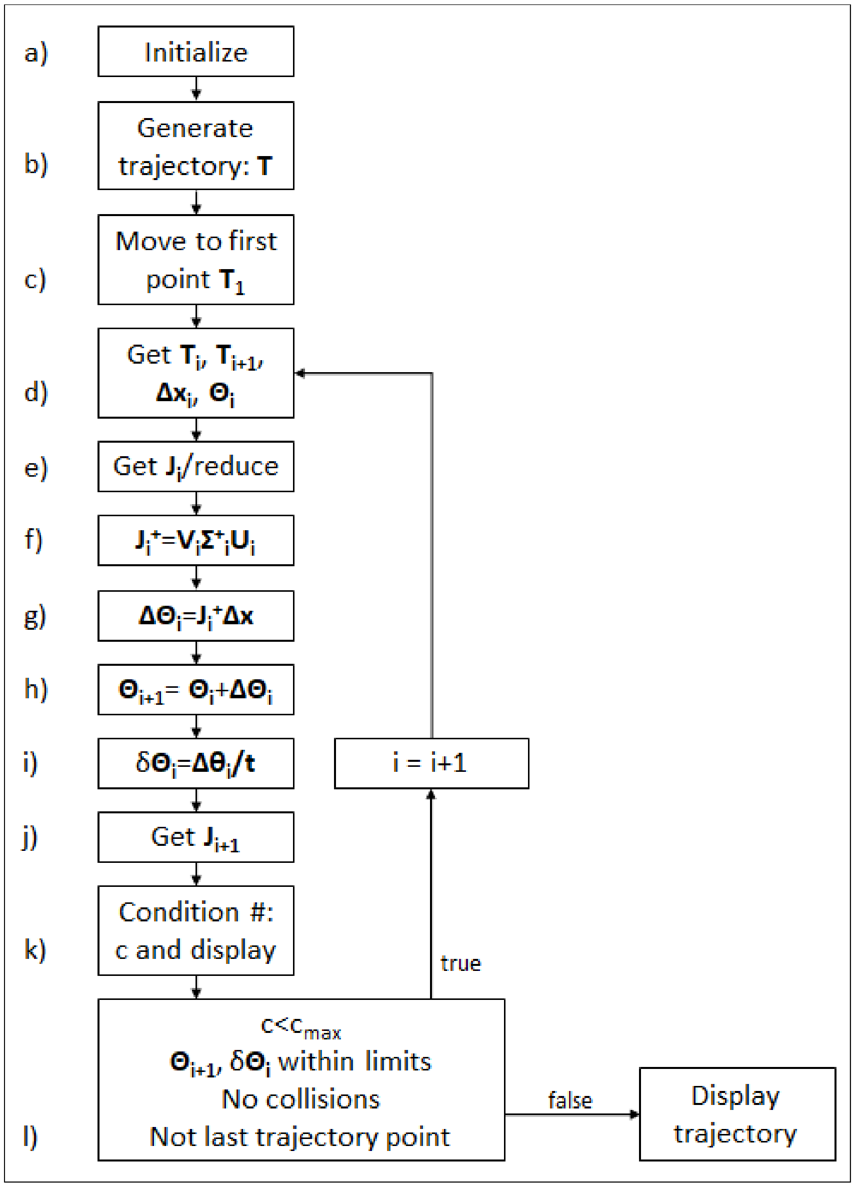

a)
Initialize
b)
Generate trajectory: $\mathbf{T}$
c)
Move to first point $\mathbf{T}_1$
d)
Get $\mathbf{T}_i$, $\mathbf{T}_{i+1}$, $\boldsymbol{\Delta x}_i$, $\boldsymbol{\Theta}_i$
e)
Get $\mathbf{J}_i$/reduce
f)
$\mathbf{J}_i^+=\mathbf{V}_i\boldsymbol{\Sigma}^+_i\mathbf{U}_i$
g)
$\boldsymbol{\Delta\Theta}_i=\mathbf{J}_i^+\boldsymbol{\Delta x}$
h)
$\boldsymbol{\Theta}_{i+1}= \boldsymbol{\Theta}_i+\boldsymbol{\Delta\Theta}_i$
i)
$\delta\boldsymbol{\Theta}_i=\boldsymbol{\Delta\theta}_i/\mathbf{t}$
i = i+1
j)
Get $\mathbf{J}_{i+1}$
k)
Condition #: c and display
true
l)
$c<c_{max}$
$\boldsymbol{\Theta}_{i+1}$, $\delta\boldsymbol{\Theta}_i$ within limits
No collisions
Not last trajectory point
false
Display trajectory

Figure 4.2: T-TAP steps.

a) To initialize T-TAP, we pull configuration data from the parameter server, initialize robot kinematics and collision scene, obtain robot information from MoveIt!, obtain collision scene from `/move_group/monitored_planning_scene` topic, set the planning scene to the collision scene (used for collision detection), obtain the current robot joint positions from the `/joint_states` topic published by the robot state or by a robot simulator, get the names of the joints from the kinematic state, set the kinematic state to the current joint positions, and initialize visualization markers.

b) Next, fill in the trajectory.

c) Convert the first trajectory point to a `geometry_msgs::Pose_Stamped` type and send the robot to the first point in the trajectory using MoveIt!.

d) T-TAP looks at the current position and the next trajectory position. We find the change in x vector, Δx, from the current robot position to the next trajectory point. We obtain the joint angles from the current robot position.

e) Obtain the Jacobian of the current kinematic state and remove the rows of the Jacobian and Δx according to the reduced task space of the trajectory.

f) Apply SVD to the Jacobian and calculate the pseudoinverse.

g) Calculate $\Delta\Theta$.

h) Add $\Delta\Theta$ to Θ.

i) Calculate $\dot{\Theta}$ using $\Delta\Theta$ and the time step.

j) Get the Jacobian of the new state.

k) Use the new Jacobian to calculate the condition number and display it.

l) Check to see if we should stay in the loop. If $c < c_{max}$, if joint limits are upheld, if there are no collisions, and if we are not at the last trajectory point, we increment i and restart the loop.

If the above conditions fail, we exit the loop and display whichever portion of the trajectory was completed.

T-TAP uses several classes which all have different applications in the code. The `planning_scene::PlanningScene` and `moveit::planning_inter-face::PlanningSceneInterface` classes are utilized for collision detection. After we load the already existing collision scene into the planning scene, we use the `checkCollision` method to check if the current robot state has any collisions, either with itself or with the environment.

After T-TAP determines if any collisions exist, the `collision_detect-ion::CollisionRequest` class uses `getCollisionMarkersFromContacts` to display the collisions. These are published to the `/interactive_robot_marray` topic.

The `robot_state::RobotState` class is used for obtaining the Jacobian using the `getJacobian method`. Since the Jacobian is an `Eigen::MatrixXd`, the built-in SVD function of `Eigen` can be used to calculate the pseudoinverse and the condition number. The `robot_state::RobotState` class is also used to check joint position limits and joint velocity limits using `satisfiesPosit-ionBounds` and `satisfiesVelocityBounds`.

For visualization, we use `visualization_msgs::MarkerArray` and `visualization_msgs::Marker` to display collision points and condition numbers. These are published using a ROS publisher to the `/condition` and `/interactive_robot_marray` topics, respectively.

These classes are part of MoveIt! and are used to limit what the user needs to download in addition to T-TAP. Since MoveIt! is paired with ROS, the user does not need to download any packages in addition to T-TAP.

As in the path reachability section of T-TAP, the starting position of the robot is critical. In this case, the robot plans to the first point, although the user can edit the code to use a pre-determined starting pose. Since the robot plans to the first point, the results of the trajectory reachability may change each time the user runs the code as the starting position may change. Intelligent selection of the starting pose is discussed in the future work section of Chapter 6.

The condition number cutoff was determined through testing during the development of T-TAP. For this case, the cutoff, c_{max}, is a condition number of 100. In testing, T-TAP would preform well until condition numbers of about 60. Then the condition number would jump (usually to 300+) and the behavior of the robot would become erratic. Therefore, we stop the robot when the condition number exceeds 100 as a conservative cutoff. The user may need to edit the condition number cutoff based on the task or robot.

4.3 Summary

This chapter describes the implementation of T-TAP into ROS. MoveIt! is heavily utilized to reduce the number of packages the user must download. The next chapter shows how the requirements listed in 3 were met by T-TAP.

Chapter 5

Demonstration

Chapter 3 determined the requirements needed for the proposed tool, T-TAP, summarized in Table 5.1.

Requirement	**Definition**
Quick Run Time	Run time of less than 10 minutes for most tasks
Obstacle Awareness	Accounts for obstacles and movement of obstacles
Task Generality	Works for tasks with $m \leq 6$
Path Reachability Visualization	Identifies where in the task the robot can and cannot reach and displays this to the user
Joint Limit Check	Ensures all joint limits are met
Metric Visualization	Visualizes the selected metrics in a way that is intuitive to the user
Metric Extensibility	Can include additional metrics

Table 5.1: Table of requirements and their definitions.

The demonstrations shown in this chapter include a surface finish task and an inspection task and were selected to exhibit the fulfillment of these requirements

5.1 SIA5 Path Demonstration

The path demonstration is shown for the SIA5 surface finish task described in Chapter 1.

5.1.1 Surface Finish Task

The surface finish task is taken from the multi-use workcell described by Paredes et al. [31]. The tool of the SIA5 robot follows a path along the surface of a hemispherical object. For this effort, placement of the task was arduous and repetitive, as there was no visual feedback to determine why the task location was failing. Instead, the task location was iteratively displaced in small amounts to try to find a reachable position near the chuck. When that failed, random positions for the task location were attempted until a reachable position was found. The final position of the task was selected not because it was optimal, but because it was difficult to find a reachable position.

Figure 5.1 shows the surface finish path over the chuck, which was the originally desired position for the task. At this location, the robot simulation could not reach the furthest points on the hemisphere. For example, it took hours of trial and error to conclude that the robot could not reach the surface finish task when it was placed over the chuck. Meanwhile, it took the T-TAP code 5 minutes and 35 seconds to identify that the robot could not reach all the points as well as what specific points in the trajectory were not reachable.

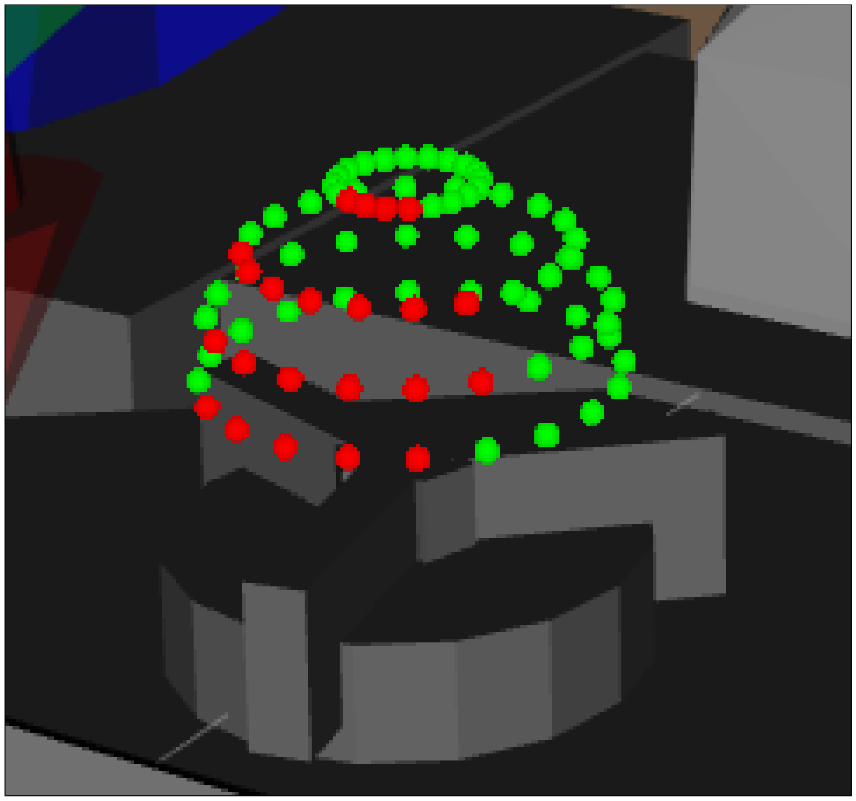


Figure 5.1: The desired surface finish task location. Red dots are unreachable by the robot and green dots are reachable.

Moving the path closer to the base of the robot by seven centimeters made the path fully executable, as can be seen in Figure 5.2. Therefore the chuck should be moved closer to the base of the robot if possible. The T-TAP displayed this result in 6 minutes and 07 seconds.

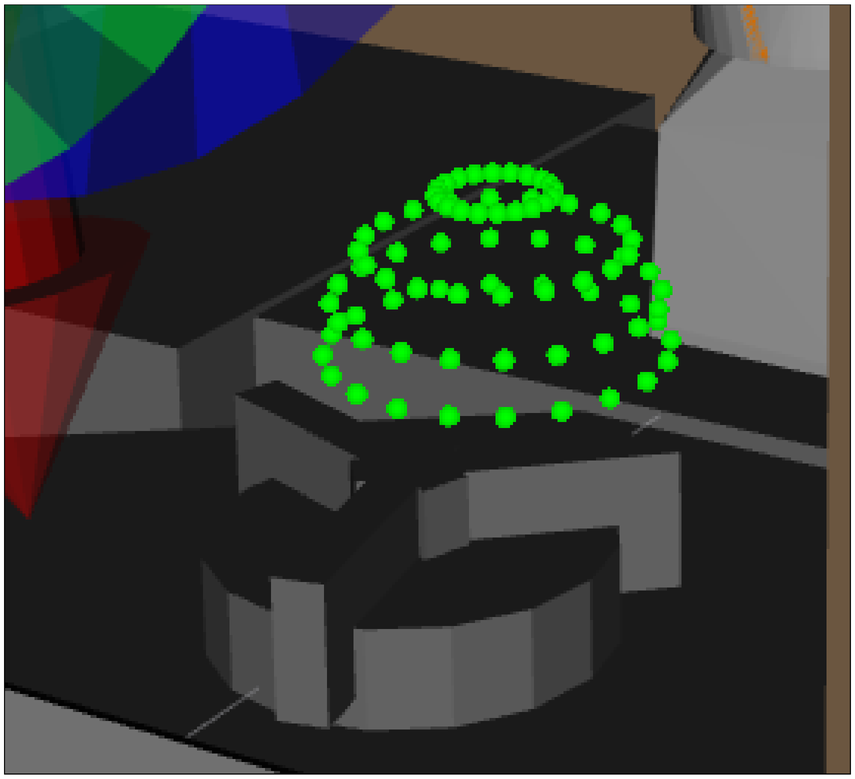


Figure 5.2: The suggested surface finish task location.

Next, we analyzed the path reachability by reducing the output space of the task from 6 to 3. The task was placed in its original location shown in Figure 5.1. We loosened all orientation constraints on the task to see if this improved task reachability. Figure 5.3 shows the result of this change. The path overall had more reachable points, although some points were still problematic. The time out period for the reduced output space reachability was set to 30 seconds to allow for multiple retries with the reduced task space. This run of the code took 10 minutes and 34 seconds, which is slightly higher the subjectively selected run time of less than 10 minutes. However, it is possible that the code could be optimized to reduce the run time.

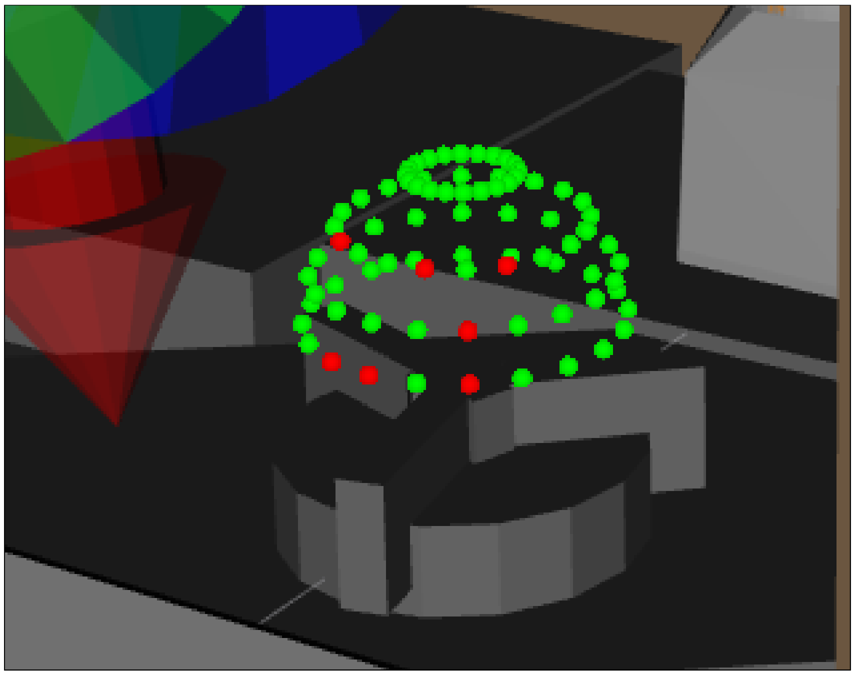


Figure 5.3: Updated reachability with loose EEF orientation constraints.

Finally, we move the task position down three centimeters so that the task intentionally collides with the chuck to ensure that the obstacle awareness is functional, which is confirmed in Figure 5.4. This run took 5 minutes and 30 seconds.

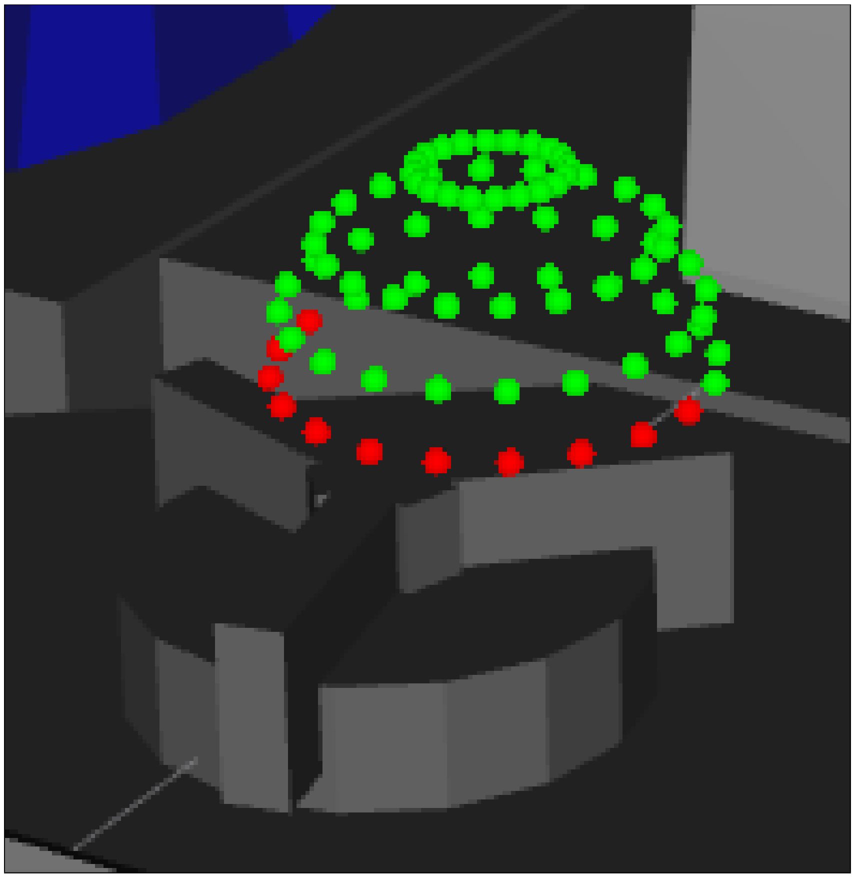


Figure 5.4: Reachability with collision.

This demonstration showed T-TAP's quick run time, obstacle awareness, task generality, and path reachability capabilities. Table 5.2 shows which requirements were met by this demonstration.

5.2 SIA5 Trajectory Validation using T-TAP

Next, we input the reachable path into the trajectory reachability code. The trajectory was reachable, as shown in Figure 5.5, with a time step of 1

Requirement	**Definition**	
Quick Run Time	Run time of less than 10 minutes for most tasks	✓
Obstacle Awareness	Accounts for obstacles and movement of obstacles	✓
Task Generality	Works for tasks with $m \leq 6$	✓
Path Reachability Visualization	Identifies where in the task the robot can and cannot reach and displays this to the user	✓
Joint Limit Check	Ensures all joint limits are met	
Metric Visualization	Visualizes the selected metrics in a way that is intuitive to the user	
Metric Extensibility	Can include additional metrics	

Table 5.2: Table of requirements and their definitions.

second to travel between waypoints. This time step was selected as it was sufficiently slow to avoid joint velocity limit issues, but in a real task would be driven by the task speed. This run of the code took 3 minutes and 24 seconds. In this case, the condition number is displayed to the user based on the cutoff values discussed in Chapter 4. It is simple to add a new metric. The metric calculation can be added into the loop and a new marker publisher can be created.

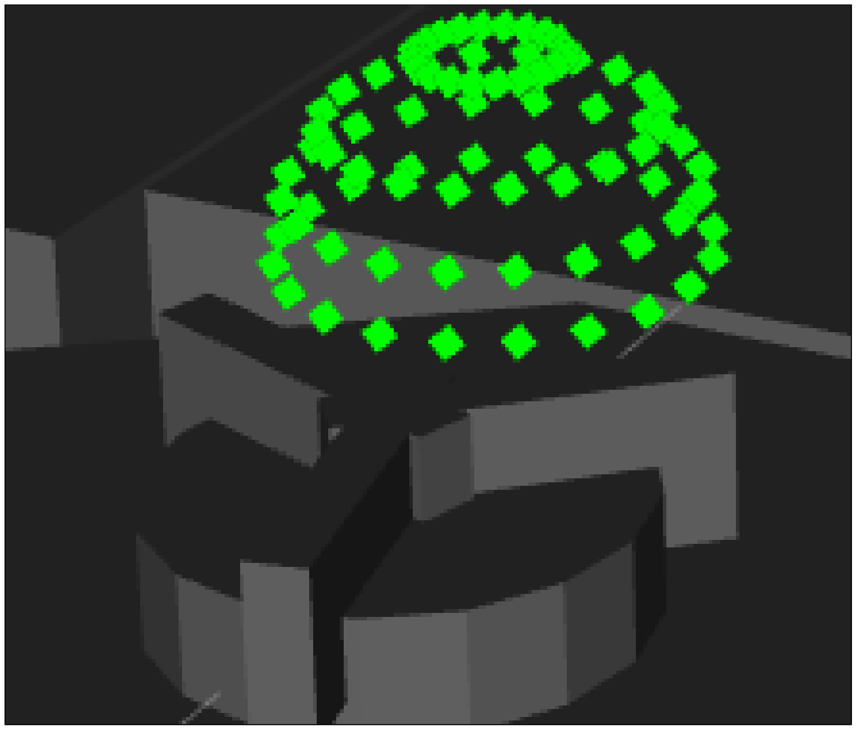


Figure 5.5: Trajectory reachability.

To check for obstacle awareness, we again move the trajectory to a spot where it will collide with the chuck. The results of this test are shown in Figure 5.6. This took 35 seconds as the first waypoint was in collision.

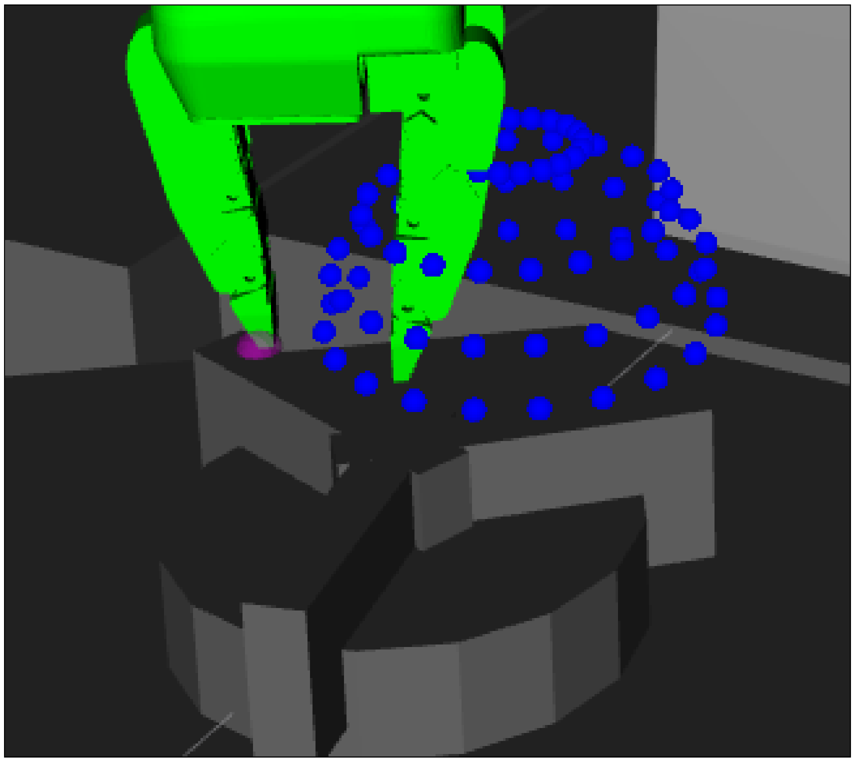


Figure 5.6: Trajectory collision with the collision point marked purple.

Next, we want to see if reducing the output space of the task will make the original task location achievable. To do this, we remove all orientation constraints on the task. This run of the code took 3 minutes and 38 seconds. The results are shown in Figures 5.7 and 5.8.

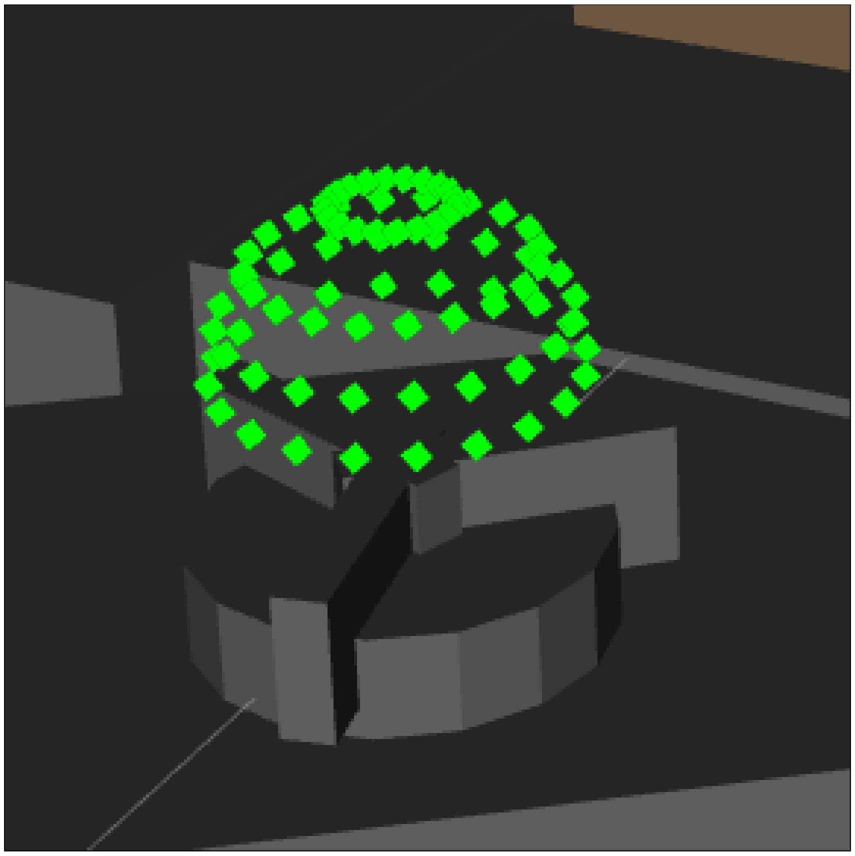


Figure 5.7: Reachability of the orientation-relaxed trajectory.

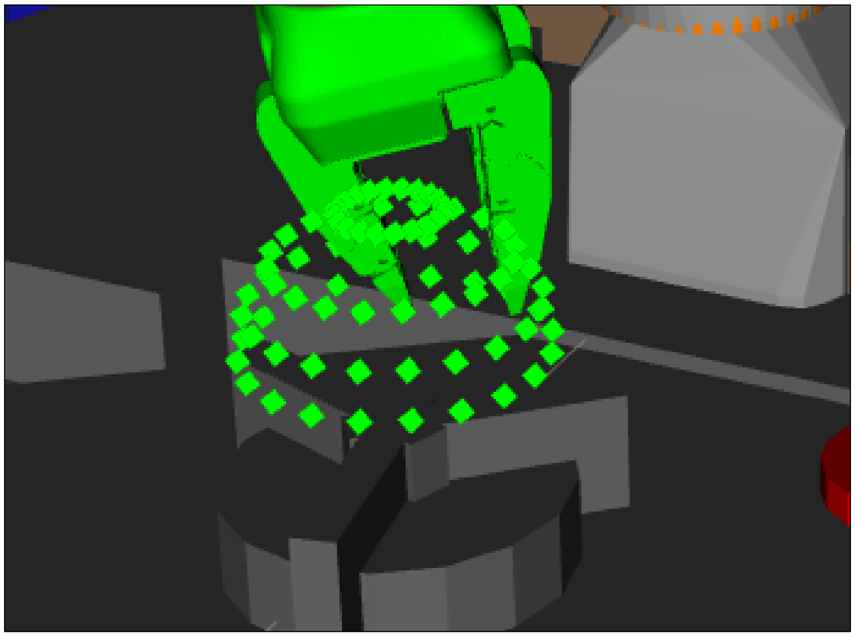
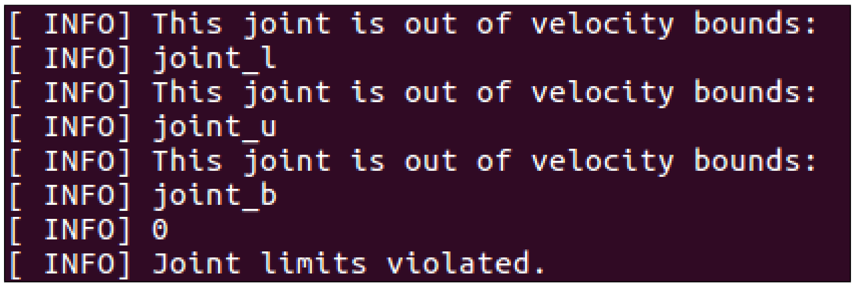


Figure 5.8: Robot executing the orientation-relaxed trajectory.

To check the joint limit capabilities of the T-TAP, we set the time step to 0.1 seconds, which should breach some velocity limits. The terminal printout of this test is shown in Figure 5.9. It shows the user which joints are in violation and which type of violation is occurring. As the joint velocity limit failed at the first point, this step took only 33 seconds.


```
[ INFO] This joint is out of velocity bounds:
[ INFO] joint_l
[ INFO] This joint is out of velocity bounds:
[ INFO] joint_u
[ INFO] This joint is out of velocity bounds:
[ INFO] joint_b
[ INFO] 0
[ INFO] Joint limits violated.
```
Figure 5.9: Trajectory reachability with joint limits exceeded.

All requirements except for path reachability were met by this demonstration. This is summarized in Table 5.3.

Requirement	**Definition**	
Quick Run Time	Run time of less than 10 minutes for most tasks	✓
Obstacle Awareness	Accounts for obstacles and movement of obstacles	✓
Task Generality	Works for tasks with $m \leq 6$	✓
Path Reachability Visualization	Identifies where in the task the robot can and cannot reach and displays this to the user	
Joint Limit Check	Ensures all joint limits are met	✓
Metric Visualization	Visualizes the selected metrics in a way that is intuitive to the user	✓
Metric Extensibility	Can include additional metrics	✓

Table 5.3: Table of requirements and their definitions.

5.3 UR3 Path Demonstration

To show that the T-TAP works on non-redundant robot arms, we ran a inspection task demonstration on the Universal Robotics UR3. The inspection task is inspired by a previous demonstration in the NRG. LANL had a need to inspect detonator caps. The caps are placed in a tray, as shown in Figure 5.10, and a camera attached to a UR3 moves over the detonators to determine whether the caps are face up or not. The user needs to place the tray in a location where the camera can view each of the caps.

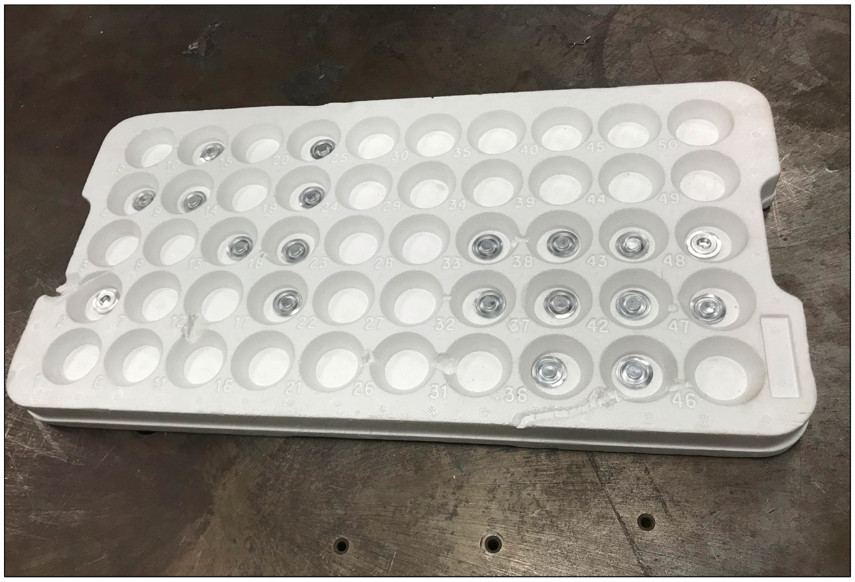


Figure 5.10: Mock detonator caps in a tray.

Figure 5.11 shows a reachable tray position. This run took 27 seconds and overall it took less than 15 minutes to find a position where all of the detonators could be reached.

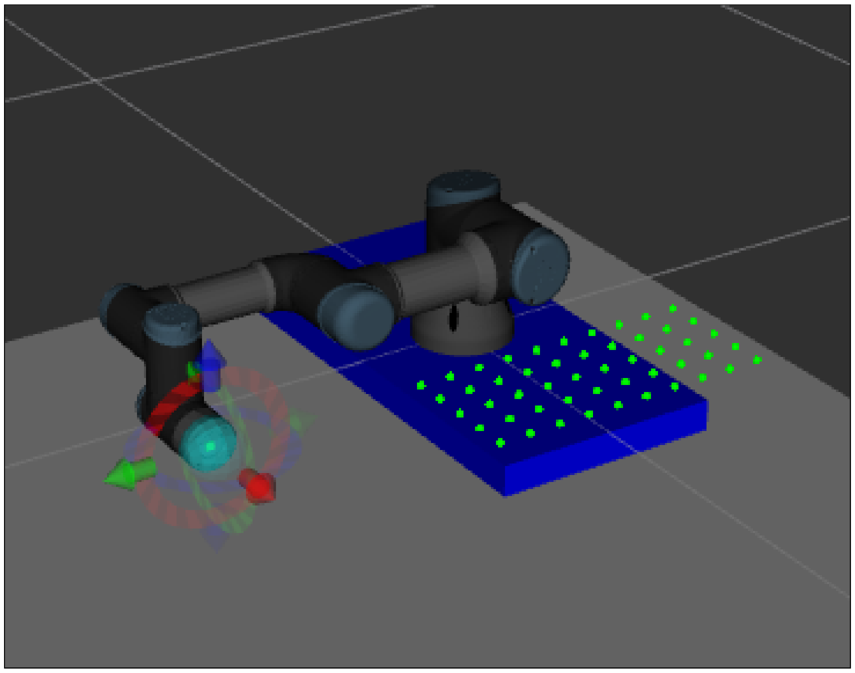


Figure 5.11: A reachable tray position.

5.4 UR3 Trajectory Demonstration

Next we analyze the inspection trajectory. Figure 5.12 shows the results. This starting position was a poor choice as the robot ran into joint limits, as shown in Figure 5.13. This took 32 seconds to run. The author recommends selecting another starting position to see if that improves the results or removing the position constraints on the wrist 3 axis. Aside from this test, starting pose was not a major factor in the success of a trajectory. However, the code is shown to be functional on a UR3.

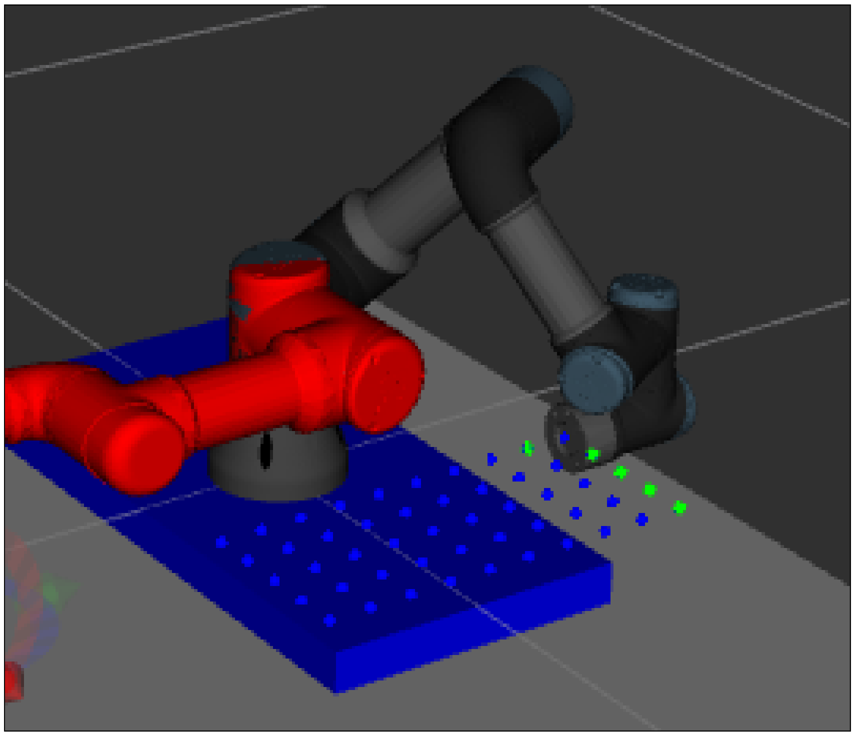
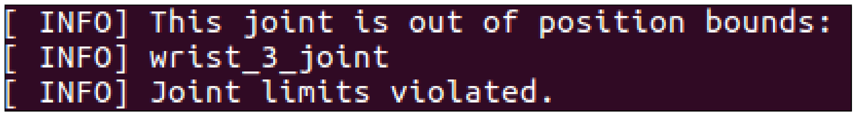


Figure 5.12: The inspection trajectory analysis.



```
[ INFO] This joint is out of position bounds:
[ INFO] wrist_3_joint
[ INFO] Joint limits violated.
```

Figure 5.13: The inspection trajectory analysis error code.

5.5 Summary

Path reachability and trajectory verification were shown on both the SIA5 and UR3 robot arms. T-TAP was tested to ensure it met all of the requirements. All failure conditions were created and tested such as: exceeding joint limits and collisions. The effects of a reduced task output space were studied and shown to improve the likelihood of a successful trajectory. These tasks may seem simple but demonstrate the needed capabilities of T-TAP. More complex tasks are possible but would test the capabilities of the robot and not T-TAP.

Chapter 6

Conclusions and Future Work

6.1 Summary

Selecting a location in the workspace where a robot can complete a task is a well-defined problem in the literature. In Chapter 2, we find that existing solutions to this problem are (as noted in Chapter 1):

- Computationally expensive, and cannot provide immediate feedback to a system developer,
- Do not include the obstacles in their analysis,
- Do not account for the task spaces where $m < 6$, and
- Typically focus on reachability which can validate paths, but not trajectories.

This work describes the creation of a new solution, T-TAP, that has the following features:

- generalized to work with any task spaces where $m \leq 6$;
- alerting the user to which designated points along the trajectory's path are reachable (or not);

- for reachable paths, informing the user if the trajectory will fail and and for what reason(s) (joint limit violation, velocity limit violation, collision, singularity, etc.);
- visually reporting these results to the user; and
- visualizing calculated performance metrics over the trajectory and simplifying the inclusion of new metrics.

This system was designed to reduce the burden on the user when adding new tasks for a robot to perform, thus lowering the barriers to automation.

6.2 Future Work

There are several areas of future work that were determined to be outside of the scope of this project. First, the selection of a starting robot pose for the trajectory is significant. A poorly chosen starting pose could easily cause the trajectory to fail, i.e. if the starting pose is close to joint position limits. An updated version of T-TAP could include methods for choosing the starting pose, such as selecting a pose where the joint positions are as close to the middle of the joint range as possible. Otherwise, the randomized starting poses could be tested and the one that creates the best trajectory, using the condition number or other metrics as a guide, would be selected as the starting position.

T-TAP path analysis removes constraints on a task with $m < 6$ by searching around the given point for a successful, new waypoint. This does not capture scenarios where the successful point is just out of reach of our widened search area. Another method for reducing the task output space for the path code may be beneficial, depending on the user's needs.

Due to the inverse kinematics formulation, Equation 6.1 holds only for small $\mathbf{\Delta\Theta}$. This limits the step size between waypoints of the trajectory. More research is needed to determine an acceptable maximum step size for a trajectory. Additionally, the step size could be analyzed against performance efficiency and the minimum amount of points required to confirm the reachability of the trajectory.

$$\mathbf{\Delta\Theta} = \mathbf{J}^{+}\mathbf{\Delta x} \tag{6.1}$$

Additionally, T-TAP could feature adaptive step sizing, where the step size decreases as the trajectory becomes problematic, for example when the robot is approaching a singularity. Another potential feature is the ability to dynamically adjust points in RViz to avoid problematic areas. Some tasks also have probabilistic uncertainty for the point location, which is currently not handled in this version of T-TAP.

While T-TAP accounts for obstacles when checking a path or trajectory, it does not intelligently avoid obstacles. Utilizing potential fields or some other method for object avoidance would improve the performance of T-TAP.

T-TAP was created for stationary robot arms, but it could be updated to work on mobile manipulators and assist in the selection of a base position for the mobile manipulator to perform a task.

6.3 Conclusion

T-TAP helps the user visualize path and trajectory reachability. It does this in two steps; first analyzing reachability of the path and then validating the desired trajectory given timing information. Chapter 1 provided a motivating example and a shared vocabulary for this effort. The motivating example was the multiuse workcell, shown in Figure 6.1, where selecting the placement of the varied tasks was problematic.

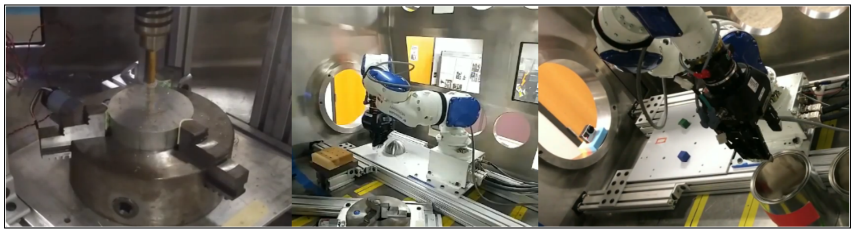


Figure 6.1: The three tasks of the multi-use workcell: drilling, sorting, and surface finishing [1].

Chapter 2 examined the research in this area and determined that no existing solution was sufficient for the problem. Existing solutions such as Reuleaux produce reachability maps, such as the one shown in Figure 6.2, that do not give the user easily understood, task-dependent reachability. Additionally, solutions such as Reuleaux take hours to compute.

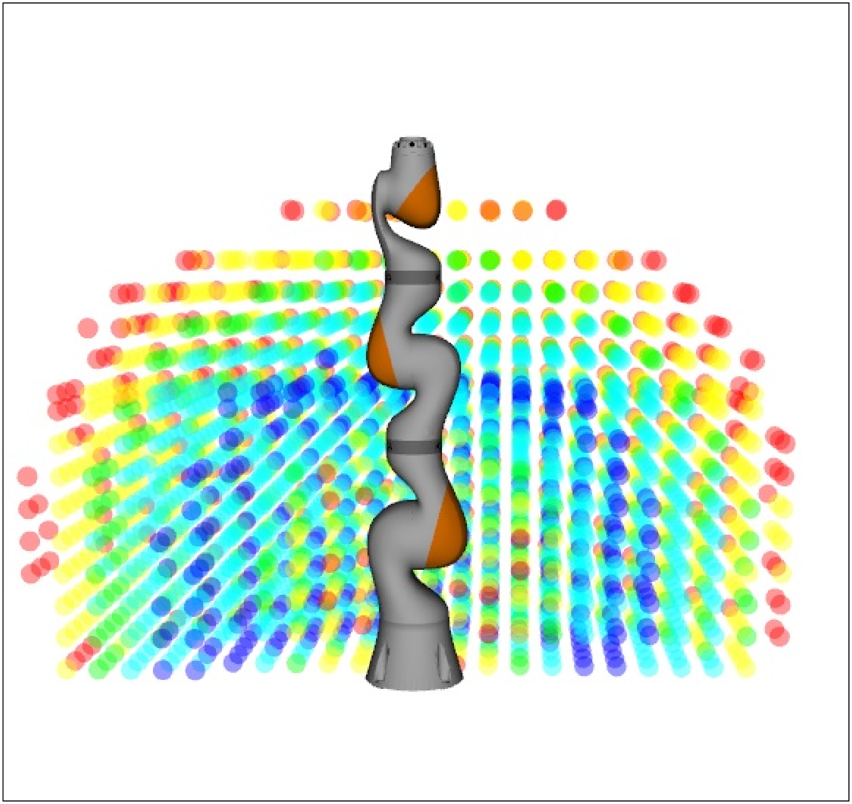


Figure 6.2: The Reuleaux reachability map for a LWR 7 DoF arm. Red indicates areas of low reachability and blue indicates areas of high reachability [29].

Chapter 3 detailed the invention of T-TAP and created a list of requirements for T-TAP: quick run time, obstacle awareness, task generality, path reachability verification, joint limit checks, metric visualization, and metric extensibility. These requirements are listed in Table 6.1.

Requirement	**Definition**
Quick Run Time	Run time of less than 10 minutes for most tasks
Obstacle Awareness	Accounts for obstacles and movement of obstacles
Task Generality	Works for tasks with $m \leq 6$
Path Reachability Visualization	Identifies where in the task the robot can and cannot reach and displays this to the user
Joint Limit Check	Ensures all joint limits are met
Metric Visualization	Visualizes the selected metrics in a way that is intuitive to the user
Metric Extensibility	Can include additional metrics

Table 6.1: Table of requirements and their definitions.

After testing each point in a path for reachability, T-TAP operates around Equation 6.2 to calculate the change in joint angles for a change in $\mathbf{x}$. T-TAP steps through the trajectory to verify whether or not the trajectory is viable.

$$\boldsymbol{\Delta\Theta} = \mathbf{J}^{+}\boldsymbol{\Delta}\mathbf{x} \tag{6.2}$$

Chapter 4 showed the creation of T-TAP in ROS, which follows the steps shown in Figure 6.3.

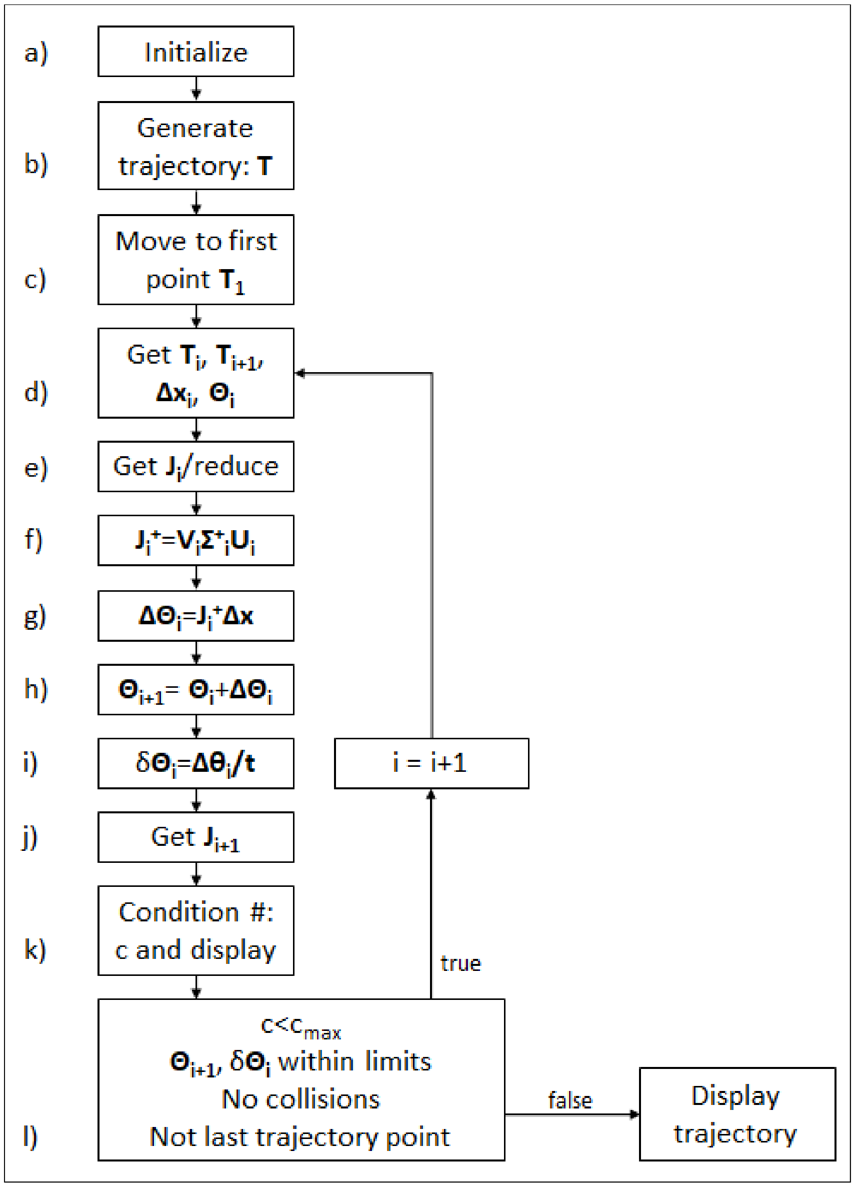

a)
Initialize
b)
Generate trajectory: $\mathbf{T}$
c)
Move to first point $\mathbf{T}_1$
d)
Get $\mathbf{T}_i$, $\mathbf{T}_{i+1}$, $\mathbf{\Delta x}_i$, $\mathbf{\Theta}_i$
e)
Get $\mathbf{J}_i$/reduce
f)
$\mathbf{J}_i^+ = \mathbf{V}_i \mathbf{\Sigma}^+_i \mathbf{U}_i$
g)
$\mathbf{\Delta\Theta}_i = \mathbf{J}_i^+ \mathbf{\Delta x}$
h)
$\mathbf{\Theta}_{i+1} = \mathbf{\Theta}_i + \mathbf{\Delta\Theta}_i$
i)
$\delta\mathbf{\Theta}_i = \mathbf{\Delta\theta}_i / \mathbf{t}$
i = i+1
j)
Get $\mathbf{J}_{i+1}$
k)
Condition #: c and display
true
l)
$c < c_{max}$
$\mathbf{\Theta}_{i+1}$, $\delta\mathbf{\Theta}_i$ within limits
No collisions
Not last trajectory point
false
Display trajectory

Figure 6.3: T-TAP steps.

Chapter 5 proved that all of the requirements were met with demonstrations that displayed the capabilities of T-TAP, such as that shown in Figure

6.4.

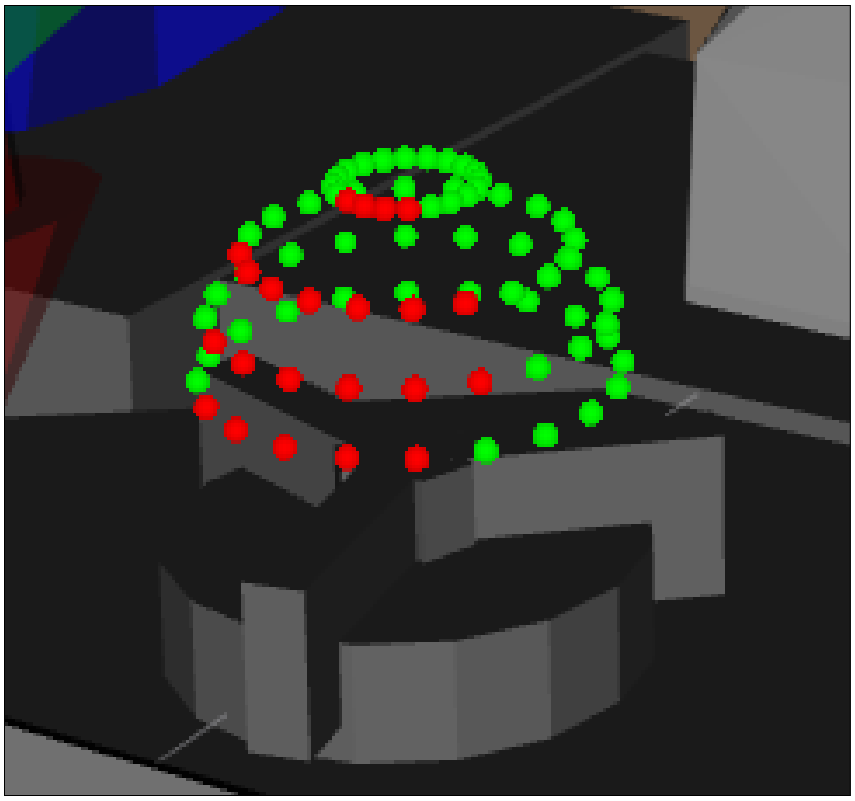


Figure 6.4: The results of the path reachability analysis for a surface finish task. Red dots are unreachable by the robot and green dots are reachable.

Bibliography

[1] Demonstration of Multiple Manufacturing Tasks in a Glovebox. `"https://www.youtube.com/watch?v=HqEOeY3gKrI"`. Accessed: 2019-05-06.

[2] IKFast Kinematics Solver. `"http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/ikfast/ikfast_tutorial.html"`. Accessed: 2019-04-15.

[3] Universal Robots. `"https://s3-eu-west-1.amazonaws.com/ur-support-site/50241/UR3_User_Manual_en_US.pdf"`. Accessed: 2019-05-06.

[4] Hervé Abdi. Singular value decomposition (svd) and generalized singular value decomposition. *Encyclopedia of measurement and statistics*, pages 907–912, 2007.

[5] D Alciatore and C Ng. Determining manipulator workspace boundaries using the monte carlo method and least squares segmentation. *ASME Robotics: Kinematics, Dynamics and Controls*, 72:141–146, 1994.

[6] Adrien Baranes and Pierre-Yves Oudeyer. Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems*, 61(1):49–73, 2013.

[7] Patrick Beeson and Barrett Ames. Trac-ik: An open-source library for improved solving of generic inverse kinematics. In *2015 IEEE-RAS 15th*

International Conference on Humanoid Robots (Humanoids), pages 928–935. IEEE, 2015.

[8] Dominik Bertram, James Kuffner, Ruediger Dillmann, and Tamim Asfour. An integrated approach to inverse kinematics and path planning for redundant manipulators. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1874–1879. IEEE, 2006.

[9] Felix Burget and Maren Bennewitz. Stance selection for humanoid grasping tasks by inverse reachability maps. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 5669–5674. IEEE, 2015.

[10] Samuel R Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation*, 17(1-19):16, 2004.

[11] Sachin Chitta, Dave Hershberger, Acorn Pooley, Dave Coleman, Michael Gorner, Francisco Suarez, and Mike Lautman. MoveIt! Tutorials. `"http://docs.ros.org/kinetic/api/moveit_tutorials/html/"`. Accessed: 2018-07-24.

[12] Rosen Diankov. Automated construction of robotic manipulation programs. 2010.

[13] Jun Dong and Jeffrey C Trinkle. Orientation-based reachability map for robot base placement. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 1488–1493. IEEE, 2015.

[14] Shaun Edwards. The descartes planning library for semi-constrained cartesian trajectories, October 2015.

[15] James A Hansen, KC Gupta, and SMK Kazerounian. Generation and evaluation of the workspace of a manipulator. *The International Journal of Robotics Research*, 2(3):22–31, 1983.

[16] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[17] Dan Kalman. A singularly valuable decomposition: the svd of a matrix. *The college mathematics journal*, 27(1):2–23, 1996.

[18] Chetan Kapoor, Murat Cetin, Mitch Pryor, Chris Cocca, Troy Harden, and Delbert Tesar. A software architecture for multi-criteria decision making for advanced robotics. In *Intelligent Control (ISIC), 1998. Held jointly with IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA), Intelligent Systems and Semiotics (ISAS), Proceedings*, pages 525–530. IEEE, 1998.

[19] Kazem Kazerounian and Zhaoyu Wang. Global versus local optimization in redundancy resolution of robotic manipulators. *The International*

Journal of Robotics Research, 7(5):3–12, 1988.

[20] Karan Khokar, Patrick Beeson, and Rob Burridge. Implementation of kdl inverse kinematics routine on the atlas humanoid robot. *Procedia Computer Science*, 46:1441–1448, 2015.

[21] Stuart Kieffer, Vassilios Morellas, and Max Donath. Neural network learning of the inverse kinematic relationships for a robot arm. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 2418–2425. IEEE, 1991.

[22] Charles A Klein and Bruce E Blaho. Dexterity measures for the design and control of kinematically redundant manipulators. *The International Journal of Robotics Research*, 6(2):72–83, 1987.

[23] Charles A Klein and Ching-Hsiang Huang. Review of pseudoinverse control for use with kinematically redundant manipulators. *IEEE Transactions on Systems, Man, and Cybernetics*, (2):245–250, 1983.

[24] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[25] Zexiang Li and S Shankar Sastry. Task-oriented optimal grasping by multifingered robot hands. *IEEE Journal on Robotics and Automation*, 4(1):32–44, 1988.

[26] Z. C. Lia and C. H. Menq. The dexterous workspace of simple manipulators. *IEEE Journal on Robotics and Automation*, 4(1):99–103, Feb 1988.

[27] Anthony A Maciejewski and Charles A Klein. Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments. *The international journal of robotics research*, 4(3):109–117, 1985.

[28] Anthony A Maciejewski and Charles A Klein. Numerical filtering for the operation of robotic manipulators through kinematically singular configurations. *Journal of Robotic Systems*, 5(6):527–552, 1988.

[29] A. Makhal and A. K. Goins. Reuleaux: Robot Base Placement by Reachability Analysis. *ArXiv e-prints*, October 2017.

[30] Judith Müller, Udo Frese, and Thomas Röfer. Grab a mug-object detection and grasp motion planning with the nao robot. In *Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on*, pages 349–356. IEEE, 2012.

[31] Edwin Paredes, Christina Petlowany, Matthew Horn, Adam Allevato, and Mitch Pryor. Automated Glovebox Workcell Design. *WM2018 Conference, March , 2018, Phoenix, AZ*, Mar 2018.

[32] C. Peterson. Industrial Automation and Control in Hazardous Nuclear Environments. MS Thesis, The University of Texas at Austin, Austin, TX, May 2015.

[33] Donald L Pieper. The kinematics of manipulators under computer control. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1968.

[34] Pete C Pittman, TA Staab, David C Nelson, William W Santistevan, and Wendel G Brown. Automation of the lanl aries lathe glovebox. Technical report, Los Alamos National Laboratory, 2001.

[35] Mitch Pryor. Task-based resource allocation for improving the reusability of redundant manipulators. 2002.

[36] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[37] Nathan Ratliff, Matt Zucker, J Andrew Bagnell, and Siddhartha Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 489–494. IEEE, 2009.

[38] J Kenneth Salisbury and John J Craig. Articulated hands: Force control and kinematic issues. *The International journal of Robotics research*, 1(1):4–17, 1982.

[39] John Schulman, Jonathan Ho, Alex X Lee, Ibrahim Awwal, Henry Bradlow, and Pieter Abbeel. Finding locally optimal, collision-free trajectories

with sequential convex optimization. In *Robotics: science and systems*, volume 9, pages 1–10. Citeseer, 2013.

[40] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer-Verlag, Berlin, Heidelberg, 2007.

[41] Ioan A. Sucan and Sachin Chitta. MoveIt! `"http://moveit.ros.org/about/"`. Accessed: 2018-07-12.

[42] A. W. Thompson. Nuclear residues repacking glovebox, building 440, rocky flats nuclear weapons plant, 2002. [Online; accessed June 27, 2018].

[43] Nikolaus Vahrenkamp, Tamim Asfour, and Rüdiger Dillmann. Robot placement based on reachability inversion. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1970–1975. IEEE, 2013.

[44] Charles W Warren. Global path planning using artificial potential fields. In *Proceedings, 1989 International Conference on Robotics and Automation*, pages 316–321. Ieee, 1989.

[45] J. Williams. Improved Manipulator Configurations for Grasping and Task Completion Based on Manipulability. MS Thesis, The University of Texas at Austin, Austin, TX, December 2010.

[46] Tsuneo Yoshikawa. Manipulability of robotic mechanisms. *The international journal of Robotics Research*, 4(2):3–9, 1985.

[47] Franziska Zacharias, Christoph Borst, Michael Beetz, and Gerd Hirzinger. Positioning mobile manipulators to perform constrained linear trajectories. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 2578–2584. IEEE, 2008.

[48] Franziska Zacharias, Christoph Borst, and Gerd Hirzinger. Capturing robot workspace structure: representing robot capabilities. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 3229–3236. Ieee, 2007.

[49] Franziska Zacharias, Wolfgang Sepp, Christoph Borst, and Gerd Hirzinger. Using a model of the reachable workspace to position mobile manipulators for 3-d trajectories. In *Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on*, pages 55–61. IEEE, 2009.

Vita

Christina Petlowany was born in California and moved to Texas to attend Rice University in Houston. She worked on many projects at Rice, notably a wheelchair for a teenager with arthrogryposis and a food waste collection system, both of which were honored at Rice's annual engineering design showcase. She graduated summa cum laude from Rice University before joining the Nuclear and Applied Robotics Group at the University of Texas at Austin, where she was awarded the Provost's Excellence Fellowship and a U.S. Department of Energy Nuclear Engineering University Program Fellowship.

Email address: cpetlowany@gmail.com

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.