Implementing Motion Planning Software for Gretchen’s Manipulator

Caroline Pitts
May 6, 2016

Abstract

Gretchen is a mobile Fetch robot with a seven degree of freedom manipulator. Her current software generally creates motion plans that add a great deal of unnecessary movement to the solution path. This is a significant problem due to the fact unnecessary manipulator movement causes an increase to the probability of collision. The goal of this project is to replace Gretchen’s current motion planning software so that it produces nearly optimal paths. After applying a greedy smoothing heuristic to a Rapidly Exploring Random Tree algorithm, this project’s software produces paths that are both direct and smooth in position, acceleration, and velocity. Furthermore, this software can plan to avoid simple obstacles that have already been placed into Gretchen’s environment. In the future, this software could be improved by producing trajectories that speed up manipulator movement.
1 Introduction

Gretchen is a mobile manipulator that comes from the company Fetch robotics. She has a mobile base and a seven degree of freedom robotic arm. Figure 1 shows an example of a Fetch robot like Gretchen. The Fetch robots are generally meant for warehouse use and research.

![Fetch Robot](image)

Figure 1: On the left is Fetch robot and on the right is a Freight robot

Many companies that make robots today use the Robot Operating System (ROS) to help program various actions that they would like their robots to accomplish, and Fetch Robotics is no exception. ROS has different software libraries that make programming certain robots much easier. For a robot as complicated as Gretchen ROS is very useful because accessing all of Gretchen’s data becomes fairly easy, and in addition, it helps a great deal with finding the transformations between Gretchen’s different joints. Furthermore, this software is open source and, as a result, is easy to obtain.

Like many other open source software libraries, ROS has a few downsides. It is generally poorly documented, and it often uses other open source resources that usually are not tailored to every robot that uses ROS. MoveIt! [2], the software library that ROS provides to move mobile manipulators, works fairly well for robotic arms that only have a few degrees of freedom. In order to plan this movement, it utilizes an open source software library called the Open Motion Planning Library (OMPL). When OMPL attempts to move Gretchen’s manipulator from a specified start state to an end state, a great deal of unnecessary motion occurs as the arm takes the path that OMPL has planned for it. This is due to the fact that much of OMPL’s software uses random sampling-based planning to create manipulator trajectories, which often results in trajectories that are not optimal. Additionally, some of the less randomized planners that OMPL provides take a significant amount of time and still do not consistently provide optimal solutions to the motion planning problem.

All of this means that when working with Gretchen’s motion planning software it is necessary to carefully monitor the movement of her arm. This helps to make sure that it will not hit objects that Gretchen’s camera cannot see and confirms that it will not move violently enough to cause the base of the robot to change position. The need to monitor Fetch robots as they move their manipulators significantly reduces their utility in a warehouse as it wastes company time to have someone constantly monitor manipulator trajectories. Furthermore, it makes research far less efficient.
Figure 2: A picture of a Fetch robot with its joints labeled by name with arrows showing the way in which they move.

The result of this project attempts to fix the problem stated above by replacing the motion planning software for Gretchen’s manipulator. This motion planning software causes Gretchen’s manipulator to move more directly and smoothly from a given start state to a given end state; furthermore, it allows Gretchen’s arm to move around simple objects that have already been added to her environment.

Section 2 will describe the motion planning problem in more detail along with the algorithm that was used to solve this problem. Section 3 will talk about the various tasks that had to be implemented in order to create said motion planning software. Section 4 will look at the software created in this project and will compare it to the software that is on the original robot. Section 5 will give an overview of this project and will discuss future work.

2 Background

Overall, Section 2 will give more information on some of the background knowledge that will become useful to help to understand the rest of this paper. Section 2.1 will give a description of forward and inverse kinematics. Section 2.2 will talk about the motion planning problem and its importance to robotics and the world in general. Section 2.3 will discuss the various kinds of solutions to the motion planning problem, and Section 2.4 will describe the particular algorithm that this software uses to solve the motion planning problem.

2.1 Inverse and Forward Kinematics

Before solving the motion planning problem for Gretchen, it was important to solve for her inverse and forward kinematics. Section 3 will give more detail on how and why this task was accomplished.
First, it is important to define an end effector. In robotics, an end effector is the joint that completes a robotic arm. It generally is what interacts with other objects. Gretchen’s end effector is the gripper joint. This definition is important because it is very useful to try and move the end effector from one position in the world frame to another, but this can only be done using inverse kinematics.

The problem of inverse kinematics for a robotic arm is defined as finding the joint angles for each joint of a robotic arm that will allow the end effector to move to a given position in the world frame. The opposite problem of inverse kinematics is called forward kinematics. This problem is defined as finding an end effector’s position in the world frame given the joint angles of a manipulator.

For Gretchen, inverse kinematics is an especially difficult problem to solve given the fact that there are seven unknown joint angles. This problem involves a great deal of computation, but was a good precursor to solving the motion planning problem.

2.2 The Motion Planning Problem

As stated previously, the purpose of this project is to solve the motion planning problem for Gretchen’s manipulator specifically. That being said, motion planning is an important problem in many areas of research and industry. Firstly, it is an NP-complete problem[4] and therefore is interesting to those who are looking into minimizing algorithm run-time. Furthermore, there are many other fields in which it has become relevant such as computer graphics, biology, driver-less cars, and computer assisted surgery[5].

![Figure 3: A representation of a motion planning problem and its solution](image)

The most prevalent application of motion planning, however, is in the field of robotics. There are many different problems that motion planning can solve in this field alone, such as navigating a robot through a cluttered space, navigating a drone through the open sky, or, in the case that is most important to this project, navigating a manipulator from a given start position to a given end position. The result of the motion planning solution is a set of discrete states that form a path between a start and end state through a free space or a space containing obstacles. An example of a motion planning problem and its solution can be seen in Figure 3. These solutions are necessary in preventing robot programmers from having to specify each step of a robot’s trajectory, which saves a lot of time and effort and which allows a programmer to easily change his or her goal.

Unfortunately, motion planning is a very difficult problem to solve for Gretchen in particular. Figure 2 shows the seven different joints in Gretchen’s arm and the various directions in which they can move. Because Gretchen has a seven degree of freedom arm, it is far more difficult to find a solution for that planning problem as opposed to finding a solution for a robot with a two degree of freedom arm or for a robot that can only move in three different directions.
The difficulties that come from planning for a seven degree of freedom arm lie in the complexity of its configuration space. Most humans think about the three dimensional world around us as the space a robot might have to explore. This is called the workspace. Instead, a robot has to explore the configuration space. For a manipulator, one possible configuration is defined by a specific set of parameters. The configuration space is the set of vectors that contain these parameters. For Gretchen specifically, the configuration space is each position that the arm can take, and the parameters that define these positions are the joint angles of each joint that makes up the arm. This means that exploring the entire configuration space for Gretchen’s manipulator is computationally expensive, and, as a result, it is important to come up with a creative solution to this problem. This will be discussed further in the next section.

2.3 Solutions to Motion Planning Problem

Many solutions to the motion planning problem have been proposed, but most fall under two important categories. These are combinatorial planning and sampling-based planning. Combinatorial planning involves dividing the entire configuration space into discrete areas prior to solving the motion planning problem, while sampling-based planning uses collision detection to explore the configuration space incrementally[6]. Sampling-based planning on occasion may not explore different pieces of the free space, but this is not a problem as long as at least one solution is found.

As mentioned previously, exploring the entire configuration space for a seven-degree-of-freedom manipulator can be potentially a very computationally expensive process depending on how complicated the motion planning problem is[7]. If there are many obstacles and the manipulator is trying to move into a very tight space the planning problem becomes significantly more difficult. As a result, it made sense to use a sampling-based algorithm to plan for Gretchen’s manipulator movement. Two of the most currently and widely used sampling-based algorithms are the Rapidly Exploring Random Trees[8] (RRT) algorithm and Probabilistic Road-maps[9] (PRM) algorithm. These approaches are very similar except that RRT algorithms store the solution path states in a tree structure whereas PRM algorithms store these in a general graph structure. Furthermore, RRT traces the parent nodes from the end to start state in order to produce the final solution, whereas PRM finds the shortest path in the graph by running a shortest path algorithm.

Both RRTs and PRMs are good solutions to the motion planning problem, and deciding which algorithm to use should be based purely on the motion planning problem at hand. The motion planning software that was applied to Gretchen’s manipulator used the RRT algorithm to produce solution paths, and the next section will give a complete explanation of how this algorithm works.

2.4 Rapidly Exploring Random Trees

The RRT algorithm has been widely used in motion planning since it was proposed by Steve Lavalle in 1998[8]. Its popularity is due to the fact that it is fairly easy to understand and implement, it runs quickly, and it works well for systems with a high degree of freedom. Like most other algorithms, it also has a few downsides. When the motion problem becomes very difficult, RRTs can take an extremely long time to produce a motion plan, meaning that it is important to somehow limit the number of times that the algorithm can run. Overall, though, it is a very robust method to solve the motion planning problem and works well for Gretchen’s manipulator.

General RRT algorithms have a few common simple steps. First, the given start state is added as a node to the tree. Then, the next few steps repeat until a path is found going from the given start to the end state. The algorithm samples a random state from the configuration space, and finds the nearest node in the tree to that random state. Then it attempts to connect the nearest node in the tree to that random state incrementally. If the tree is not able to extend because it hits an obstacle, a new state is sampled, or if the tree reaches the randomly sampled state, then a new state is sampled, and the process begins again. Once the tree extends
Figure 4: An example of a RRT solution where the each state in the tree is a point and each obstacle is a box.

to the end state, the algorithm is nearly finished. It then needs to trace the back pointers in the tree from the end state to the start state. This allows the algorithm to output the computed path.

Algorithm 1, shown below gives a detailed overview of the RRT algorithm:

```
algorithm 1: RRT algorithm

In two dimensions, the output of RRT may look as the purple line does in Figure 4. For this motion planning problem, each state in the tree is an x, y coordinate, and the obstacles are sets of simple two dimensional boxes; furthermore, both the workspace and the configuration space are in two dimensions because in this case they are the same space.

For Gretchen, the output of RRT will be an arm trajectory. As mentioned previously, the configurations space will be in seven dimensions, and the world space will be the three dimensional world that humans live in and experience. The RRT algorithm as applied to Gretchen will be discussed in more detail in Section 3 below.
```
3 Results

As a whole, Section 3 will reveal the different tasks that were accomplished over the course of this project. Section 3.1 will discuss the process of creating smooth trajectories for Gretchen’s manipulator using inverse kinematics and joint space goals. Section 3.2 will describe how the software that creates manipulator trajectories fits in with certain ROS libraries. Then, Section 3.3 will explain how the RRT algorithm was tailored to work with Gretchen’s manipulator. Lastly, Section 3.4 will discuss adding obstacles to Gretchen’s environment.

3.1 Creating Smooth Trajectories

Before solving the motion planning problem, it was important to make sure that Gretchen’s manipulator could move if it was following trajectories that were made simply by solving for the inverse kinematics of its end effector. This was an important step due to the fact that solving the motion planning problem would be more difficult than solving for the inverse kinematics of the manipulator. Furthermore, this was a chance to become familiar with the libraries that would, in the future, be used to solve the motion planning problem.

In order to begin the process of creating basic trajectories for Gretchen’s manipulator, it was necessary to load her joint information into a C++ program that would simulate her movement. This was called “helloFetch.” Facts such as joint dimensions, shapes and connectedness are examples of the kind of information that was given to the simulator. These facts were contained in two different files called a Unified Robot Description Format (URDF) and a Semantic Robot Description Format (SRDF), which are both standardized formats that describe a robot’s joints and linkages in ROS. The output of this program was a simulation where manipulator trajectories could be visualized and tested.

Next, the information from the URDF and SRDF was used to solve for both the forward and inverse kinematics of Gretchen’s manipulator. This was done by using a set of previously written functions that came from a program called "fakerae." As a result, it became possible to move the manipulator from a given starting point to a given ending point in the world frame. In order to have Gretchen perform this task, it was necessary to create trajectories that would move smoothly from one point to another. This was done by using a function called "smoothstep." It would smooth the trajectory as a function of time and was described by: 
$$f(t) = 3t^2 - 2t^3.$$ 
This caused there to be no spikes in the acceleration of the arm.

Smoothstep could produce a set of points that would create smooth trajectories for Gretchen’s manipulator, and the joint angles needed to reach each of these points could be calculated using inverse kinematics. As a result, it was easy to simulate various trajectories for Gretchen’s manipulator, which would allow Gretchen to perform some simple tasks. These were moving Gretchen’s end effector in a square, and moving it to give a high five. It is important to keep in mind here that at this point in the process, manipulator trajectories were handcrafted and were not made using collision detection. As a result, they had to be checked for possible collisions during simulation.

In addition to moving Gretchen’s manipulator through world space, it was helpful to have it move through joint space. This was partially spurred by the fact that her two most frequent configurations were described by joint angles. Furthermore, it is easier to move through joint space due to the fact that inverse kinematics does not become part of the problem.

In order to tackle the problem of moving through joint space, it made sense to try and move Gretchen’s manipulator from what was called the "tuck state" to the "zero state" and vice versa. These are the two frequent configurations mentioned above and can be seen in Figure 5 and 6. Moving through joint space was a similar problem to moving through world space; however, instead of solving inverse kinematics for every point in world space that the manipulator needs to get to, it is possible to just set joint angles. Again, it was possible to use smoothing function to linearly interpolate between joint angles and to move the arm smoothly while doing so. In this way it was easy to create a set of seven movements that would tuck and untuck the manipulator.

Each trajectory created by helloFetch was sent to a text file that recorded the joint angles
for each movement the user wanted Gretchen’s manipulator to take. These text files were then loaded into a program written using ROS libraries. Section 3.2 will discuss what happened next.

3.2 Running Trajectories Using ROS

Throughout this project, helloFetch was always independent from the software written using ROS libraries. The main ROS program was called “trajectoryManager.” It was used to run manipulator trajectories both on the robot and in simulation software developed by ROS, which is called Gazebo. Because helloFetch and trajectoryManager are independent of each other, it was necessary to first import the previously mentioned joint angle text files into trajectoryManager. These joint angles could then be transformed into joint velocities and accelerations by using the second-order forward difference approximation of the first derivative. In a separate text file it then became possible to check on the velocities and accelerations that the arm would be running to make sure that neither would be too high.

Once the safety of the trajectory was established, it was time to actually move the manipulator. First, it was important to check that the true starting position of the robot was close to the text file’s starting position. Otherwise, the arm might move in erratic ways in order to get to the text file’s starting position. Once this was established, it was possible to move the arm using a library in ROS called the “SimpleActionClient.” This would move each joint if it was given a specified position, velocity, and acceleration.

Before running any of the text file trajectories on Gretchen, it was really important to run these trajectories in Gazebo to make sure that nothing would happen that would potentially harm the robot. Using the URDF and SRDF mentioned in Section 3.1, Gazebo could create a model of Gretchen. Furthermore, Gretchen’s software already had an environment for Fetch that included a floor and a sun. This environment was used to run manipulator trajectories and
is shown in Figure 7. If everything ran smoothly in Gazebo, it was safe to run trajectories on the real robot. Each of the trajectories mentioned in Section 3.1 ran very smoothly on Gretchen and the tucking and untucking trajectories were used throughout the project to get Gretchen’s manipulator to its home configuration.

3.3 RRT Algorithm Applied to Gretchen

The next step after creating smooth trajectories with inverse kinematics and by specifying joint angles, was to write motion planning software in helloFetch. Due to the high dimensionality of Gretchen’s arm and familiarity with the algorithm, it made sense to use an RRT algorithm to solve the motion planning problem. Furthermore, there was already RRT software that would solve the motion planning problem in a two dimensional space. With some modifications, it could also be used to solve the motion planning problem for Gretchen’s manipulator.

One of the first considerations for Gretchen’s RRT algorithm concerned how to represent the configuration space and workspace. After some thought, it made sense to represent each point in the workspace as a C++ vector of size three. The functionality for this kind of vector had already been encapsulated into the kakeawave program and was called “vec3.” The representation of each configuration on the other hand had to be created in helloFetch. The easiest course of action here was to use the C++ Eigen/Core library. This library can easily construct a one dimensional matrix that has a size of seven, and this is what was used to create a new variable type called “Vector7.”

Next, a Fetch RRT class had to be created so that the already functional RRT software could run with Gretchen instead of running on the two dimensional space shown in Figure 4. This class would use the variable types Vector7 and vec3, and it contained a few RRT helper functions. The most important of these is a function that samples a random configurations state, a function that finds the nearest node in the tree to that random state, and a function that extends the nearest node in the tree to that random state. Section 2.4 describes how the RRT algorithm uses these helper functions.

Once this Fetch RRT class was complete, it was possible to use the previously written RRT software to provide a solution for Gretchen’s motion planning problem. The output of this solution was a set of path states that Gretchen’s manipulator should follow to get from a given
start joint state to a given end joint state. Unfortunately, the raw trajectories that this software created were not smoothed over time and they were not optimal. The arm would essentially never take the shortest path possible to get to its goal. Most importantly, the accelerations and velocities were too high to run the trajectories created by this software on the robot. Clearly, improvements needed to be made to the resulting RRT paths.

![Diagram of RRT path smoothing](image)

**Figure 8:** Shows an example of the greedy smoothing algorithm working on an indirect path

Said improvements were made by implementing what is called a greedy smoothing heuristic[10] to the resulting RRT path. This heuristic essentially shortens the RRT path by removing many of the unnecessary path states. As a result, the manipulator can move more closely around various obstacles. The heuristic accomplishes this task by repeating the following steps 1500 times. First, two different, random states are picked from the path produced by the RRT. Then, a call is made to the function in the Fetch RRT class that attempts to create a path that extends from one RRT state to another. If this extension is a successful operation, then it replaces the path that originally connected the two random RRT states. Otherwise the original RRT path continues to connect these states. A two dimensional visualization of this software can be seen in Figure 8.

The result of this greedy smoothing heuristic is a small number of path states, usually below 15, that take a nearly optimal path from the given starting joint states to the ending joint states. At this point, each path state is generally somewhat far away from its neighbors. As a result, the same smoothing function previously mentioned in Section 2.1 must be applied to move the manipulator in between path states, which leads the manipulator to stop at each path state in the trajectory. When a smooth trajectory has been finalized, it is outputted to a text file and then it is executed using the previously mentioned trajectoryManager program.

At this point in the process, Gretchen's manipulator is simply avoiding self collisions. Next, boxes were added to its environment in order to make the motion planning problem significantly more difficult. Section 3.4 goes into more detail regarding this step.

### 3.4 Adding Obstacles

Adding obstacles to helloFetch was the final step needed to complete the motion planning software for Gretchen's manipulator. First and most importantly, a floor needed to be added to the simulation. Without it, each trajectory needed to be monitored in order to make sure that Gretchen's manipulator would not collide with the ground plane. The floor was added as a large box that would cover each of the points Gretchen's end effector might be able to reach. In order to add the floor in a way where it wouldn't interfere with any joint but the manipulator, Gretchen's base joint was allowed to touch the ground plane.
Next, other boxes were added to the simulation environment in order to make the motion planning problem more difficult. These boxes were inserted as obstacles and would directly affect a typical RRT path. They were added into the simulation software and into Gazebo, as can be seen in Figures 9 and 10.

While these boxes made the RRT path more difficult to plan, the resulting software was still very efficient and ran in a similar amount of time to Gretchen’s original planning software.

4 Discussion

Overall this project resulted in new software that would move Gretchen’s manipulator by using inverse kinematics, by interpolating through joint space, and by solving the motion planning problem. The real question here, though, is how does Gretchen’s new motion planning software compare to MoveIt!? First, it is important to look at the benefits of the motion planning software that was produced by this project.

First of all, this motion planning software produces paths that are much more direct than those produced by MoveIt! This feature is one of the largest benefits of the overall project. Often, Gretchen’s original paths were so suboptimal that it was necessary to monitor each movement Gretchen would make. Now, the trajectories that are produced by the helloFetch software run without complication and require almost no monitoring prior to running them on the robot.

Second, Gretchen’s new paths are very smooth. This means that there are very few spikes in the arms acceleration and that these spikes do not affect the overall safety of running a trajectory on Gretchen’s manipulator. Previously, manipulator trajectories would sometimes run in such a way that would cause Gretchen’s base to change position. This is a very large problem, because motion planning usually requires the planner to know where in the world space a robot is before it can begin to make a motion plan. Moving the base of the robot may cause the planner to unintentionally collide with the obstacles around the robot. Gretchen’s new software never moves the base of the robot around at all.
Finally, the planner is efficient and runs in a matter of seconds. This was rarely a problem for the motion plans that MoveIt! tended to produce; however, the plans that were less randomized and therefore more optimal took significant time to plan. Gretchen's new motion planning software rarely takes longer than MoveIt! and the new software is more optimal overall.

While the benefits of the new software are significant, there are some aspects of it that would be good to improve upon in the future. To begin, helloFetch is currently totally independent of ROS. This is not a huge problem due to the fact that trajectories can be passed into trajectoryManager through text files. If this software was to be used in an official warehouse setting however, planning would need to be done on the fly. This means that it would be better to run the motion planning software in one simple program that could somehow use helloFetch to produce RRT paths.

Additionally, it would be nice to produce trajectories that would not stop at each RRT path state. This slows the motion of the arm down significantly, and in the real world would make moving Gretchen's manipulator over and over again far less efficient. One benefit of motion plans created in MoveIt! is that they tend to move in a more continuous way.

This point ties into one of the final negatives of Gretchen’s new motion planning software. The motion of the arm when it is completing trajectories outputted by helloFetch is quite slow. In part, this feature is due to the fact that Gretchen is a very complicated robot and breaking her manipulator would have had very negative consequences for this project. Without creating smooth manipulator trajectories, the probability of harming a joint in Gretchen’s hardware goes up significantly. Mostly, though, the speed of the arm is impeded because it stops at each path state.

The decision of whether to use Gretchen’s current or new software should be made based on the motion planning problem that Gretchen’s user is trying to solve. In a fairly free space, Gretchen’s original software will tend to allow the arm to move more quickly to its goal. In a room filled with more obstacles, Gretchen’s new software will move more safely and more directly from its start state to its goal state.

5 Conclusion

The overall purpose of this project was to create smoother and more direct motion planning software for Gretchen’s manipulator. Before trying to solve the motion planning problem, trajectories were created by using inverse kinematics and by specifying joint states. Once it was established that these trajectories would successfully and smoothly move Gretchen’s manipulator, a solution to the motion planning problem could more easily be found. This was accomplished by using an RRT algorithm and a greedy smoothing heuristic. The path that resulted from this software was direct and smooth, but it was independent of ROS, it stopped at every path state, and it was slow.

In the future, there are a few ways that this project can be extended. First, it would be a good step to address the negatives of this software. It would be ideal to create RRT paths that are still direct and smooth but would be nice if the arm ran more quickly when following these paths. Furthermore, it would be nice if the arm would move continuously to its goal state instead of stopping at each step of the RRT path. Also, the software would be more useful if helloFetch was somehow integrated with ROS libraries. This task is not very difficult, but fell outside of the scope of this project.

Finally, this project could be extended so that Gretchen’s manipulator would perform certain tasks by calling on the motion planning software. At this point the motion planning software’s most useful function is to move the arm from one set of joint states to another. Previous tasks that Gretchen’s manipulator has been used for are picking up objects, placing objects, and pressing the button on an elevator. HelloFetch could plan for each of these tasks easily.

Overall, this project produced successful motion planning software and was a great way to learn about motion planning for a manipulator with many degrees of freedom.
6 Appendix

Gretchen’s motion planning software can be found at the link:
https://github.com/swatbotics/hubomz/tree/devel/urdf

Any software that was written using ROS libraries can be found at the link:
https://github.com/cpitts1

References


