5-2019

Motion Planning for a Continuum Robotic Mobile Lamp: Navigating the Configuration Space to Assist with Aging in Place

Zachary Hawks
Clemson University, zacharyj.hawks@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses

Recommended Citation
Hawks, Zachary, "Motion Planning for a Continuum Robotic Mobile Lamp: Navigating the Configuration Space to Assist with Aging in Place" (2019). All Theses. 3115.
https://tigerprints.clemson.edu/all_theses/3115

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
MOTION PLANNING FOR A CONTINUUM ROBOTIC MOBILE LAMP: NAVIGATING THE CONFIGURATION SPACE TO ASSIST WITH AGING IN PLACE

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

by
Zachary Hawks
May 2019

Accepted by:
Dr. Ian Walker, Committee Chair
Dr. Ioannis Karamouzas
Dr. Adam Hoover
Abstract

For a robot to operate autonomously, it must have a method of planning its motion through its environment without the explicit guiding control of a human operator. In this thesis, a new approach was implemented to plan a collision-path for a mobile robot featuring a novel continuum arm.

We consider motion planning in the configuration spaces of robots containing continuum elements. The configuration space structure of extensible continuum sections was first analyzed, with practical constraints unique to continuum elements identified. The results were applied to generate the configuration space of a hybrid continuum lamp/mobile base robot developed as a part of a wider project aimed at robots in the home to assist aging-in-place. A conventional motion planning (Rapidly-exploring Random Tree search, RRT/A*) approach was subsequently applied for the robot in the aging-in-place application scenario.

The RRT generated complete paths through various environments and was successfully able to connect the start configuration to the goal configuration using the robot’s specific configuration space. Once the RRT completed, an A* search algorithm was run on the graph and the optimal path was found. This path, consisting of series of actions necessary for the robot to move from configuration to configuration, was then communicated to two generations of robot hardware using a local wireless network. The robots then executed the actions and moved through the environment.
Dedication

I’d like to dedicate this thesis to Karen: You always believed in me even when I stopped believing in myself. Your support and encouragement got me through the hardest moments, and I will always love you for that.
Acknowledgments

I’d like to thank my adviser Dr. Ian Walker for all his support and guidance in this work. I’d also like to acknowledge my committee members Dr. Ioannis Karamouzas and Dr. Adam Hoover for their contributions.

In addition, I’d like to thank the other members of our lab for all their help and camaraderie. Specifically, I want to thank Chase Frazelle, whose help got me through the toughest problems and whose friendship made grad school more than just a job. I couldn’t have done it without you, Chase.
Table of Contents

Title Page ........................................................................... i
Abstract ............................................................................. ii
Dedication ........................................................................... iii
Acknowledgments ................................................................. iv
List of Figures ..................................................................... vii
1 Introduction ..................................................................... 1
2 Home+ and the Second Generation Continuum Robotic Mobile Lamp . 5
  2.1 In-Home Scenario ......................................................... 5
  2.2 Implementation of Path Planning for Mobile Base of h+lamp ....... 11
  2.3 Results .......................................................................... 13
  2.4 Analysis ......................................................................... 14
3 Third Generation Continuum Robotic Mobile Lamp: CuRLE .......... 18
  3.1 Structural Overview ....................................................... 18
  3.2 Electronics Upgrades ...................................................... 20
  3.3 Details of the Hardware Upgrades ................................. 22
  3.4 Kinematics ................................................................... 35
4 Configuration Space of CuRLE ........................................... 38
  4.1 Continuum Configuration Space .................................... 38
  4.2 A Comparison: Equivalent Rigid-Link Robot Configuration Space 45
  4.3 Continuum Robotic Lamp Element: CuRLE ..................... 52
  4.4 Configuration Space of Mobile Base ................................. 55
5 Motion Planning .................................................................. 58
  5.1 Path Planning for the Mobile Base ................................ 59
  5.2 Path Planning for the Continuum Section ....................... 67
6 Validation of Motion Planning with CuRLE Robot Hardware .......... 76
# Table of Contents

6.1 CuRLE Software Implementation .............................................. 76
6.2 Validating the Continuum Section Controller .............................. 79
6.3 In-Home Scenario .............................................................. 79
6.4 Sub-Scenario 1: Simple Base Movement and Grasping the Cup with Continuum Element ......................................................... 80
6.5 Sub-Scenario 2: Complex Base Movement and Placing the Cup with Continuum Element ......................................................... 83
6.6 Sub-Scenario 3: Simultaneous Movement Between Base and Continuum Element ............................................................. 84

7 Conclusions and Suggestions for Future Research ........................ 98
  7.1 Conclusions .......................................................................... 98
  7.2 Future Work ....................................................................... 100

Appendices ................................................................................. 105
  A CuRLE Robot Software: Arduino ............................................. 106
  B CuRLE Robot Software: Raspberry Pi .................................... 164
  C CuRLE Robot Software: Central Computer .............................. 171
  D Motion Planning Software ....................................................... 173
  E Continuum Element Simulation Software ................................. 225
  F Mobile Base Simulation Software ........................................... 282

Bibliography ................................................................................ 304
# List of Figures

1.1 The evolution of the continuum robotic mobile lamp element of the home+ suite. (a) The original first generation lamp reported in [1]. (b) The second generation, called h+lamp, also described in [1]. (c) The third generation, called CuRLE (Continuum Robotic Lamp Element) that forms the basis for this thesis. ........................................... 4

2.1 The home+ suite of robotic furnishing elements. (a) h+cube (b) h+lamp (c) h+armoire .......................................................... 6

2.2 First generation lamp hardware, as detailed in [1]. ........................................... 8

2.3 (a) The full replicated and upgraded hardware of the h+lamp. (b) A top-down view of the base of h+lamp showing the tendon motors and electronics. (c) A side-view of the of the base of h+lamp showing all of the electronics (i.e. Arduino Mega, motor drivers, voltage regulator, power supply, Wi-Fi shield) ........................................... 9

2.4 A side-view of the h+lamp drive subsystem. ................................................ 10

2.5 Tele-operation method of the h+lamp hardware using custom-built controller and Bluetooth. ................................................ 10

2.6 Tele-operation method of the h+lamp hardware using Xbox360® controller and wireless LAN. ........................................... 11

2.7 The path for the h+lamp to follow through the in-home scenario experiment. 14

2.8 A top-down view of the physical task space of the in-home scenario used to validate the motion planning algorithms using the h+lamp robot hardware. The start location is shown in the top right in green with the goal location shown in the bottom left in yellow. The configuration space obstacles are shown in blue. ........................................... 15

2.9 A series of top-down views showing the h+lamp moving through the physical task space of the in-home scenario. ........................................... 16

3.1 Third generation continuum robotic mobile lamp (CuRLE) with internal LED stip lit. ........................................... 19

3.2 The full base of CuRLE. ............................................................ 21

3.3 The central structure of CuRLE that houses all of the electronics is shown with the motor plate removed. ........................................... 23
3.4 The assembled center structure with different electronic components visible. (a) The Raspberry Pi. (b) The Arduino Due (c) Voltage regulator (bottom) and motor drivers (top) .................................................. 24
3.5 The two main electronics plates removed from the central structure. (a) Top plate. (b) Left-half and (c) right-half of bottom plate. ................................. 24
3.6 (a) The turntable mechanism of CuRLE. (b) Zoomed in view of the connection between the worm drive and the turntable. ................................. 25
3.7 The central structure mounted on the turntable mechanism. ...................... 26
3.8 (a) Top-side view of the passive drive element showing the through-holes for the metal rods that provide stability and alignment in the shocks. (b) Side view of the shock assembly mounted to the ball-bearing casters (passive drive element). .......................................................... 28
3.9 (a) Side view (under frame) of the shock assembly for one differential drive motor. (b) In-line view of the drive motor shock assembly. (c) Side view (outside frame) of the shock assembly for the drive motor. ............................ 29
3.10 The motor plate which is mounted to the top of the central structure. Here is where the tendon motors are mounted with the tension sensors and the spools to wind the tendons and measure length. ........................................... 31
3.11 (a) The 3D-printed assembly of a single tendon motor ............................. 32
3.12 (a) The gripper mounted as CuRLE’s end-effector with the LED lights lit and the flexible grasping “fingers” removed. (b) The gripper with the “fingers” added. .......................................................... 34
3.13 The power system of CuRLE. (top) 4S LiPo battery to power the electronics. (middle) 4S LiPo battery to power the LEDs, LED strip, and servo motors in the gripper. (bot) 4S LiPo battery (later replaced with a 6S) to power all of the motors. .......................... 36

4.1 An example of a single section extensible continuum robot [2] ......................... 39
4.2 A simple sketch demonstrating the kinematic variables of a single section extensible continuum element. .................................................. 40
4.3 Single section continuum robot bending (a) counter-clockwise and (b) clockwise in the yz-plane. .......................................................... 41
4.4 A visualization of $C^2_{space}$. The blue plane extends to $\pm \infty$. The red circle indicates $C^2_{space}$ with the physical constraint of $\theta \leq 2\pi$. ...................... 41
4.5 A visualization of $C^3_{space}$. The dotted lines bordering the solid and the arrows indicate that $u, v \rightarrow (\pm \infty)$ and $s \rightarrow (+\infty)$. The prism is bounded by the plane $s = 0$. .......................................................... 42
4.6 An illustration of physical constraints of bending a continuum robot. In (a), $L_1 = L_2 = s$. In (b), the robot has bent counter-clockwise, causing $L_2$ to lengthen and $L_1$ to shorten, while $s$ remains constant. ................. 44
4.7 A visualization of $C^3_{space}$. In (a) the physical constraints of the backbone are illustrated. The maximum bend can be achieved when $s = \frac{s_{max} - s_{min}}{2}$, which is the widest plane in the center of the pyramid. In (b), the physical constraint of $\theta \leq 2\pi$ is applied to (a), which forms the “rounded” pyramid shape. The largest “uv-plane” indicated circle in (b) is the same circle shown in Fig. 4.4.

4.8 A sketch showing (a) the task-space-equivalent rigid-link RRPRR robot in the same configuration(s) as (b) the continuum element. This is the result of the kinematic mapping $F$.

4.9 The task space of both the continuum section (black) and rigid link structure (green) for different values of $u$ for the continuum section and $[\theta_1, d]$ for the rigid-link robot.

4.10 $Q^2_{space}$ of the rigid-link robot. The arrows indicate the “wrapping” phenomenon that occurs when $\theta_1$ and $\theta_2$ go beyond the bounds $[0, 2\pi)$.

4.11 A visualization of $Q^2_{space}$ of the rigid-link robot to show the “wrapping” phenomenon. A configuration $q$ is any point on the surface. Changing $\theta_1$ “rotates” $q$ around the axis, $a_c$, running through the center of the torus. Changing $\theta_2$ “rotates” $q$ around $a_t$ which is the tangent to the path of rotation of $\theta_1$.

4.12 $Q^3_{space}$ of the rigid-link robot. The arrows indicate the “wrapping” phenomenon that occurs when $\theta_1$ and $\theta_2$ go beyond the bounds $[0, 2\pi)$. The configuration space (red) of the rigid-link structure, $Q^C_{space}$, once it has been restricted by $F$ to have the equivalent task space as the continuum section. This is displayed within the full c-space (blue), $Q^3_{space}$, from Fig. 4.12.

4.13 The configuration space of the mobile base in the scenario presented above.

4.14 $C_{space}$ of CuRLE.

4.15 CuRLE’s base is a square frame with rounded corners. The radius of the “disc” used to estimate the base of CuRLE is shown in green.

5.1 Plots showing the different graphs through the simulated environment for different types of RRTs. The path started at the green node and followed the yellow path to the magenta goal node, avoiding the configuration obstacles defined by the slashed lines surrounding the red obstacles in the task space.

5.2 Plot showing the RRT (type=VERTEX) expanding into an open environment (no obstacles). The green node indicates the start.

5.3 (a) RRT Experiment 1 Results. (b) RRT Experiment 2 Results.

5.4 The configuration space of the mobile base in the scenario presented above.

5.5 The RRT growth over time.

5.6 A progression of images showing the simulated robot moving through the scenario.

5.7 The task space showing the cup from the scenario. The middle cup is the “goal” cup that CuRLE will pick up and deliver to the user.
5.8 The configuration space obstacles of the task space. The start configuration, is shown in (a), while (b) is the magnified obstacle from (a) where the goal configuration is located. (b) is the c-space obstacle of the shelf shown in Fig. 5.7.

5.9 The interactive GUI that serves as the front end for simulation environment.

5.10 The simulated CuRLE in the start (vertical) and goal (bent) configuration. The objective of the scenario was to pick up the cup, shown as a light grey prism, on the shelf.

6.1 (a) The state of CuRLE after $\omega$ has aligned with the goal $\omega$. (b) CuRLE has grasped the cup. (c) The results of a second path generated by the RRT (shown in Fig. 6.11) that guided CuRLE to pick the cup off the shelf.

6.2 The in-home scenario explored to demonstrate the full functionality of CuRLE. The Locations discussed below are numbered in the image. The first shelf is located at Location 2 and the second shelf is at Location 3.

6.3 Output of the RRT showing the path required for the mobile base of CuRLE to navigate from the start location to the first shelf.

6.4 Output of the RRT showing the path required for CuRLE to grasp the cup on the first shelf. (b) is a magnified portion (a) to better show the start and goal configurations.

6.5 Output of the RRT showing the path required for CuRLE to pick the cup up off the first shelf. (b) is a magnified portion (a) to better show the start and goal configurations.

6.6 (a-b) Results from Sub-Scenario 1 showing the execution of the RRT from Fig. 6.3 for the mobile base (i.e. CuRLE move from Location 1 to Location 2).

6.7 (a-b) Results from Sub-Scenario 1 showing the execution of the RRT from Fig. 6.4 (i.e. CuRLE grasp cup).

6.8 (a) Results from Sub-Scenario 1 showing the execution of the RRT from Fig. 6.5 (i.e. CuRLE lift cup off shelf). (b) Results from Sub-Scenario 2 showing the execution of the RRT from Fig. 6.9.

6.9 Output of the RRT showing the path required for the mobile base of CuRLE to navigate from the first shelf to the second shelf.

6.10 Output of the RRT showing the path required for CuRLE to place the cup on the second shelf. (b) is a magnified portion (a) to better show the start and goal configurations.

6.11 Output of the RRT showing the path required for CuRLE to release the cup on the shelf and move away. (b) is a magnified portion (a) to better show the start and goal configurations.

6.12 (a-b) Continued results from Sub-Scenario 2 showing the execution of the RRT from Fig. 6.9 for the mobile base (i.e. CuRLE move from Location 2 to Location 3).
6.13 (a-b) Results from Sub-Scenario 2 showing the execution of the RRT from Fig. 6.10 (i.e. CuRLE release the cup). 92
6.14 Output of the RRT showing the path required for the mobile base of CuRLE to navigate from the second shelf to the "docking station". 93
6.15 Output of the RRT showing the path required for CuRLE to return to its "home" state ($[u = 0, v = 0, w = 0]$). 94
6.16 (a-b) Results from Sub-Scenario 3 showing the execution of the RRT from Fig. 6.14 for the mobile base executing in parallel with the output of the RRT from Fig. 6.15 (i.e. CuRLE return to "home" state while moving from Location 3 to Location 4). 95
6.17 (a-b) Continued results from Sub-Scenario 3 demonstrating the parallel execution of the two RRTs. 96
6.18 Final results of entire experimentation showing the small amount of error in the final position of the mobile base. 97
Chapter 1

Introduction

This thesis addresses the problem of motion planning for mobile robots featuring novel continuum sections. We discuss the nature of the configuration space of, and its use in motion planning for, continuum robots.

Continuum robots are composed of one or more continuum sections. A continuum section is kinematically described by continuous and smooth curvature [3, 4]. Continuum robots theoretically possess infinite degrees of freedom (DoF), unlike standard rigid-link robots which have finite DoF. Continuum sections are most often tendon or pneumatically driven, or composed of concentric tubes. These robots are often inspired by elements in biology, such as plant tendrils, an elephant trunk, or octopus tentacles. Because of their underlying curvature, continuum robots are often compliant in nature and are used to explore hard-to-reach areas [5, 6].

Generating control algorithms for robots, including continuum robots, is a well-studied problem, and multiple solutions are widely accepted and used [4, 7]. Often, these robots are tele-operated and a lot of effort has been focused on intuitive control methods for different robots [8, 9]. In contrast, autonomous motion requires motion planning.

Motion planning is the process of determining a path between two configurations
of the robot using its kinematics. A robot’s configuration can be described as vector of the current value(s) of the independent kinematic variables of the robot. The set of all possible configurations is the configuration space of the robot.

For conventional, non-continuum robots, classical motion planning techniques using configuration space have been well studied [10]. For example, rapidly exploring random tree (RRT and RRT*) algorithms have been shown to successfully span the configuration space for mobile robots and rigid-link robots [11, 12]. The A* algorithm, given a graph and proper heuristic function, will guarantee the optimal path between any two nodes if one exists [13].

In the past, a variety of motion planning techniques have been proposed for continuum robots. The motion planning problem for active cannulas (concentric tube robots in medical applications) within tubular environments is formulated as an constrained optimization problem in [14].

Constrained optimization is also used in [15] to formulate and solve the motion planning problem for a soft planar continuum manipulator. Grasp planning for continuum robots using a bounding circle technique was investigated in [16] and [17]. A follow the leader approach for tendon-driven continuum robots is introduced in [18]. Researchers have used sampling based approaches based on the techniques of Rapidly-Exploring Roadmaps (RRM) [19], Rapidly-Exploring Random Graphs (RRG) [20, 21], and Rapidly-Exploring Random Trees (RRT and RRT*) [22, 23] to plan motions for concentric tube continuum robots in tubular environments for medical applications. RRTs are also used by [24] for steering bevel-tip needles in 3D (medical) environments.

However, it appears that the principles of the classical motion planning techniques such as RRT and A* have not yet been applied to tendon-actuated continuum robots in general non-tubular environments. This is in part due to a lack of formal analysis of the nature of the configuration space of tendon-actuated continuum robot elements. In this thesis, we
define and discuss the configuration space of single section extensible continuum robots and use the configuration space to plan paths for hybrid mobile/continuum robots using RRT. We select RRT for its ability to operate well in dynamic environments as an anytime-approach [11]. As we will detail further, both the environment and current objective might change rapidly, and the RRT approach will help to account for this.

The thesis is organized as follows: first in Chapter 2 we discuss the history of home+, our collection of robotic home furnishing elements designed to assist in the home with aging-in-place, and introduce the continuum robotic mobile lamp in Fig. 1.1(a-b) [1]. In that Chapter we also discuss the key motivation —to envision autonomous control for the h+lamp —behind our investigation of classical motion planning techniques applied to the configuration space of a hybrid mobile/continuum robot. In Chapter 3, we then discuss the hardware upgrades that we have made to the h+lamp that have led to the development of CuRLE, the third generation hybrid mobile/continuum lamp, shown in Fig. 1.1(c). The research in this thesis reported in Chapter 4 analyzes the configuration space for tendon-driven continuum sections for the first time [25]. The analysis is then applied to the specific example of CuRLE and further detailed in Chapter 4. In the process, we illustrate and highlight several previously unconsidered structural constraints imposed by the tendon-actuated continuum geometry.

The insight gained, and the configuration space models constructed, in Chapter 4 are exploited in Chapter 5 to develop motion planning algorithms for CuRLE. This work represents the first motion planning for hybrid mobile robot/tendon-driven continuum robots. The efficacy and potential of the results are demonstrated via a series of demonstrations using CuRLE reported in Chapter 6. Conclusions and suggestions for future work are presented in Chapter 7.
Figure 1.1: The evolution of the continuum robotic mobile lamp element of the *home*+ suite. (a) The original first generation lamp reported in [1]. (b) The second generation, called h+lamp, also described in [1]. (c) The third generation, called CuRLE (Continuum Robotic Lamp Element) that forms the basis for this thesis.
Chapter 2

*Home*+ and the Second Generation

Continuum Robotic Mobile Lamp

This Chapter describes the first attempt at realizing the *home*+ continuum robotic mobile lamp as an autonomous system. In doing so, we first discuss the key motivation driving the *home*+ research project in more detail, and then describe how the work in this thesis supports that effort.

2.1 In-Home Scenario

With the current societal move towards smart devices in every home, we envision a collection of robotic furnishing elements that can provide at-home care and assistance. As we age, we often lose the ability to perform simple day-to-day tasks and eventually reach a point where we can no longer live without assistive care. Our suite of robots, collectively called *home*+, are intended to collaborate with individuals over time in the home to help with these day-to-day tasks and prolong the time that the individual can live independently [1].
We often envision at-home robotic care to be administered by fully functional, android-esque robots such as those seen in cinema and dreamed of in science fiction. That technology, however, has yet to be created, while many of the tasks that individuals need assistance with can be accomplished by robotic technology that currently exists. For instance, people need help retrieving objects from high shelves, so a continuum robotic mobile lamp that includes the ability to do such tasks (in addition to functioning as a lamp) was developed prior to the work reported in this thesis. This first generation robot, shown in Fig. 2.2, was added to the home+ suite. The other elements of home+ are a robotic end-table, or "cube", which is detailed in [1], and a robot ”armoire” (Fig. 2.1). These robots coordinate together with the user to accomplish every-day tasks and assist the user with aging-in-place.
One of the key motivations driving the home+ effort has been to evaluate various levels of user interaction with the suite of robots. For the work reported in this thesis, we selected the continuum robotic mobile lamp experiment with the spectrum of user control. To this end, we made extensive upgrades to allow for tele-operation and autonomous motion, as seen in Fig. 2.3 and 2.4. With this second generation robot, referred to as h+lamp hereafter, we were able to conduct preliminary experiments to evaluate motion planning for the mobile base.

2.1.1 Tele-Operation of h+lamp

On one end of the spectrum of control, the user is fully in command of the robot via tele-operation, in which the user provides input to the robot via some form of user-interface. The first generation lamp was controlled by a series of switches directly wired between the power supply and the actuators [1]. Our initial upgrades moved this tethered, analog control method to be wireless and digital. A hardware controller, shown in Fig. 2.5, received input from the user and communicated the commands with wireless Bluetooth technology to the robot. This was subsequently replaced with an Xbox360® controller communicating over a wireless LAN connection by installing Wi-Fi enabling hardware into the h+lamp, shown in Fig. 2.6. A central computer decoded input on the controller via C++ code which then sent the commands to the robot over Wi-Fi.

2.1.2 Autonomous Motion of the h+lamp

The other end of the control spectrum is full autonomy of the robot. In this mode, the user would give verbal, high-level commands, such as "Bring me my cup." and the h+lamp would autonomously navigate (avoiding obstacles) to where the cup is stored, grab the cup, and bring it back to the user. This is the mode the work in this thesis was intended to
Figure 2.2: First generation lamp hardware, as detailed in [1].
Figure 2.3: (a) The full replicated and upgraded hardware of the h+lamp. (b) A top-down view of the base of h+lamp showing the tendon motors and electronics. (c) A side-view of the base of h+lamp showing all of the electronics (i.e. Arduino Mega, motor drivers, voltage regulator, power supply, Wi-Fi shield)
Figure 2.4: A side-view of the h+lamp drive subsystem.

Figure 2.5: Tele-operation method of the h+lamp hardware using custom-built controller and Bluetooth.
enable. As a first attempt to do this, the passive mobile base (realized by steel ball-bearing casters) was replaced with a differential drive system. This is detailed in section 2.2. To implement full autonomy, sensing capabilities would have to be realized for the h+lamp to recognize user input via voice command and detect the environment obstacles, in addition to navigating the task space. The realization of this is beyond the scope of the project reported in this thesis. As described later in Chapter 5, in this work the user’s commands (i.e. the goal configurations) and the location of all obstacles are assumed known \textit{a priori} and we demonstrate progress towards autonomy with the novel motion planning algorithms we developed.

2.2 Implementation of Path Planning for Mobile Base of \textit{h+lamp}

The differential drive system developed as part of this thesis work consisted of two encoded DC gear-motors and a dual H-bridge motor driver controlled by PWM signals from the Arduino Mega development board that functioned as the micro-controller for the robot. The drive system, shown in Fig. 2.4, was powered by a 4S LiPo battery and step-down voltage regulator, while the electronics were separately powered by 4 AA batteries.
Fig. 2.3 shows the h+lamp in this prototype version.

As mentioned earlier, the h+lamp connected to a central computer over Wi-Fi. Rather than transmit user inputs from an Xbox360® controller, however, the central computer ran the motion planning algorithms to autonomously move the robot from configuration to configuration. These algorithms, RRT and A*, are detailed in Chapter 5.

The output of the RRT/A* algorithms was the path the robot must follow to reach the goal. The path consisted of the set of actions \( U = \{\mu_1, \mu_2, \ldots, \mu_n\} \), which was wirelessly transmitted, one action at a time, to the robot via a TCP/IP socket connection. The robot acknowledged the initial transmission ("init") and began executing the actions as they arrived. If a new action arrived before the robot finished its current action, the new action was queued and executed next.

The motors were controlled by a PID controller implemented in the Arduino IDE. As each action arrived, the robot calculated the new set-point of each wheel based on its dynamics. A simple state-machine controled the flow of execution. The robot was idle until it received the “init” command from the central computer. At this point, the robot acknowledged the central computer and entered a waiting state until an action command arrived. Once an action arrived, the robot moved through the action vector, first performing a rotation, then translation, then the final rotation. Each individual movement was executed by calculating the distance each wheel would travel. For a rotation, the distance \( \sigma_i \) traveled by each wheel is given by Eqn. (2.1) where \( L_{\text{wheels}} \) is the distance between the wheels and \( \phi_i \) is the rotation of the current action.

\[
\begin{bmatrix}
\sigma_1 \\
\sigma_2
\end{bmatrix} = \begin{bmatrix}
\phi_i \times \frac{L_{\text{wheels}}}{2} \\
\phi_i \times \frac{L_{\text{wheels}}}{2}
\end{bmatrix}
\] (2.1)
For a translation, the distance $\delta_i$ traveled by each wheel is given by Eqn. (2.2) where $\delta$ is the translation distance of the current action $\mu$.

$$
\begin{bmatrix}
\delta_1 \\
\delta_2
\end{bmatrix} =
\begin{bmatrix}
\delta \\
\delta
\end{bmatrix}
$$

(2.2)

Either $\delta_i$ or $\sigma_i$ was added to the current position of the wheel to give the desired set-point for each wheel. The current position of each wheel $p$ is given by Eqn. (2.3) where $E_{\text{count}}$ is the current encoder count, $C_{\text{rev}}$ is the counts per revolution of the wheel, and $r_{\text{wheel}}$ is the radius of the wheel.

$$
p = E_{\text{count}} \times C_{\text{rev}} \times r_{\text{wheel}} \times 2\pi
$$

(2.3)

Once the desired set-point was calculated, the PID controller calculated the error for each wheel and updated the velocity of the motors to move the error to zero. Once the error was consistently below the minimum threshold value, the state-machine moved to the next value of the action vector. If the action vector was complete and there were actions waiting in the queue, the top value was popped out of the queue and execution began again; otherwise the system moved back to the ready state awaiting the next action.

### 2.3 Results

After successfully running the RRT, watching the graph grow, and watching the simulated robot successfully navigate the path (Chapter 5), we implemented the RRT on the h+lamp. Before running the full path, the PID motor controller was thoroughly tested.
Figure 2.7: The path for the h+lamp to follow through the in-home scenario experiment.

and tuned by serially sending the robot one action at a time. We also verified the state machine logic and saw the robot successfully queue actions as it received them, executing them in order one at a time. After placing the robot in the work space, which is shown in Fig. 2.8, we executed the RRT and sent the path to the robot over Wi-Fi. The path for the robot to follow is shown in Fig. 2.7. The best results of the robot executing this path are shown in Fig. 2.9.

2.4 Analysis

The results revealed the need for improvements in the prototype hardware. The robot began execution, but due to errors in the motor control, was not able to successfully navigate to the goal. Errors in the motor control occurred due to slight differences in the velocities of the two different drive motors. For rotations, the velocities of the wheels were not an issue and final position was reached. Since the PID control operated on errors in the position, all rotations were successful. For translations, however, the unsensed velocities
Figure 2.8: A top-down view of the physical task space of the in-home scenario used to validate the motion planning algorithms using the h+lamp robot hardware. The start location is shown in the top right in green with the goal location shown in the bottom left in yellow. The configuration space obstacles are shown in blue.
Figure 2.9: A series of top-down views showing the h+lamp moving through the physical task space of the in-home scenario.
of the wheels matter significantly. If one wheel moves faster than the other, there will be slight drift in the path towards the slower wheel. While the PID controller will force the positional error to 0 for both wheels, this occurs at different times. The faster wheel stops while other keeps moving. As a result, the robot ends up in the wrong position, and since all the planning happens offline, the system is unable to sense and correct for the error and does not update its trajectory as a result. As the errors accumulated, eventually the robot collided with obstacles, (Fig. 2.9(c)) or left the work space (Fig. 2.9(b)).

Another significant cause of error occurred due to the height mis-match between the active drive elements (the wheels connected to the motors) and the passive drive elements (the ball-bearing casters). In order to improve balance and stability, we added two passive casters to the bottom of the h+lamp frame. We designed the casters to be the same height as the wheel assembly, but due to the uneven surface of the task space floor, situations arose where the passive elements maintained contact with the floor while the wheels were "lifted" enough to cause slipping. To correct this, we significantly reduced the passive element height, thereby sacrificing four points of contact with the floor for three, but guaranteeing that the wheels would always maintain contact with the floor. A by-product of this modification was that robot would "rock" back and forth around the wheel axis when it accelerated. This caused further drift when executing the path from the RRT.

Our first attempt at validating the motion planning algorithms for the mobile base was successful in the sense that robot received the commands and was able to serially execute them. Due to hardware limitations and errors in the closed-loop control, the robot did not accurately reach the goal. However, we showed that the motion planning algorithms we developed allow the mobile base of the h+lamp to move through the task space. In the next Chapter, we describe work done to fully upgrade the h+lamp to reduce the hardware issues seen in this Chapter.
Chapter 3

Third Generation Continuum Robotic Mobile Lamp: CuRLE

Given the problems observed during experimentation discussed in Chapter 2, we constructed brand new robot hardware and installed upgraded features to solve the issues we experienced, as well as to expand the capabilities of the robot. We named this third generation robot CuRLE: Continuum Robotic Lamp Element, shown in Fig. 3.1. This chapter describes our motivation for, and the details of, the extensive upgrades to produce the new hardware.

3.1 Structural Overview

The structural hardware of CuRLE is comprised of three main components: the base frame, the center structure, and the continuum backbone. The base frame, shown in Fig. 3.2 is built from aluminum beams and 3D-printed plastic corner pieces. The frame is square with rounded corners. Mounted below the frame is a differential drive system, described in section 3.3.2. Mounted inside to the bottom of the frame is a turntable mechanism,
Figure 3.1: Third generation continuum robotic mobile lamp (CuRLE) with internal LED stip lit.
described in section 3.3.1. Mounted on the turntable is the center structure.

The center structure is comprised of a 3D-printed plastic base plate, 3D-printed walls, and a 3D-printed top plate. The linear actuator (the only vestigial hardware from the h+lamp) is mounted to the base plate along with the bottom acrylic electronics plate (Fig. 3.5(b-c)). The walls attach to plastic struts that connect to the top and bottom plates. Small protruding shelves can be attached to the inside of each wall, which support the top acrylic electronics plate (Fig. 3.5(a)). All of the motor assemblies, described in section 3.3.3, are mounted to the top plate of the center structure (i.e. the motor plate).

The continuum element is mounted at the end of the linear actuator and is comprised of a PEX backbone (same material, but wider diameter than used for h+lamp’s backbone) and new 3D-printed vertebrae. The actual ”lamp” features of CuRLE are comprised of LED’s mounted to the final vertebra (i.e. the end-effector) and an LED strip that runs through the backbone. These lighting features are described in section 3.3.5. Also mounted to the end-effector is a gripper, detailed in section 3.3.4. The final subsystem is the power system, described in section 3.3.6, which is mounted to the base of the frame.

3.2 Electronics Upgrades

To enable the hardware to execute as intended for the work in this thesis, the electronics of the h+lamp were necessarily upgraded. The Arduino Mega was replaced with the Arduino Due, which has more computational power and higher clock speeds. The faster clock is needed to enable the encoder counter chips, which replaced the old method of directly reading the encoders via interrupts on the Arduino. The new chips allow the Arduino to focus its execution on other tasks (it no longer has to constantly handle interrupts from the encoders) and reduce the number of digital pins required to interface with the motors.

To replace the Wi-Fi shield of the h+lamp, and eventually move CuRLE to a fully
Figure 3.2: The full base of CuRLE.
stand-alone unit, a Raspberry Pi was installed to serve as the "central computer" for the robot. Via remote wireless access, this computer serves as the connection point for the robot and relays external commands to the Arduino via a hardware serial connection. The Raspberry Pi can also relay traffic from the Arduino, enabling two-way communication which was non-existent on the h+lamp. Figs. 3.3 and 3.4 show the electronics as they are mounted within CuRLE.

All of the electronics, excluding those mounted to the continuum element and those mounted beneath the center structure, are attached to the two acrylic plates shown in Fig. 3.5(a-c). The base plate holds the power electronics and the two computing devices (Raspberry Pi computer and Arduino Due micro-controller). The top plate holds the encoder counter chips, relay switches, and dual H-bridge motor drivers (same type as those in h+lamp). The top plate also contains all the connection points for every motor (tendon, turntable, and drive).

### 3.3 Details of the Hardware Upgrades

In the following sections, we describe the different hardware components of CuRLE.

#### 3.3.1 Turntable

Since CuRLE needed to possess the capacity to autonomously execute the output from motion planning algorithms developed for the continuum element (detailed later in Chapter 4), we sought to fully separate the continuum c-space from the c-space of the mobile base by adding a revolute joint at the base of the lamp. This joint allows the continuum element to rotate independent of the movement of the mobile base. This added redundancy creates new configuration opportunities for CuRLE.

The revolute joint, (variable $\omega$), is realized by a worm drive run by DC gear-motor
Figure 3.3: The central structure of CuRLE that houses all of the electronics is shown with the motor plate removed.
Figure 3.4: The assembled center structure with different electronic components visible. (a) The Raspberry Pi. (b) The Arduino Due (c) Voltage regulator (bottom) and motor drivers (top)

Figure 3.5: The two main electronics plates removed from the central structure. (a) Top plate. (b) Left-half and (c) right-half of bottom plate.
Figure 3.6: (a) The turntable mechanism of CuRLE. (b) Zoomed in view of the connection between the worm drive and the turntable.
Figure 3.7: The central structure mounted on the turntable mechanism.
with an optical encoder. The worm drive is mounted to the base of the frame such that the axis of rotation of the worm gear is perpendicular to the plane of the floor and runs through the center of CuRLE (i.e. the axis of rotation is the central z-axis of CuRLE). The worm drive is attached to a turntable which consists of concentric metal discs attached by ball-bearings to facilitate rotation. The turntable mechanism is shown in Fig. 3.6. The outer ring of the turntable is fixed to the frame, and the central structure of CuRLE is mounted to the inner ring (Fig. 3.7). As such, when the worm drive actuates, the entire center structure, including the continuum element, rotates around the central z-axis of the robot.

Since the drive motors and the two of the three power supplies do not rotate with the turntable, we used coiled wire to maintain the electrical connections as \( \omega \) varies. Since these wires still have a maximum extension before disconnecting, the turntable is restricted to a set range of rotation, which is discussed further in Chapter 4.

### 3.3.2 Differential Drive Subsystem

To remove the “rocking” behavior of the h+lamp’s drive system, spring-loaded shocks were designed and built for the passive drive elements (ball-bearing casters) and the active drive elements (differential drive motors). These are shown in Fig. 3.8 and Fig. 3.9 respectively. The shock system serves to remove the “rocking”, and also to enable CuRLE to navigate uneven terrain. Since CuRLE’s operating environment is the home of the user, we envision varying floor surfaces (e.g. carpet, tile, wood) and potential obstacles (e.g. clothing, children toys). With the suspension, CuRLE will successfully maintain the required contact with the floor to navigate the space.

The assembly is composed of two 3D-printed pieces: one (top) which mounts to the frame and one (bottom) which attaches to the drive element (both passive and active). Metal rods are attached to the bottom piece with extremely strong adhesive. The springs
Figure 3.8: (a) Top-side view of the passive drive element showing the through-holes for the metal rods that provide stability and alignment in the shocks. (b) Side view of the shock assembly mounted to the ball-bearing casters (passive drive element).

are then threaded into grooves in the bottom plastic such that the metal rods run through the spring. Plastic caps are threaded to the other end of the springs. These caps have linear ball-bearing bushings attached to the other end and the metal rods then run through these bushings. The plastic caps with bushings are then inserted into a groove into the top plastic and the same adhesive (the grey substance in Fig. 3.8 and Fig. 3.9) is used to weld the caps to the top piece. Holes run through the top piece for the metal rods to pass through as the springs compress. The springs were selected such that the estimated weight of the robot (50 lb) would compress the springs to half of their total displacement (1 in), thereby allowing the shocks to increase/decrease equally to traverse uneven terrain.
3.3.3 Tendon Motors

To simplify the continuum kinematics, we modified CuRLE to a 4-tendon continuum section compared to the 3-tendon design in h+lamp. The new kinematics are detailed in section 3.4. With four tendons instead of three, control of only $u$ and only $v$ is possible by actuating a single pair of tendons. This, in addition to new revolute variable $\omega$ detailed in section 3.3.1, allows us to easily restrict $v$ to be constant (specifically $v = 0$) and still achieve a desirable workspace. The use and advantage of this will be discussed in Chapter 4.

The hardware to attach the motors was upgraded from that in the h+lamp to account for the increased motor count. In addition, we modified the spools the tendons wind around to more accurately measure and track the length of the tendon. We also added a tension sensing subsystem to prevent slack from developing in the tendons. Each motor assembly, consisting of a DC gear-motor with optical encoder, 3D-printed mount, 3D-printed spool,
tension sensor, and 3D-printed tension sensor mount, is mounted at a 90° offset from its neighbors and is shown in Fig. 3.11(a). All four motor assemblies are attached to the top plate of the center structure, shown in Fig. 3.10.

3.3.3.1 Tension Sensors

Because the kinematics rely on accurately tracking the length of the tendons, it is essential that the tendons do not develop slack. Change in tendon length is calculated from the change in the motor encoder count based on the current circumference of the tendon spool. Slack in the tendon causes the amount of tendon spooled to be less than the amount reported by the motor encoder, thereby forcing the controller to evolve the continuum element into a shape that is not accurately defined by its current configuration.

The tension sensor consists of a spring-loaded linear potentiometer mounted inline with the tendon. The tendon is threaded through an enclosed ”spool”, shown in Fig. 3.11(b), that is attached to the end of the tension sensor. Tension in the tendon compresses the potentiometer, and slack allows the it to extend. By keeping the ”tension” reading above a suitably selected threshold, a controller keeps slack out of the tendon and enables accurate changes to CuRLE’s configuration.

3.3.3.2 Tendon Spools

To accurately a measure change in tendon length, we measure how much of the tendon has wound/unwound from the spool by calculating the change in the motor’s encoder counts. Knowing the ratio of counts measured ($\Delta E_{cw}$ as determined by the encoder counter chips mentioned in section 3.2) to counts per revolution ($C_{PR}$) of the motor shaft and the
Figure 3.10: The motor plate which is mounted to the top of the central structure. Here is where the tendon motors are mounted with the tension sensors and the spools to wind the tendons and measure length.
radius of the spool ($r_{spool}$) gives Eqn. (3.1) for the change in length ($\Delta l$).

$$\Delta l = \left( \frac{\Delta E_{cnt}}{C_{PR}} \right) \ast \left( 2\pi r_{spool} \right)$$  \hspace{1cm} (3.1)

As the tendon winds/unwinds around the spool, $r_{spool}$ increases/decreases. To accurately "track" $r_{spool}$, we designed the spool groove to be the width of the tendon (shown in Fig. 3.11(c)), which guarantees, in theory, that every revolution of the spool increases the radius by exactly the width of the tendon (1mm in the case of CuRLE). By tracking the revolutions of the spool (via the encoder counts) we can thereby accurately measure length and thereby achieve valid configurations.

### 3.3.4 End-Effector

The gripper of the h+lamp was limited to open-loop control due to the lack of feedback on its controlling motor. As such, the motor would often continue to "grasp" after the object was already acquired, causing the plastic joints to jam and eventually snap. The
new gripper’s "grasping" mechanism is controlled by a servo motor with built-in position control. By varying the duty cycle of the controlling PWM signal, this gripper will evolve its configuration and hold a specified position.

The new gripper, seen in Fig. 3.12(a-b), also contains second servo motor (identical to the first) that functions as a "wrist" for the end-effector. The wrist (variable \( \gamma \)) enables the gripper to grasp items in configurations that its predecessor could not. The nature of continuum element kinematics allows for a wide set of end-effector Cartesian locations, but few in this set orient the end-effector in a convenient way to grasp objects. The ability to rotate the end-effector via the wrist increases the set.

The final upgrades to the gripper are flexible "finger" attachments that enhance grasping. The "fingers" are 3D printed using a flexible thermoplastic polyurethane material that wraps around an object when the claw grasps it. These "fingers" are shown in Fig. 3.12(b).

### 3.3.5 Lighting Elements

Fig. 3.12(b) shows the LED’s mounted on the end-effector to realize CuRLE’s function as a lamp. There are eight 9mm white LEDs connected in parallel with a 5V source. A digital switch controlled by the Arduino Due toggles the LED’s. In addition, an LED strip was run down the center of the PEX backbone to provide more lighting to the user. The lit LED strip is shown in Fig. 3.1.

### 3.3.6 Power Subsystem

Since CuRLE now possess more actuators and electronics than its predecessor, we installed additional power supplies to robot. The 4S LiPo battery used to power the motors in h+lamp (see Chapter 2) was replaced with a 6S LiPo with a greater capacity. The same
Figure 3.12: (a) The gripper mounted as CuRLE’s end-effector with the LED lights lit and the flexible grasping ”fingers” removed. (b) The gripper with the ”fingers” added.
12V switching voltage regulator provides a constant DC supply to the drive motors, linear actuator, tendon motors, and worm drive motor. The 4 AA batteries used to power the electronics in h+lamp were replaced with a 4S LiPo battery to power the new electronics in CuRLE. This supply is passed to two separate voltage regulators, one that provides 12V and one that provides 5V, to power the Arduino Due and Raspberry Pi, respectively. Finally, we installed a third 4S LiPo battery to power the new LED strips and the servo motors of the gripper. This battery is also connected to a 12V (LED strip) and a 5V (LED’s and servo motors) voltage regulators. Fig. 3.13 shows the power supplies of CuRLE.

3.4 Kinematics

We next present the kinematics for a single section 4-tendon continuum section. While these equations hold for the more general case of an extensible section, the arc length \( s \) of the continuum element remains fixed in CuRLE. The tendons are arranged such that tendons 1 and 3 (variables \( l_1 \) and \( l_3 \)) are an opposing pair in the \( v \)-plane \( (u = 0) \) and tendons 2 and 4 (variables \( l_2 \) and \( l_4 \)) are an opposing pair in the \( u \)-plane \( (v = 0) \).

\[
\begin{align*}
l_1 &= s + (-d) \cdot v \\
l_2 &= s + (d) \cdot u \\
l_3 &= s + (d) \cdot v \\
l_4 &= s + (-d) \cdot u
\end{align*}
\]
Figure 3.13: The power system of CuRLE. (top) 4S LiPo battery to power the electronics. (middle) 4S LiPo battery to power the LEDs, LED strip, and servo motors in the gripper. (bot) 4S LiPo battery (later replaced with a 6S) to power all of the motors.
\[
\begin{bmatrix}
  l_1 \\
  l_2 \\
  l_3 \\
  l_4 \\
\end{bmatrix} = 
\begin{bmatrix}
  0 & -d & 1 \\
  d & 0 & 1 \\
  0 & d & 1 \\
  -d & 0 & 1 \\
\end{bmatrix} \cdot 
\begin{bmatrix}
  u \\
  v \\
  s \\
\end{bmatrix}
\] 

(3.6)

\[
u = \frac{l_2 - l_4}{2d}
\] 

(3.7)

\[
v = \frac{l_3 - l_1}{2d}
\] 

(3.8)

\[
s = \frac{l_1 + l_3}{2} = \frac{l_2 + l_4}{2} = \frac{l_1 + l_2 + l_3 + l_4}{4}
\] 

(3.9)

\[
\begin{bmatrix}
  u \\
  v \\
  s \\
\end{bmatrix} = \left(\frac{1}{4d}\right) 
\begin{bmatrix}
  -2 & 0 & 2 & 0 \\
  0 & 2 & 0 & -2 \\
  d & d & d & d \\
\end{bmatrix} \cdot 
\begin{bmatrix}
  l_1 \\
  l_2 \\
  l_3 \\
  l_4 \\
\end{bmatrix}
\] 

(3.10)
Chapter 4

Configuration Space of CuRLE

This chapter starts by defining for the first time the general configuration space for a single section extensible continuum element [25]. We identify several interesting and unique practical aspects of continuum section C-space, particularly in the case of tendon-actuated sections. To highlight these unique aspects of continuum robots, we compare to this configuration space with that of a rigid-link robot that is selected to have a similar task space. We then discuss the constraints to the continuum configuration space that are imposed by the physical construction of CuRLE. Finally, we discuss the configuration space of CuRLE’s mobile base, and how we can isolate it from the configuration space of the continuum section by making suitable assumptions about the task space.

4.1 Continuum Configuration Space

We begin by considering the underlying structure of continuum robot configuration space in the presence of physical and actuation constraints. Specifically, we consider the c-space of the basic element of continuum robots: a single extensible section. We assume the section to be of constant curvature, a common assumption in the literature [4].
4.1.1 Single Section Continuum Robot

A single section extensible continuum robot, such as the one in Fig. 4.1, has three Degrees of Freedom (DoF) and can be described by three kinematic variables: \{u, v, s\} where is \(s\) is the arc-length of the section and \(u\) and \(v\) represent the components of a rotation axis with respect to the base of the section [26]. Fig. 4.2 illustrates this. Let \(c \in C_{\text{space}}^3\) be a configuration in the configuration space of the robot where \(c = [u \ v \ s]^T\).

To better envision \(C_{\text{space}}^3\), let us first consider the configuration space \(C_{\text{space}}^1\) where only \(u\) varies. We thus restrict the length \(s = s_{\text{fixed}}\) and set \(v = 0\). A configuration is then defined as \(c = [u] \in C_{\text{space}}^1\). As we vary \(u\) in the positive direction (equating to counterclockwise rotation), the section will bend to the left in the \(yz\)-plane, as shown in Fig. 4.3a.
Once \( u = 2\pi \), the section’s tip will meet the base and form a perfect circle. Continuing to increase \( u \) will cause the robot to “bend within itself” and theoretically it would continue to “encircle” itself as \( u \to \infty \). Increasing \( u \) in the negative direction (clockwise) will cause the same planar motion mirrored across the \( z \)-axis (Fig 4.3b). As \( u \to (-\infty) \), the robot will continue to encircle itself to generate the remaining set of possible planar configurations of the section. Therefore, the configuration space of the robot where only \( u \) varies is \( C^{1}_{\text{space}} \equiv \mathbb{R} \).

If we remove the restriction on \( v \) and allow it to also vary, then the configuration space changes to a 2D space where any configuration is defined as \( c = [u \ v]^T \in C^{2}_{\text{space}} \). The
total “bend” of the robot, $\theta$, in the plane of curvature is defined by [26].

$$\theta = \sqrt{u^2 + v^2} \quad (4.1)$$

When $\theta = 2\pi$, the section once more forms a perfect circle, with its tip touching the base. Varying the vector $[u \ v]^T$ generates all bending directions (planes of curvature) and increasing the magnitude of the vector generates all possible configurations in each of these planes via (4.1). Since $u, v \in \mathbb{R}$, $c^2_{space} \equiv \mathbb{R}^2$ and can be visualized as the infinite plane described in Fig. 4.4.

If we now allow $s$ to vary as well, a configuration is defined as $c = [u \ v \ s]^T \in c^3_{space}$. Since $s \in (0, \infty)$, then $c^3_{space} \in \mathbb{R}^3$ s.t. $s > 0$. Fig. 4.5 illustrates this space.
Figure 4.4: A visualization of $C^2_{\text{space}}$. The blue plane extends to $\pm \infty$. The red circle indicates $C^2_{\text{space}}$ with the physical constraint of $\theta \leq 2\pi$.

Figure 4.5: A visualization of $C^3_{\text{space}}$. The dotted lines bordering the solid and the arrows indicate that $u, v \rightarrow (\pm \infty)$ and $s \rightarrow (+\infty)$. The prism is bounded by the plane $s = 0$. 

42
4.1.2 Physical Constraints in a Single Section Continuum Robot

At this point, however, we introduce constraints in the configuration space imposed by physical limitations of the robot. The first constraint to consider is on length. Any physical robot will have a maximum and minimum length, imposing an upper and lower bound on arc-length: \( s_{\text{min}} \leq s \leq s_{\text{max}} \).

Another constraint is imposed by the physical width of the backbone of tendon actuated continuum robots. This physical distance, \( d_r > 0 \), is the distance from the center of the backbone to its outer edge. With \( s \) the length down the exact center of the backbone, and \( L_1 \) and \( L_2 \) the tendon lengths along its outside in the plane of bending (Fig. 4.6), when the robot is perfectly straight (i.e. \( u = v = 0 \)), then \( L_1 = L_2 = s \) (Fig. 4.6a). For the robot to bend counter-clockwise in the plane, the length of the left side of the robot, \( L_2 \), must shorten at the same rate that the length of the opposite side of the robot, \( L_1 \), lengthens. This is illustrated in Fig. 4.6b.

Because of this, the section cannot bend at all when it is at maximum or minimum length. When \( s = s_{\text{max}} \) and \( u = v = 0 \), then \( L_1 = L_2 = s_{\text{max}} \). To bend, \( L_1 \) or \( L_2 \) must lengthen, but each is already at the maximum length. The same reasoning is applied when \( s = s_{\text{min}} \). At maximum/minimum length, \( C_{\text{space}}^3 = \left\{ \begin{bmatrix} 0 & 0 & s_{\text{max/min}} \end{bmatrix}^T \right\} \). For intermediate values of \( s \), a similar situation holds —bending can occur up to one tendon achieving max length. The practical configuration space can now be visualized in Fig. 4.7(a). When \( s = \frac{s_{\text{max}} - s_{\text{min}}}{2} \) the robot will be able to achieve the greatest amount of bending and will have the largest “\( uv \)-plane”.

This pyramid shape assumes there are a pair of opposing tendons in the \( u \)-plane and another pair of opposing tendons in the \( v \) plane, which is consistent with the CuRLE hardware. The flat surfaces of the pyramid reflect this alignment, meaning that if the pairs of tendons were rotated to align in other planes, the pyramid would rotate such that the flat
Figure 4.6: An illustration of physical constraints of bending a continuum robot. In (a), $L_1 = L_2 = s$. In (b), the robot has bent counter-clockwise, causing $L_2$ to lengthen and $L_1$ to shorten, while $s$ remains constant.

surfaces would face the tendon directions.

A further practical constraint arises due to “encircling” imposed when $\theta > 2\pi$. Given a specific physical construction of a practical continuum section, it is likely that it will not be able to “encircle” itself. Even if it could, it cannot continue doing so as $u, v \to (\pm)\infty$. Therefore, there will exist some boundary for $u$ and $v$ imposed by physical constraints. In this thesis, and consistent with our hardware, we set this boundary to be at $\theta = 2\pi$, which bounds $C^2_{space}$ as shown in Fig. 4.4. Expanding the $\theta \leq 2\pi$ constraint to $C^3_{space}$ gives the space seen in Fig. 4.7(b), which is the practical configuration space for a single section extensible continuum robot with physical constraints exploited herein.
4.2 A Comparison: Equivalent Rigid-Link Robot Configuration Space

To highlight the unique issues presented by continuum section structures, we compare with the case of a kinematically similar rigid link robot.

4.2.1 Equivalent Rigid Link Robot

To analyze a rigid link robot structure with the same DoF as the continuum robot section, we use for comparison a 3 DoF robot (Fig. 4.8) with a constrained RRPRR joint configuration similar to the planar RPR robot described in [27].

In this, we constrain the third and fourth revolute joint angles to exactly match the first and second revolute joint angles, respectively, which gives the configuration vector $q = [\theta_1 \ \theta_2 \ d \ \theta_1 \ \theta_2]^T$. The prismatic joint can extend/retract between known maximum and
Figure 4.8: A sketch showing (a) the task-space-equivalent rigid-link RRPRR robot in the same configuration(s) as (b) the continuum element. This is the result of the kinematic mapping \( F \).

minimum lengths. We select this rigid link configuration since we can construct a kinematic mapping between its configuration space and the configuration space of the continuum robot section that restricts the rigid-link robot task space to the equivalent task space of the continuum section. This mapping, \( F \), is described in section 4.2.4. Fig. 4.9 shows the two robots sharing the same task space.
4.2.2 Rigid-Link Robot Configuration Space

Figure 4.9: The task space of both the continuum section (black) and rigid link structure (green) for different values of $u$ for the continuum section and $[\theta_1, d]$ for the rigid-link robot.

Figure 4.10: $Q^2_{\text{space}}$ of the rigid-link robot. The arrows indicate the “wrapping” phenomenon that occurs when $\theta_1$ and $\theta_2$ go beyond the bounds $[0, 2\pi)$. 
Figure 4.11: A visualization of $Q^2_{\text{space}}$ of the rigid-link robot to show the “wrapping” phenomenon. A configuration $q$ is any point on the surface. Changing $\theta_1$ “rotates” $q$ around the axis, $a_c$, running through the center of the torus. Changing $\theta_2$ “rotates” $q$ around $a_t$ which is the tangent to the path of rotation of $\theta_1$.

For the two independent revolute DoF: $\theta_1, \theta_2 \in [0, 2\pi)$ we define $q = [\theta_1 \ \theta_2]^T \in Q^2_{\text{space}}$. Rather than the infinite plane in Fig. 4.4, the space manifests as a square with a “wrapping” phenomenon that causes $\theta_1, \theta_2 \geq 2\pi, \forall \theta_1, \theta_2 < 0$ to “wrap” back to $0 \leq \theta_1, \theta_2 < 2\pi$, as seen in Fig. 4.10. This space, while locally 2D, is globally best visualized as the surface of a torus, like that in Fig. 4.11.

Adding the prismatic joint modifies the ”square” in Fig. 4.10 into the ”rectangular prism” shown in Fig. 4.12. As with the $Q^2_{\text{space}}$, the same “wrapping” phenomenon occurs whenever one of the joints goes beyond 0 or $2\pi$. The configuration space can be defined as $\forall q = [\theta_1 \ \theta_2 \ d]^T \in Q^3_{\text{space}}$ s.t. $0 \leq \theta_1 < 2\pi, \ 0 \leq \theta_2 < 2\pi, \ d_{\min} \leq d \leq d_{\max}$.
Figure 4.12: $Q_{space}^3$ of the rigid-link robot. The arrows indicate the “wrapping” phenomenon that occurs when $\theta_1$ and $\theta_2$ go beyond the bounds $[0, 2\pi)$.

### 4.2.3 C-Space of Continuum Section vs Rigid-Link Structure

The key difference between the continuum and rigid-link configuration spaces (c-space) is the “wrapping” phenomenon that occurs in $Q_{space}^3$. In the ideal continuum c-space, there exists exactly 1 straight path connecting any two configurations $c_1, c_2 \in C_{space}^3$. For the rigid-link robot, there are always 2 straight paths between any configurations that involve a change in $\theta_1$ or $\theta_2$. For configuration changes that exclusively involve the prismatic joint, there is exactly 1 path.

An interesting difference between the c-spaces is their sizes. Since both $C_{space}^3$ and $Q_{space}^3$ are finite their sizes can be compared by calculating the volume of each space. This can be done by calculating the volume of the 3D shapes shown in Fig. 4.7(b) and Fig. 4.12.
The size of $Q_{space}^3$ is the volume of the rectangular prism. The size of $C_{space}^3$ is the volume of the “rounded” pyramid, which is slightly less than the full 2-sided square pyramid but greater than a 2-sided cone where the base has radius $2\pi$.

$$
(4\pi^2)(s_{max} - s_{min}) \left(\frac{\pi}{3}\right) < V_{cont} < (4\pi^2)(s_{max} - s_{min}) \left(\frac{8}{3}\right)
$$

$$V_{rigid} = (4\pi^2)(d_{max} - d_{min})
$$

Since the rigid-link robot was chosen to be kinematically similar to the continuum robot, we can conclude that the configuration space for a kinematically equivalent continuum section is larger than the rigid-link robot’s configuration space (Eqn. 4.3).

$$s_{max} = d_{max} , \ s_{min} = d_{min}$$

$$\Rightarrow V_{rigid} < V_{cont}
$$

4.2.4 Ensuring Equivalent Task Spaces

The only difference between the two task spaces is the physical shape of the arm of the robot that creates them. This is either a constant curvature curve between the base and the end-effector (continuum) or a straight line (rigid-link), as shown in Fig. 4.9. This shape, however, is important in the context of motion planning, as it has to be accounted for when checking for collision-free paths through the space.

For the rigid-link robot to have the same task-space as the continuum robot, we construct a function $F$ that maps every configuration $c \in C_{space}^3$ to a configuration $q \in Q_{space}^3$. This function, $F$, is neither one-to-one nor onto, and is shown in Eqn. (4.4).
\[ F : C^3_{\text{space}} \rightarrow Q^3_{\text{space}} \text{ s.t. } F(c) = q \text{ where} \]

\[
c = \begin{bmatrix} u \\ v \\ s \end{bmatrix} \in C^3_{\text{space}} \text{ and } q = \begin{bmatrix} \theta_1 \\ \theta_2 \\ d \end{bmatrix} \in Q^3_{\text{space}}
\]

\[
\Rightarrow F(c) = \begin{bmatrix} \frac{u}{2} \\ \frac{v}{2} \\ \left( \frac{2s}{\sqrt{u^2+v^2}} \right) \sin \left( \frac{\sqrt{u^2+v^2}}{2} \right) \end{bmatrix}
\]

(4.4)

Let \( Q^C_{\text{space}} \triangleq \mathbb{R}\{F\} \) where \( \mathbb{R}\{F\} \) is the range of the function \( F \). Then \( Q^C_{\text{space}} \in Q^3_{\text{space}} \) is a subspace of \( Q^3_{\text{space}} \). Since \( C^3_{\text{space}} \) is centered at \( u = v = 0 \), and has a radius of \( 2\pi \), \( Q^C_{\text{space}} \) will be centered at \( \theta_1 = \theta_2 = 0 \). Now, we restrict \( \theta_1, \theta_2 \) such that \( \sqrt{\theta_1^2 + \theta_2^2} \leq \pi \). This is the effect of applying \( F \) to Eqn. (4.1). If this is not done, then the value of \( d \), as described be Eqn. (4.4), would take on negative values, because of the \( \sin \left( \frac{\sqrt{u^2+v^2}}{2} \right) \) term. This restriction of \( \theta_1, \theta_2 \) also removes the “wrapping” phenomenon and causes \( Q^C_{\text{space}} \) to have the same “rounded” pyramid shape as \( C^3_{\text{space}} \), as shown in Fig. 4.13.

Finally, by comparing (4.2) and (4.4), the volume of \( Q^C_{\text{space}} \) is significantly smaller than the volume of the \( C^3_{\text{space}} \), but the task space is equivalent.

\[
(\pi^2)(s_{\text{max}} - s_{\text{min}}) \left( \frac{\pi}{3} \right) < V_{\text{equiv}} < (4\pi^2)(s_{\text{max}} - s_{\text{min}}) \left( \frac{2}{3} \right)
\]

(4.5)
Figure 4.13: The configuration space (red) of the rigid-link structure, $Q^C_{space}$, once it has been restricted by $F$ to have the equivalent task space as the continuum section. This is displayed within the full c-space (blue), $Q^3_{space}$, from Fig. 4.12.

4.3 Continuum Robotic Lamp Element: CuRLE

Recall from Chapter 3, CuRLE is a tendon-driven, non-extensible, single-section continuum arm mounted onto a mobile base which is controlled by a differential drive. CuRLE ’s end-effector is a 2-fingered gripper featuring a series of LEDs to give the lamp light. Additional lighting is provided by LED strips inside the continuum body of the lamp.

The continuum arm is mounted on a prismatic joint (variable length $L$) which serves to raise/lower the base of the continuum arm but does not change the continuum arc length $s$. The prismatic joint is further mounted on a revolute joint (variable $\omega$) which allows the entire arm to be rotated about the z-axis (yaw). Another revolute joint (variable $\gamma$) is mounted at the end of the continuum arm to serve as a “wrist” for the gripper.

4.3.1 Adding Constraints

In order to visualize the configuration space of CuRLE and conduct practical motion planning through the home environment space, we constrain several DoF. For the re-
mainder of this work, we fix $L$ to its minimum length. Since the continuum arm is non-
extensible, the arc-length $s$ is necessarily constrained to be constant. The revolute joint $\gamma$
-serving as the wrist for the gripper adds redundancy, but remains fixed in the experimenta-
tion described here.

With these restrictions, we discuss the mobile base and the kinematic variables
$[\omega \ u \ v]$ for the continuum arm. Since the base serves to move the continuum lamp ele-
ment through the home space and the continuum element performs manipulation, we di-
vide the configuration space into two parts. We assume that there will not be obstacles
that CuRLE has to “pass under” meaning that nothing in the task space would collide with
the continuum arm but not the mobile base. As such, we form the configuration space of
the continuum arm $c = [\omega \ u \ v] \in C_{\text{space}}$ and the configuration space of the mobile base
$q = [x \ y \ \theta_b]^T \in C^\text{base}_{\text{space}}$.

### 4.3.2 Configuration Space of CuRLE

Recall that for a fixed arc-length $s$, a single section continuum robot has a practical
configuration space of a circle bounded by $\theta = 2\pi$, shown in Fig. 4.4. Since the continuum
arm of CuRLE can physically collide with the mobile base, we further modify the boundary
to $\theta \leq \pi\sqrt{2}$ (the value of the $\theta$ when $u = v = \pi$).

The revolute joint at the base of the continuum arm is described by $\omega \in [0, 2\pi)$. In
the ideal case, $\omega$ displays the same “wrapping” behavior that the revolute joints in the
rigid-link configuration space. As such, adding $\omega$ changes the configuration space to 3
dimensions by revolving the “uv-circle” around a central axis. The shape, depicted in Fig.
4.14, echoes the torus described by $Q^2_{\text{space}}$, the c-space of 2 revolute DoF. Unlike $Q^2_{\text{space}}$,
however, CuRLE ’s is “solid”, i.e. true 3D, meaning that configurations $c$ are not limited to
the surface of the torus only. $\omega$ selects the “slice” (a circle) of the torus and $u$ and $v$ select
the point $c$ within that circle.

Due to physical constraints, however, we limit $\omega \in [-\pi, \pi)$ and do not allow wrapping, which represented in Fig. 4.14 by the solid black plane at $\omega = -\pi$. Configurations take on any value in the volume except those in the black plane.
4.4 Configuration Space of Mobile Base

Since we separate the configuration space of the continuum element from the C-space of the mobile base by making the assumptions described above, we can formally define the configuration space of the mobile base \( C_{\text{base}} \) in (4.6) where \( q \) is a configuration of the robot in the configuration space \( C_{\text{space}} \), \( x \) and \( y \) are the positions of the robot along the \( x \)-axis and \( y \)-axis respectively, and \( \theta_b \in [0, 2\pi) \) is the orientation of the robot with respect to the positive \( x \)-axis.

\[
\forall q \in C_{\text{base}} : q = [x \ y \ \theta_b]^T \quad (4.6)
\]

The ideal configuration space (i.e. no physical constraints) has the shape of a rectangular prism with an infinite base (i.e. the \( x,y \)-plane) and a height, \( \theta_b \in [0, 2\pi) \), of \( 2\pi \). In any practical application, however, the parameters \( x \) and \( y \) will always be bounded by the walls of the room.

Obstacles in this configuration space are described as the Minkowski difference of the robot and the obstacle in the task space [11]. All the obstacles to the mobile base used in our experiments were designed to be convex polygonal in shape, since collision-avoidance can often be simplified by drawing a “bounding-box” around the obstacle. The base of CuRLE is a rectangular box with rounded corners (Fig. 4.15) meaning that the obstacles would resembled “twisted pillars” in the configuration space [13]. To again simplify the design, the robot is treated as a disc with a diameter equal to the diagonal of the frame. With the robot as a disc, collision detection can disregard \( \theta_b \) and simply check for collisions in the \( x,y \)-plane [10].

For the robot to move between two configurations, \( q_0 \) and \( q_1 \), an action vector \( \mu \) is
Figure 4.15: CuRLE’s base is a square frame with rounded corners. The radius of the "disc" used to estimate the base of CuRLE is shown in green.
The \( \min \text{rot}(\varphi) \) function in (4.7) returns the minimum rotation needed to move between two angles. If the calculated rotation is greater in magnitude than \( \pi \) radians, then a rotation in the opposite direction provides an overall shorter rotation. (i.e. a counterclockwise rotation might be shorter than rotating in a clockwise direction and vice versa).

The final aspect of defining the configuration space is a metric \( d(q_0, q_1) \) that defines the “distance” between the two configurations \( q_0 \) and \( q_1 \) [13]. For CuRLE, the simplifications made to the configuration space allow the metric \( d(q_0, q_1) \) to be defined as the Euclidean distance (L2-norm) between the x and y coordinates of the \( q_0 \) and \( q_1 \).

With the configuration spaces of the continuum element and the mobile base defined, we now have the foundation to discuss the motion planning algorithms we developed to navigate these spaces.
Chapter 5

Motion Planning

In this chapter, we introduce new motion planning algorithms specifically created for CuRLE and simulations used to test and verify them. First, we discuss the RRT used for the mobile base of the robot (Chapter 2), using the configuration space discussed in Chapter 4. Next, we detail the simulation environment for the mobile base used to test the algorithms’ output. This is the simulation from the discussion in Chapter 2 which was used to verify the RRT before implementing on the h+lamp hardware.

We then discuss the necessary modifications made to the RRT to path plan for the continuum arm element of the robot. The new simulated environment developed to test this second algorithm is also described. Finally, we will show how the two RRTs can be combined to run in parallel and provide the results for those (simulated) experiments. The physical implementations corresponding to these simulated tests will be discussed in Chapter 6.
5.1 Path Planning for the Mobile Base

5.1.1 Rapidly-Exploring Random Tree (RRT) Algorithm

An RRT is a probabilistic approach to motion planning [11]. The objective is to connect a pre-defined start configuration with a desired goal location by constructing a tree of randomly selected nodes from the configuration space. The RRT is an iterative process that rapidly expands to cover the free-space of an environment. At each iteration, a random configuration is sampled and connected to the tree via the closest node that is currently in the tree, as long as the new node is located in the free-space and the new edge is collision-free. After a $N$ iterations, the goal node is "selected" as the random node and connected to the tree. The start and goal configuration are now connected and a search algorithm (e.g. A*) can be run to determine the best path through the tree. Since the RRT is a probabilistic method, it will always connect to the goal provided that a solution exists and given infinite execution time. Since it randomly samples the free space to generate paths, it has a fast solve time for relatively open free spaces [13]. As such, it is often used as an anytime-approach in dynamic environments.

To implement the RRT, we used the open-source Boost Graph library for C++ [28]. Each vertex of the graph contains a configuration $q$, a node id, a heuristic value, and a cost value. Both heuristic and cost values are later used in the A* search algorithm (see below). The edges of the graph contain the distance between vertices, measured as the Euclidean distance, as well as the action vector, $\mu$, to move between the two nodes. The graph is directed, connected, and contains no cycles. Most importantly, the graph can be reconfigured.

In our design, there are three types of RRTs that can be selected. These types are defined as VERTEX, EDGE, and EXT VERTEX. The VERTEX type is the simplest RRT. As new nodes are randomly generated ($q_r$) and verified as valid configurations (i.e. no
collision with the configuration obstacles), the algorithm will search the graph for the node that is closest to \( q_r \), using the metric \( d(q_0, q_1) \) defined for the configuration space (L2-norm). Once the closest node is found, the two vertices are connected after first checking that the new edge does not pass through an obstacle and second, checking that the new edge is greater than a minimum threshold (passed as a configuration parameter to the RRT). This collision detection is done by sampling along the edge at a fixed \( \delta \), which varies based on the specific configuration variables used. The samples start at both the \( q_{\text{closest}} \) and \( q_r \) nodes and move towards the center of the edge. If any of the samples collide, then \( q_r \) is discarded and the loop continues. If there is no collision, then the edge distance is verified to be greater than the minimum threshold. This is done to prevent unnecessary nodes stacking on each other. If this threshold is met, then \( q_r \) is added to the graph along with the new edge connecting \( q_r \) and \( q_{\text{closest}} \) and the process repeats.

With the EDGE type of RRT, the edges of the graph are also considered when searching for the closest connection point. While this increases the search time, it generates a graph that better covers the free space. This process is done by calculating the normal distance from \( q_r \) to the line that contains each edge. The point of intersection, if it lies on the line segment that defines the edge, will become a new node in the graph \( q_i \). The same process then repeats for collision detection of edges, this time checking the new edge between \( q_i \) and \( q_r \). If all edges are collision free and greater than the threshold, then the old edge is deleted and both \( q_i \) and \( q_r \) are added to the graph, along with the appropriate new edges.

The EXT_VERTEX type of RRT extends the simple VERTEX type by breaking up long edges into multiple nodes and edges along the same path. The algorithm compares the new edge distance to a maximum threshold (passed as a parameter) and will split the edge based on the size. This method prevents the need to check edges, but helps create a wider spread of nodes that cover the free space. Fig. 5.1a-5.1c show a comparison of the three
different types of RRT with same random seed, another configuration of the RRT.

To generate random nodes \( q_r \), a separate uniform random number generator (URNG) is run for each variable of \( q \). The bounds of each URNG correspond to the bounds of each variable in the configuration space (as detailed in Chapter 4). These values are specified to the RRT by a configuration file that is read at run-time. Fig. 5.2 shows that the spread of the RRT evenly covers the free space.

In addition to the parameters already discussed, (RRT type, minimum edge distance, maximum edge distance, random seed) the RRT can be configured in regard to how many nodes are generated. As with all RRTs, after a certain number of iterations of generating random nodes, the goal node is “selected” as the \( q_r \) and the algorithm then attempts to connect the goal to the tree [13]. For all our experiments, the algorithm searched for the goal 4% of the time. There is another parameter that specifies the minimum number of nodes in the tree. Even if the goal connects on the first try, it can be worth running the RRT longer to cover the free space more uniformly and thereby find better paths. The final parameter is the maximum number of nodes to add. This parameter serves to cut off the RRT after a certain point if it cannot reach the goal node. Since the RRT is a sampling based approach, it is only probabilistically complete. With infinite time, the goal will be reached if there exists a path. Rather than halt the RRT after a specified time, our algorithm halts after a maximum number of nodes are added and a flag is set to indicate if the goal was not found.

5.1.2 A*

The output of the RRT is a connected graph of valid configurations for the robot. To evolve from the start to the goal configuration, an A* search algorithm is run on the graph. The A* heuristic function \( h(q) \) is the L2-norm Euclidean distance from \( q \) to \( q_{goal} \).
Figure 5.1: Plots showing the different graphs through the simulated environment for different types of RRTs. The path started at the green node and followed the yellow path to the magenta goal node, avoiding the configuration obstacles defined by the slashed lines surrounding the red obstacles in the task space.
Figure 5.2: Plot showing the RRT (type=VERTEX) expanding into an open environment (no obstacles). The green node indicates the start.
The cost function $g(q)$ is defined to be the L2-norm traveled along each edge from the $q_{start}$ to the $q$. With this cost function, our heuristic function is admissible, following the triangle inequality [13]. An admissible heuristic function always underestimates the cost from any configuration to the goal configuration. Unless the $q_{start}$ is directly connected to $q_{goal}$, the path will consist of at least two edges. Let $q_0 = q_{start}$ and $q_2 = q_{goal}$. Let $g_{0,1}$ be the L2-norm (i.e. cost) from $q_0$ to $q_1$ and $g_{1,2}$ be the L2-norm (i.e. cost) from $q_1$ to $q_2$. While $q_0$ and $q_2$ are not connected, let $g_{0,2}$ be the the cost to travel from $q_0$ directly to $q_2$ (i.e. $h(q_0) = g_{0,2}$). If $q_{start}$ is directly connected to $q_{goal}$ then the heuristic value is equal to the cost, so the admissibility still holds. Therefore, the actual cost to reach the goal is $g_{0,1} + g_{1,2} \leq g_{0,2}$ by the triangle inequality. This will guarantee that our A* provides the optimum path from $q_{start}$ to $q_{goal}$ given the tree generated by the RRT. If the RRT was not successful, then an error is generated and A* is never run.

The output of the A* is the set of action vectors $U = \{\mu_1, \mu_2, \ldots, \mu_n\}$ to move the robot from configuration to configuration. For the mobile base, each action vector $\mu$ has two rotations and a translation. The robot must first rotate to face the new configurations, translate in a straight line to reach the configuration’s $x$ and $y$ position, then finally rotate to orient itself with the configuration’s $\theta$. The resulting movement of performing $U$ serially results in superfluous rotations. When traveling from $q_0$ to $q_1$ then on to $q_2$, there is no reason for the robot to orient itself with $\theta_{q_1}$, but rather simply orient itself to face $(x, y)_{q_2}$. The set $U$ is passed to a smoothing function Eqn. (5.1) that combines the second rotation with the first rotation of the following action. The $\text{min}_\text{rot}(\phi)$ function is the same as that used in Eqn. (4.7).
smooth \((\phi_i, \phi_{i+1})^T, (\phi_{i+1,1}, \phi_{i+1,2})^T) = \) (5.1)

{} 

\(\phi_i, 2 = \min_rot(\phi_i, 2 + \phi_{i+1,1}) ;\)

\(\phi_{i+1,1} = 0 ;\)

\}

5.1.3 Testing the RRT and A*

To collect data about the RRT and A* performance, we performed two sets of experiments and the results are shown in the tables Fig. 5.3. For each of these experiments, 20 random seeds were chosen. The RRT/A* algorithm was then run with each of these seeds and the results (number of nodes, time, path length, etc.) were averaged (\(\mu\)) and the standard deviations calculated (\(\sigma\)).

Figure 5.3: (a) RRT Experiment 1 Results. (b) RRT Experiment 2 Results.
Within Experiment 1 and 2, the type of the tree (see above) was changed and the same 20 seeds were used again. Between Experiment 1 and 2, the **Parameters** shown in the first row of the tables were varied and the experiment run again.

From the data, we conclude that the EXT_VERTEX is the best type of tree to run. The time of the RRT execution, while almost doubling that of the VERTEX runtime, is orders of magnitude less than the EDGE time. The total distance traveled for the EXT_VERTEX is roughly equivalent to the EDGE type, which is an indication of a more “optimal” path. While the raw **Path Rotation** is greater than EDGE, the optimized path rotation is less in both experiments. Another benefit of the EXT_VERTEX is that the edges of the path are, on average, shorter. This can be inferred from the **Path Length** as measured in number of nodes. Since small errors are magnified over larger distances, the robot has a better chance to maintain course. As the data shows, EXT_VERTEX performs the best for the scenario we have used, and is the method used in all of the simulations in the following sections.

### 5.1.4 Simulation of Mobile Base RRT

To evaluate the RRT approach, a scenario was designed in which the mobile base would move from one side of an environment to the other while avoiding obstacles. Fig.5.4 shows the scenario. In the image, the red areas represent the obstacles in the task space. The thick black border represents the extreme limits of the x-axis and y-axis for this workspace (i.e. the walls of the room). The dashed lines represent the boundaries of the configuration space obstacles.

While the robot is estimated as a disc, meaning the configuration obstacles formally should have rounded borders from the Minkowski differences, the obstacles are simply estimated as the bounding rectangle that encloses the Minkowski difference. This is done
for simplicity in the collision detection. In the scenario, the start position is shown as the green node and the goal is the pink node.

Preliminarily to implementing the path execution on the robot, animations were created in MATLAB to visualize the paths as they are created. Fig. 5.5 shows snapshots of the trees as they are animated, as well as the final RRT with the path generated by the A* algorithm highlighted.

After the central computer runs the RRT and A* algorithms, it generates *.csv files of the graph and path. These files are parsed by a MATLAB script to generate the animations. A different script creates a robot trajectory based on the actions and simulates the robot as it would follow the path through the environment, as seen in Fig.5.6.

5.2 Path Planning for the Continuum Section

5.2.1 RRT and A*

Using the configuration space for CuRLE that we established in Chapter 4, we apply the same classical motion planning techniques to map and navigate the continuum
element c-space as we did for the mobile base c-space.

The same RRT described in section 5.1.1 was used to generate a path for the continuum element. The C++ code was modified so that the configurations in each vertex were continuum configurations \( c = [u \ v \ \omega]^T \in C_{space} \) and every edge is the “action” vector \( \mu \) needed to move from one configuration to the next. Actions in this space are simply the difference in each configuration variable. The start and goal location, along with the location of all configuration space obstacles, were known \textit{a priori} and all of the obstacles remain convex polyhedrons. Following the evaluation described in section 5.1.3, the RRT type \textsc{ext\_vertex} was chosen for CuRLE. All the other RRT configuration parameters (percent of iterations to connect goal, minimum nodes, maximum nodes, etc.) were set to be the same as the mobile base RRT.

Given the unique kinematic constraints of continuum robots identified earlier, the RRT algorithm had to be modified to be applicable to CuRLE. In order to pick up an object, for instance a cup on a shelf, the continuum section has to bend in such a way so that the object ends between the fingers on the gripper. Since the gripper has a fixed maximum width, the room for error is very small. In the configuration space, this means that the goal location is always in a narrow “canyon” created by the configuration space obstacles. For RRT implementations, this can make it very difficult to connect to the goal. We solved this issue by “projecting” the goal node along a straight line until it was out of the “canyon.” Once this projected goal could be attached to the graph, the goal was then achieved moving \( u \) or \( v \) in a straight line.

We solved collision detection and avoidance by taking a sampling based approach. Since the kinematics of the continuum element are complex, manifestation of task space obstacles in the configuration space of a continuum element is complex. Rather than formally defining a mapping between the task space and the configuration space for a generic obstacle, we sampled the task space with our simulated version of CuRLE and recorded all
the configurations of any collisions, which are detected by sampling along the backbone of
the continuum element and checking for a collision at each point. Fig. 5.8 shows the con-
figuration space manifestation of the task space obstacles shown in Fig. 5.7. This offline
sampling method generates a discrete lookup-table for every possible configuration, sam-
pled at a rate of 0.01 between the minimum and maximum values, for each configuration
variable.

Once the RRT algorithm connected the goal configuration to the graph, we ran the
same A* algorithm as for the base to determine the optimal path. As with the mobile
base, the A* used a heuristic of the L2-norm between a given configuration and the goal
configuration. The cost function was the L2-norm between each node in the current path
from the start node. We also implemented a weighting function to encourage movements
of the continuum arm (u and v) and penalize movements of the revolute joint at the base of
CuRLE (ω). This was done to reduce the overall execution time in the physical hardware.
The algorithm then output the full graph and optimal path to a *.csv file.

5.2.2 Simulation Results

To verify the motion planning algorithms, a simulated environment was created in
MATLAB and a model of CuRLE was developed and added to the simulation. The simu-
lation discussed herein involves a scenario where CuRLE is instructed to pick up a cup off
a shelf by the user (see Fig. 5.7). The task space obstacles were converted to configuration
space obstacles, shown in Fig. 5.8. The start configuration of CuRLE was $c_{start} = [0 0 0]^T$
and the goal configuration for the example reported herein was $c_{goal} = [-1.76 0 -\pi/2]^T$,
(i.e. the configuration needed to pick up the cup). This goal was determined by using the
interactive GUI developed with the environment to bend the simulated CuRLE until it was
in the correct configuration. Fig. 5.9 shows the interactive GUI with CuRLE in the start

69
configuration.

The output of the RRT/A* was fed into the simulation and Fig. 5.10 shows CuRLE achieving the goal configuration of grasping the “cup”. For this simulation, we set \( v = 0 \) to keep the configuration space 2D and allow for easy visualization of the nodes from the RRT algorithm.

From the simulation that we ran for both the mobile base RRT/A*, we can predict that our motion planning algorithms will successfully navigate the configuration space to guide the robot through the task space. In the proposed scenario, a path for the mobile base is found rapidly and the simulation shows that the path is collision-free. In addition, we can conclude that our work to design the RRT to be re-configurable and extensible was a success. By simply changing the ”state” definition, we generated a collision-free path for the continuum element through its task space by using its configuration space, which we verified by our simulation of CuRLE.
Figure 5.5: The RRT growth over time.
Figure 5.6: A progression of images showing the simulated robot moving through the scenario.
Figure 5.7: The task space showing the cup from the scenario. The middle cup is the “goal” cup that CuRLE will pick up and deliver to the user.

Figure 5.8: The configuration space obstacles of the task space. The start configuration, is shown in (a), while (b) is the magnified obstacle from (a) where the goal configuration is located. (b) is the c-space obstacle of the shelf shown in Fig. 5.7
Figure 5.9: The interactive GUI that serves as the front end for simulation environment.
Figure 5.10: The simulated CuRLE in the start (vertical) and goal (bent) configuration. The objective of the scenario was to pick up the cup, shown as a light grey prism, on the shelf.
Chapter 6

Validation of Motion Planning with CuRLE Robot Hardware

This Chapter details experimentation done to validate the RRT/A* algorithms developed in Chapter 5 by implementing them on CuRLE. First we describe the software running on CuRLE to implement the autonomous motion given a path through the configuration spaces. Next we describe an initial experiment to validate the RRT output for the continuum element on the CuRLE hardware. We then lay out the over-arching in-home scenario that we used to fully validate the work in this thesis regarding the configuration space and motion planning for a practical hybrid tendon-actuated continuum element/mobile base operating in free-space. The final sections detail the experiments done and evaluate the results.

6.1 CuRLE Software Implementation

The software running on the CuRLE hardware is split between the Arduino Due micro-controller and the Raspberry Pi computer. The Arduino is responsible for interfacing
with all of the actuators (motors) and sensors (encoders and tension sensors) in the system. Implemented in the Arduino IDE (C++), the software controllers running on the Arduino execute the transitions from the current configuration to the next. The Raspberry Pi serves as the central computer of CuRLE and is responsible for interfacing with the Arduino. Primarily running Python scripts, it passes external messages to the Arduino and relays any response. The Raspberry Pi also stores the current state of the robot, for persistence of memory when power is lost. Finally, since the Raspberry Pi can be accessed remotely over the network, we can take direct control of the Pi (and by extension the Arduino) to update code, execute individual commands, and initiate a debugging session.

### 6.1.1 Configuration Controller

Much like the controller of the h+lamp (see Chapter 2), the controller(s) running on the Arduino constantly monitor the current configuration of the robot (i.e. reading the sensors) and maintain a desired configuration by actuating the motors when the current configuration strays from the set point (i.e. the error is nonzero). Two PID controllers generate signals for the motors based on the current error values: one for the mobile base state and one for the continuum element state.

The controller for the mobile base is fundamentally the same as the one discussed for the h+lamp. The method for calculating error signals and set points for each drive motor is as detailed in Chapter 2. The gain values for the PID were tuned differently to work for the new hardware, but the underlying software is the same.

Because of the unique properties of the continuum element kinematics, the controller for the continuum element fundamentally operates differently from the mobile base controller. As mentioned in Chapter 3 knowing the length of the tendons is essential for determining the configuration of the robot. In order to accurately measure the length, the
radius of the spool must be known. As such, the controller must also track the radius of the spool as it changes over time. Because of this, the absolute position of the motor shaft must always be known and saved.

After initial calibration when CuRLE was first constructed, the current state of the motor shaft (measured in encoder counts) is saved to a configuration file on the Raspberry Pi. In addition, the continuum configuration is stored ($[u \ v \ \omega]$). Every minute, the Arduino communicates the state (motor encoder count and the $[u \ v \ \omega]$ corresponding to those counts) to Pi and the Pi updates the configuration file. When the system first boots, the Arduino requests the saved state from the Pi and remains idle until the Pi sends the state. The controller is able to use the "zero" state and the current encoder readings to track the radius of the spools and thereby the lengths of the tendons.

To reach a desired configurations, the same PID controller operates independently on all four tendons. This controller was tuned until the robot could accurately reach a desired configuration from its current configuration.

6.1.2 Execution Sequence

To implement the RRT output on CuRLE, the motion planning was done offline on an external computer. We then transmitted the output files to the Raspberry Pi over the local network. Rather than have the external computer execute this process automatically, we found that experimentation was easier if we manually transferred the files. Once the files were on the Raspberry Pi, we remotely accessed it via SSH and ran an interactive (via command line) Python program that initiated the Arduino. Once the Arduino received the saved state and finished its initializing, we executed a command through the Python program to load the paths (mobile and continuum element) and transmit commands to the Arduino. Once the Arduino reached the set configuration (i.e. all error signals dropped
below a minimum threshold), it sent a message back to the Raspberry Pi to indicate that it has completed its task. In addition, the Arduino flashed the lamp LEDs to visually communicate that the set configuration had been reached. The Raspberry Pi, idle until it received this message from the Arduino, then sent the next configuration in the path. This process repeated until the goal was reached.

6.2 Validating the Continuum Section Controller

As we did with the h+lamp controller, we first tested the continuum controller’s ability to “follow” an RRT path by serially executing the transition from node to node. Recall from Chapter 5 the simulation experiment done to test the RRT output for the continuum element. In simulation, we demonstrated the software model of CuRLE bending to grasp a cup on shelf. We took the same RRT output (shown in Fig. 6.10) and passed this path to CuRLE. We issued “grasp” command to grab the cup and then passed CuRLE a second path from the RRT to lift the cup from the shelf. The results of this experiment are shown in Fig. 6.1.

6.3 In-Home Scenario

To demonstrate the full functionality of the system, we explored a simple in-home scenario where the user has asked CuRLE to fetch a cup from shelf, bring the cup into an adjacent room and set it down on a different shelf, then navigate to its ”docking station” in another portion of the smaller room. The environment for this scenario is shown in Fig. 6.2. This scenario can be further broken down into three sub-scenarios, each involving planning for the mobile base and the continuum element. In each sub-scenario, the path(s) the robot must follow is shown for both the mobile base and the continuum element. We
then show and discuss the results from each experiment.

6.4 Sub-Scenario 1: Simple Base Movement and Grasping the Cup with Continuum Element

In this sub-scenario, the first set of actions moves CuRLE from the start (Location 1) to the goal (Location 2) to position the robot in front of the shelf. Next, the continuum element bends to grasp the cup. A ”grasp” command is then issued and CuRLE lifts the cup from the shelf. Fig. 6.3 shows the path generated by the mobile base RRT, and Fig. 6.4 and 6.5 show the two paths generated by the continuum element RRT.
Figure 6.2: The in-home scenario explored to demonstrate the full functionality of CuRLE. The Locations discussed below are numbered in the image. The first shelf is located at Location 2 and the second shelf is at Location 3.

6.4.1 Results and Discussion

Images from video footage of the experiment are shown in Fig. 6.6, Fig. 6.7, and Fig. 6.8a. The mobile base reached its goal with minimal error, demonstrating the vast improvement of CuRLE’s mobile base performance over that of its predecessor (h+lamp). The effort reported in Chapter 3 to upgrade the robot is thus validated in this situation.

Since the mobile base had minimal error in its position, the continuum element successfully picked up the cup from the shelf. Since there is no external sensor providing feedback to CuRLE, small errors in the position will cause failure when executing the RRT output. For instance, if the base is off by even a small distance, the continuum element will
Figure 6.3: Output of the RRT showing the path required for the mobile base of CuRLE to navigate from the start location to the first shelf.

Figure 6.4: Output of the RRT showing the path required for CuRLE to grasp the cup on the first shelf. (b) is a magnified portion (a) to better show the start and goal configurations.
miss its goal location. This was seen in initial experimentation and resulted from poor PID control. By tuning the gains, we reduced the error enough such that the continuum element was able to grab the cup.

6.5 Sub-Scenario 2: Complex Base Movement and Placing the Cup with Continuum Element

In this sub-scenario, the first set of actions moves CuRLE from the first shelf (Location 2) to the goal (Location 3) to position the robot in front of the shelf. To do this, CuRLE must travel to the ”second” room, which is on the far side of a ”wall” indicated by the long vertical obstacle in the center of the task space. Once in the other room, the continuum element bends to place the cup on the second shelf there. A ”release” command is then issued and CuRLE bends away from the shelf. Fig. 6.9 shows the path generated by the mobile base RRT, and Fig. 6.10 and 6.11 show the two paths generated by the continuum element RRT.
6.5.1 Results and Discussion

Images from video footage of the experiment are shown in Fig. 6.8b, Fig. 6.12, and Fig. 6.13. As with Sub-Scenario 1, the mobile base successfully followed the path and reached its goal location with minimal error. The continuum element successfully held the cup during the entire transportation from the ”start” to the ”goal”. Once again, since the mobile base arrived with small enough error, the continuum element was able to place the cup on the shelf.

6.6 Sub-Scenario 3: Simultaneous Movement Between Base and Continuum Element

In this sub-scenario, the first set of actions moves CuRLE from the second shelf (Location 3) to the goal (Location 4) to position the robot at its ”docking station”. To do this, CuRLE must navigate the tight space in the ”second” room. While moving through this room, the continuum element bends to return to its ”home” configuration. This experiment demonstrates the ability for the robot to execute paths from both RRTs in parallel. Fig. 6.14 shows the path generated by the mobile base RRT, and Fig. 6.15 shows the path generated by the continuum element RRT.

6.6.1 Results and Discussion

Images from video footage of the experiment are shown in Fig. 6.16 and Fig. 6.17. The final position of the robot is shown in Fig. 6.18. As with the previous two sub-scenarios, the mobile base successfully navigated from the ”start” configuration to the ”goal”. During this movement, the continuum element also executed its path from the continuum RRT. We successfully showed the ability of the robot hardware to execute both
RRTs in parallel, which was something our simulations had not been able to show.

These experiments show that our motion planning algorithms for both the mobile base and the continuum element were successfully implemented on the CuRLE robot hardware. As with CuRLE’s predecessor, h+lamp, we saw the robot begin execution of the RRT output. Unlike h+lamp, CuRLE had the appropriate hardware and well-tuned controllers that enabled it to successfully complete the tasks. We serially executed multiple RRTs for the mobile base and continuum element, and we successfully executed the two RRTs in parallel. Any error generated from the PID control was small enough that the robot still retrieved and placed the cup on both shelves respectively.
Figure 6.6: (a-b) Results from Sub-Scenario 1 showing the execution of the RRT from Fig. 6.3 for the mobile base (i.e. CuRLE move from Location 1 to Location 2).
Figure 6.7: (a-b) Results from Sub-Scenario 1 showing the execution of the RRT from Fig. 6.4 (i.e. CuRLE grasp cup).
Figure 6.8: (a) Results from Sub-Scenario 1 showing the execution of the RRT from Fig. 6.5 (i.e. CuRLE lift cup off shelf). (b) Results from Sub-Scenario 2 showing the execution of the RRT from Fig. 6.9.
Figure 6.9: Output of the RRT showing the path required for the mobile base of CuRLE to navigate from the first shelf to the second shelf.

Figure 6.10: Output of the RRT showing the path required for CuRLE to place the cup on the second shelf. (b) is a magnified portion (a) to better show the start and goal configurations.
Figure 6.11: Output of the RRT showing the path required for CuRLE to release the cup on the shelf and move away. (b) is a magnified portion (a) to better show the start and goal configurations.
Figure 6.12: (a-b) Continued results from Sub-Scenario 2 showing the execution of the RRT from Fig. 6.9 for the mobile base (i.e. CuRLE move from Location 2 to Location 3).
Figure 6.13: (a-b) Results from Sub-Scenario 2 showing the execution of the RRT from Fig. 6.10 (i.e. CuRLE release the cup).
Figure 6.14: Output of the RRT showing the path required for the mobile base of CuRLE to navigate from the second shelf to the "docking station".
Figure 6.15: Output of the RRT showing the path required for CuRLE to return to its "home" state ($[u = 0, v = 0, w = 0]$)
Figure 6.16: (a-b) Results from Sub-Scenario 3 showing the execution of the RRT from Fig. 6.14 for the mobile base executing in parallel with the output of the RRT from Fig. 6.15 (i.e. CuRLE return to "home" state while moving from Location 3 to Location 4).
Figure 6.17: (a-b) Continued results from Sub-Scenario 3 demonstrating the parallel execution of the two RRTs.
Figure 6.18: Final results of entire experimentation showing the small amount of error in the final position of the mobile base.
Chapter 7

Conclusions and Suggestions for Future Research

7.1 Conclusions

In Chapter 2, we detailed our first attempt at realizing our continuum robotic mobile lamp as an autonomous system. We recounted the wider history of the home+ effort and described the motivation for the work later conducted and described in Chapters 4 & 5. In evaluating the level of control the user should have when interacting with the home+ suite of robots, a spectrum arises with tele-operation at one end and full autonomy at the other. We described work done to implement tele-operation (i.e. the user is in full control of the robot) via different hardware controllers. To move towards full autonomy in the robot, we implemented the RRT/A* algorithms, detailed in Chapter 5, for the mobile base of the h+lamp. The robot successfully received and tried to execute the path through the in-home environment, but was partially unsuccessful due to hardware limitations and controller shortcomings.

In Chapter 3, we detailed the upgrades that we made to the original home+ robot
hardware (i.e. h+lamp) to correct the issues seen in the experiments in Chapter 2. In addition, we expanded the robot’s hardware and software, renaming it CuRLE in the process, to include the capacity to execute the output from motion planning algorithms, developed in Chapter 5, for the continuum element.

In Chapter 4, we defined the configuration space of a generic single-section extensible continuum element. To develop motion planning algorithms to control our CuRLE robot hardware, we first needed to understand and visualize the configuration space of the robot. While the c-space of a mobile robot is well understood, there has been no formal definition for the configuration space of a continuum element. This thesis presents the first definition of the configuration space in the general case, and then examines how the physical build of our continuum section introduces boundaries in the space. By understanding the unique features of continuum elements and how these features affect the c-space, we created a foundation to apply classical motion planning algorithms for a practical tendon-actuated continuum element. Further, we showed how we can separate the configuration space of the continuum element from that of the mobile base, and thereby characterize the full, complex configuration space of a hybrid continuum, rigid-link, mobile robot in terms of two simple 3D spaces.

In Chapter 5, we used the configuration spaces laid out in Chapter 4 to implement RRT/A* algorithms to plan collision-free paths through the task space of our CuRLE robot. We chose RRTs since their ability to be an anytime-approach fits well with the dynamic nature of CuRLE’s operating environment. Given a start and goal configuration, as well as the location of all obstacles, our motion planning algorithms successfully found a collision-free path for both the mobile base and the continuum element. This is the first time that an RRT approach has been applied to a practical tendon-actuated continuum element. The hybrid nature of our CuRLE robot lends novelty to the motion planning of the mobile base, even though RRTs have widely been used to provide collision-free paths. We validated our
approach by creating software models of our robot and running simulations of practical in-home scenarios, all of which were successful.

In Chapter 6, we implement the RRT/A* algorithm for the first time on a practical robot featuring a tendon-actuated continuum element. Experiments described in this Chapter showed the robot successfully executing a path through the continuum task space. We integrated the controller from Chapter 2 with the software controlling the continuum element to create a cohesive solution that was able to simultaneously navigate both configuration spaces of the CuRLE robot hardware. Experiments reported in this Chapter show CuRLE moving through the in-home scenario to successfully complete a complex series of tasks, guided by the RRT/A* algorithms. The robot demonstrated its potential to provide in-home care and assist with aging-in-place.

7.2 Future Work

7.2.1 Migrate CuRLE to a Fully Independent System

While the CuRLE hardware was able to autonomously follow the path generated by the motion planning software, the central computer was still required to run the RRT/A* and communicate the path to the robot. Since CuRLE already has the Raspberry Pi onboard to handle its wireless communication and interface with the Arduino Due, we recommend that the motion planning code be ported over to the Pi. The code already runs in C++ and uses the Boost Graph library, which is platform independent. If the Raspberry Pi handles all of the path planning, then the robot will no longer have to rely on another computer to handle the motion planning. This will help streamline the development process and take the next step towards CuRLE being an autonomous, stand-alone unit. In addition, we envision several added features that would allow users and developers to interact more
directly with Raspberry Pi without relying on a SSH or Virtual Desktop connection. Having an LCD display and user input devices would facilitate future development and provide a method for future users to engage with the robot. In addition, having an audio system would allow CuRLE to provide added feedback to the user beyond the visual feedback of the LEDs described in Chapter 6. These relatively simple upgrades would allow future developers easier avenues for new system features and would provide users with a richer, more fulfilling experience.

### 7.2.2 Further Simulation Software Development

In addition to the upgrades to make CuRLE a stand-alone unit, we recommend further development of the software model and simulation environment of CuRLE. Combining the interactive GUI described in Chapter 5 for the continuum arm with the simulation code for the mobile base also detailed in that Chapter would allow future researchers to perform more comprehensive studies and experiments with the motion planning algorithms. In this thesis, we described the justification we used to separate the configurations spaces of the continuum element and the mobile base, and we were able to experiment in simulation with each individually, but we were unable to do so with both spaces in parallel. Having the capacities to run such studies would be useful in validating the motion planning output before implementing it on the hardware. In addition, while we studied different configurations of the RRT, more research can be done in this area, particularly in regards to optimization of the RRT using the RRT* algorithm. Rather than simply find a path to the goal, the motion planning can be used to find the optimal path. Since the robot will be interacting with humans to accomplish tasks, it will be important for it to execute relatively quickly. Since the CuRLE hardware and controllers may continue to present limitations, the more comprehensive simulation software that we are proposing will allow these new experiments to
be executed and tested before transporting them to the hardware.

### 7.2.3 Remove Restrictions on Degrees of Freedom

In this work, we restricted several of the DoF of the robot, and in future development, we intend these restrictions to be lifted. One of the advantages of the limited DoF was that we were able to visualize the RRT output in the configuration space of the robot. While removing the restrictions will necessarily make visualization impossible, advancements in the simulation software, as we have suggested, will allow us to test the full, unlocked capacity of the robot. CuRLE will be able to function with all of its DoF and we can use the simulations to verify the motion planning before executing it on the hardware.

### 7.2.4 Hardware Upgrades

Many of the existing hardware limitations would be eliminated by making upgrades to the CuRLE robot. First, the stiffness of the springs in the tension sensors should be increased. Given the magnitudes of the forces required in the tendons to realize the robot kinematics, the linear spring-loaded potentiometers saturated their tension readings without being able to accurately give a measurement of the tension in the tendon. While this allowed the controller to keep "slack" out of the tendon by keeping the tension above a threshold close to the saturation value, it prevented the controller from keeping the tension balanced between opposing tendons. As such, the tension feedback controller was only marginally useful, and largely left unused during experimentation.

In addition to upgrading the tension sensors, the spool mechanism for winding the tendons around the motor shaft should be upgraded. As we previously mentioned, the spools are integral for accurately tracking the length of the tendons, which is crucial for the kinematics and control of the robot. The current hardware design does not perfectly
align the spool with the tendon as it travels up the backbone, meaning that there is a slight probability of the tendon slipping off of the spool. This occurred multiple times during testing, and were it not for the tension sensors, would have ruined controller’s ability to estimate the tendon length. In addition to routing the tendons more accurately, a new, more reliable, method should be developed to track the tendon length. This issue could be potentially solved in software by creating a tracking model to account for dynamic noise (e.g. slipping) in the tendon winding, or it could solved with a more sophisticated hardware design. We envision a collection of passive spools that guarantee tendon alignment, with an optical encoder attached to one of the spools to measure the change in tendon length without having the current issue of a varying spool radius.

The final hardware upgrade that we propose for future research is to develop a new material to replace the PEX backbone. Over time, the material properties of the current backbone cause it to deform as it bends, especially when CuRLE bends to its extremes. In fact, even the constant strain of gravity causes the material to deform, which eliminates constant curvature. As the backbone strays from constant curvature, a foundational assumption for the kinematic model, the control of the robot degrades. Since we assume constant curvature, the actions needed to reach a configuration will evolve the robot into a shape that we do not model, which causes errors in the navigation. We recommend development or selection of a synthetic material (e.g. polyurethane, silicon rubber, etc.) to create a new backbone for CuRLE. The material must be elastic enough to return to its original shape after bending but also rigid enough to support its own weight (with the help of the tendons) without collapsing under gravity.
7.2.5 Integrating with a Sensing Network

Since CuRLE is the realization of a key aspect of the "fully autonomous" end of the user control spectrum, integrating the robot with sensing technology will be vital for future development. In this thesis, the "sensing problem" was solved by assuming knowledge of the locations of the goal configurations and the exact layout of the environments \(a\ priori\). For the robotic lamp to claim full autonomy, it must be able to process user commands (e.g. voice recognition system) and sense the environment (e.g. vision system) to avoid obstacles and locate its goal. To accomplish this, we envision at least two vision systems: a "global" system that acts as a Monitor of the Operating Environment (MOE) that is mounted as an "eye-in-sky" and detects the environment and communicates with CuRLE. The second vision system is local, mounted at the end-effector of CuRLE. This would be primarily used with locating objects to pick up/place. In addition to these vision systems, we could imagine other sensors in the home as part of the "smart home" societal trend. Whether it be other members of \(home^+\), like the Linearly Actuated Robotic End-table Element (LAREE), or other devices that been integrated with "smart" technology, we hope that CuRLE will be fully integrated in the home to provide care and assist with aging-in-place.
Appendices
Appendix A  CuRLE Robot Software: Arduino

```c
#include <SPI.h>
#include <pwm_lib.h>
#include "_aux.h"

// Continuum State Variables
const float s = 1.03;
const float d = 0.022;

int32_t eCount[5];
int32_t prevCnt[] = {0, 0, 0, 0, 0};
const int32_t eCountZero[] = {77514, 72058, 70443, 73889, 0};
float spooledLen[4];
float K_tendon[] = {0.15, 0.001, 0.02, 0.15, 0.001, 0.02};

float u = 0;
float v = 0;
float w = 0;

float u_set = 0;
float v_set = 0;
float w_set = 0;

bool debugPrint = false;
bool ESTOP = true;
bool initialized = false;
bool idle = true;

// Mobile Base
float mu[3];
```
int muPtr = MU_IDLE;
float stpt[] = { 0, 0 };
float K_mobile[] = { 0.15, 0.001, 0.023, 0.025};

// Motors
MotorList motors;

// Tension Sensors
TensionSensor tSensor[4];

// Clock for Encoder Chips
// defining pwm object using pin 53, pin PB14 mapped to pin 53 on the DUE
// this object uses PWM channel 2
using namespace arduino_due::pwm_lib;
pwm<pwm_pin::PWMH2_PB14> encoderCLK;

// Gripper
// servo 44-> PWMH5 (PC19)
servo<pwm_pin::PWMH5_PC19> servo_wrist; // blue strip
// 25 -> u = -v, 65 -> v = 0, 95 -> u = v, 135 -> u = 0
int wristAngle = 65;
// servo 45-> PWMH6 (PC18)
servo<pwm_pin::PWMH6_PC18> servo_claw;
// 0 -> open, 90 -> closed (clear Solo)
int clawAngle = 0;

// Comm
String msg = "";
String cmd = "";
// Time
unsigned long timeNow = 0;
unsigned long timeLast = 0;
unsigned long timeLastSave = 0;
unsigned long timeLastPrint = 0;
unsigned long timeWait = 0;
int timeStep = 50;
int timeSaveInterval = 60000;
int timePrintInterval = 1000;

void setup() {
  digitalWrite(RESET_PIN, HIGH);
  pinMode(RESET_PIN, OUTPUT);
  Serial.begin(9600);
  while (!Serial);
  Serial1.begin(9600);
  while (!Serial1);
  Serial.println("Lamp Test");
  encoderCLK.start(ENC_CLK_PERIOD, ENC_CLK_DUTY);
  servo_wrist.start(
    SERVO_PWM_PERIOD,    // pwm servo period
    75000,    // 1e-8 s. (1 msecs.), minimum duty value
    225000,   // 1e-8 s. (2 msecs.), maximum duty value
    0,       // minimum angle, corresponding minimum servo angle
    140,     // maximum angle, corresponding minimum servo angle
    wristAngle // initial servo angle
  );
servo_claw.start(
    SERVO_PWM_PERIOD, // pwm servo period
    75000, // 1e-8 s. (1 msecs.), minimum duty value
    225000, // 1e-8 s. (2 msecs.), maximum duty value
    0, // minimum angle, corresponding minimum servo angle
    140, // maximum angle, corresponding minimum servo angle
    clawAngle // initial servo angle
);

// Switches
pinMode(SW_LED, OUTPUT);
  _aux::setswitch(SW_LED, OFF);

pinMode(SW_DRV, OUTPUT);
  _aux::setswitch(SW_DRV, OFF);

// Have to start SPI before initializing motors (for the encoders)
SPI.begin();
motors.init();

// Initialize the tension sensors (analog inputs)
analogReadResolution(12);
tSensor[0].init(T_SENSOR1);
tSensor[1].init(T_SENSOR2);
tSensor[2].init(T_SENSOR3);
tSensor[3].init(T_SENSOR4);

// status variables
ESTOP = true;
initialized = false;
// request the state data from the Raspberry Pi
sendRasPiMsg(RPI_LOAD);

void loop() {
    float len[4], deltaLen[4];
    float set[4]; // drive motors use stpt[], which is global
    static int32_t errLast[] = {0, 0, 0, 0, 0, 0};

    int tensionErr;
    float tensionK;

    static RunningSum<int32_t> errSum[6];
    static bool once = true;
    int32_t err[6];
    int32_t cnt[6], deltaC[6];
    int CPR, i, k;
    float pwm[6];
    const float TENDON_PWM_SAT = 100.0;
    const float DRIVE_PWM_SAT = 75;
    const int32_t TENDON_ERR_THRESH = 40;
    const int32_t DRIVE_ERR_THRESH = 60;
    const int TENSION_THRESH = 4000;
    bool complete[] = {false, false, false, false, false, false};
    static int completeCnt = 0;

    if (once) {
        for (int i = T_MOTOR1; i <= D_MOTOR2; ++i) {
            errSum[i].init(10);
        }
    }
once = false;

}

timeNow = millis();

if (!initialized) {
    if (timeNow - timeLast > 1000) {
        // aux::toggle(SW_LED);
        timeLast = timeNow;
    }
}
else {

   // should occur every 50ms
   if ( (timeNow-timeLast) >= timeStep ) {
        // Kinematics
        // get lengths from the current state
        len[T_MOTOR1] = s - (v*d);
        len[T_MOTOR3] = s + (v*d);
        len[T_MOTOR2] = s + (u*d);
        len[T_MOTOR4] = s - (u*d);

        // update current state[u, v, w]
        // get the current counts
        for (i = T_MOTOR1; i <= T_MOTOR4; ++i) {
            cnt[i] = eCount[i] + motors[i].getCount();
            deltaC[i] = cnt[i] - prevCnt[i];
            // this is really the delta length
deltaLen[i] = _aux::lengthOnSpool(motors[i].CPR(), prevCnt[i]) - _aux::lengthOnSpool(motors[i].CPR(), cnt[i]);

len[i] += deltaLen[i];
}

// get counts from the drive motors
for (i = D_MOTOR1; i <= D_MOTOR2; ++i) {
cnt[i] = motors[i].getCount();
}

// Set points for tendon motors
set[T_MOTOR1] = s - (v_set*d);
set[T_MOTOR3] = s + (v_set*d);
set[T_MOTOR2] = s + (u_set*d);
set[T_MOTOR4] = s - (u_set*d);

// calculate errors
for (i = T_MOTOR1; i <= T_MOTOR4; ++i) {
    CPR = motors[i].CPR();

    // calculate tension error (not used...)
    tensionErr = tSensor[i].read() - TENSION_THRESH;
    tensionErr = (tensionErr > 0) ? 0 : tensionErr;

    err[i] = _aux::deltaLengthToSetCount(CPR, cnt[i], (set[i] - len[i])) - cnt[i];
}

k = 0;
pwm[i] = (err[i] * K_tendon[0+k]) + (err[i] − errLast[i])/
static_cast<float>((timeNow−timeLast)/1000.0)∗K_tendon[1+k] + errSum[i].add(err[i])∗K_tendon[2+k];

pwm[i] = (pwm[i] < 0) ? max(pwm[i], −TENDON_PWM_SAT) : min(pwm[i], TENDON_PWM_SAT);
}

deltaC[D_MOTOR1] = (cnt[D_MOTOR1]−prevCnt[D_MOTOR1])−(cnt[D_MOTOR2]−prevCnt[D_MOTOR2]);
deltaC[D_MOTOR2] = (cnt[D_MOTOR2]−prevCnt[D_MOTOR2])−(cnt[D_MOTOR1]−prevCnt[D_MOTOR1]);

for (i = D_MOTOR1; i <= D_MOTOR2; ++i) {
    err[i] = stpt[i−D_MOTOR1] − cnt[i];
    // PID control and velocity control (i.e. try to keep them at the same speed)
    pwm[i] = (err[i] * K_mobile[0]) + (err[i] − errLast[i])/(static_cast<float>((timeNow−timeLast)/1000.0)∗K_mobile[1] + errSum[i].add(err[i]))∗K_mobile[2] − deltaC[i]*K_mobile[3];
    pwm[i] = (pwm[i] < 0) ? max(pwm[i], −DRIVE_PWM_SAT) : min(pwm[i], DRIVE_PWM_SAT);
}

if (timeNow − timeLastPrint > timePrintInterval) {
    if (debugPrint) {
        // Serial.println("spl: ["+ String(spooledList[0],4)+","+ String(spooledList[1],4)+","+ String(spooledList[2],4)+","+ String(spooledList[3],4)+"]);
        // Serial.println("dCt: ["+ String(deltaC[0])+","+ String(deltaC[1])+","+ String(deltaC[2])+","+ String(deltaC[3])+"]);
        Serial.println("dCt: ["+ String(deltaC[4])+","+ String(deltaC[5])+"]);
// Serial.println("dLn: [" + String(deltaLen[0], 4) + "," + String(deltaLen[1], 4) + "," + String(deltaLen[2], 4) + "," + String(deltaLen[3], 4) + "]");

// Serial.println("len: [" + String(len[0], 4) + "," + String(len[1], 4) + "," + String(len[2], 4) + "," + String(len[3], 4) + "]");

// Serial.println("set: [" + String(set[0], 4) + "," + String(set[1], 4) + "," + String(set[2], 4) + "," + String(set[3], 4) + "]");

Serial.println("stp: [" + String(stpt[0]) + "]");

Serial.println("err: [" + String(err[0]) + "," + String(err[1]) + "," + String(err[2]) + "," + String(err[3]) + "," + String(err[4]) + "," + String(err[5]) + "]");

// Serial.println("errSum: [" + String(errSum[0].sum()) + "," + String(errSum[1].sum()) + "," + String(errSum[2].sum()) + "," + String(errSum[3].sum()) + "]");

Serial.println("pwm: [" + String(pwm[0], 4) + "," + String(pwm[1], 4) + "," + String(pwm[2], 4) + "," + String(pwm[3], 4) + "," + String(pwm[4], 4) + "," + String(pwm[5], 4) + "]");

// Serial.println("mCt: [" + String(motors[0].getCount()) + "," + String(motors[1].getCount()) + "," + String(motors[2].getCount()) + "," + String(motors[3].getCount()) + "]");

Serial.println("mCt: [" + String(motors[4].getCount()) + "," + String(motors[5].getCount()) + "]");

Serial.println("mu: [" + String(mu[0]) + "," + String(mu[1]) + "," + String(mu[2], 4) + "]");

muPtr = " + String(muPtr) ;

}
timeLastPrint = timeNow ;
}
// update
for (i = T_MOTOR1; i <= D_MOTOR2; ++i) {
    if (i <= T_MOTOR4) {
        complete[i] = (abs(err[i]) < TENDON_ERR_THRESH);
    } else {
        complete[i] = (abs(err[i]) < DRIVE_ERR_THRESH);
    }
    if (!ESTOP) {
        motors[i].run(static_cast<int>(pwm[i]));
    }
    prevCnt[i] = cnt[i];
    errLast[i] = err[i];
}
if (_aux::checkComplete(complete, 6) && !idle) {
    completeCnt++;
    if (completeCnt >= 5) {
        motors.stopAll();
        if (advanceMu()) {
            sendRasPiMsg(RPI.Complete);
            _aux::blink();
            idle = true;
        }
        completeCnt = 0;
    }
} else {
    completeCnt = 0;
}
u = (len[T_MOTOR2] - len[T_MOTOR4]) / (2*d);
v = (len[T_MOTOR3] - len[T_MOTOR1]) / (2*d);
timeLast = timeNow;

if (timeWait > 0 && timeNow >= timeWait) {
    timeWait = 0;
    sendRasPiMsg(RPI_READY);
}

// should occur every minute (60,000 ms)
if (timeNow−timeLastSave >= timeSaveInterval) {
    // save
    sendRasPiMsg(RPI_SAVE);
    timeLastSave = timeNow;
}

// return true if done with mu (action vector)
bool advanceMu() {
    int delta;
    muPtr = min(muPtr+1, MU_IDLE);
    if (muPtr == MU_IDLE) {
        return true;
    }
    switch(muPtr) {
        case MU_ROT1:
            case MU_ROT2:
                delta = round((mu[muPtr]) * ROBOT_RADIUS / (WHEEL_RADIUS*2*PI) *
                static_cast<float>(motors[D_MOTOR1].CPR()));
                stpt[0] = motors[D_MOTOR1].getCount() + delta;
stp[1] = motors[D_MOTOR2].getCount() + delta;
break;
case MU_TRAN:
delta = round((mu[muPtr]) / (WHEEL_RADIUS*2*PI) * static_cast<float>(motors[D_MOTOR1].CPR()));
stp[0] = motors[D_MOTOR1].getCount() + delta;
stp[1] = motors[D_MOTOR2].getCount() - delta;
break;
default:
stp[0] = motors[D_MOTOR1].getCount();
stp[1] = motors[D_MOTOR2].getCount();
break;
return false;
}

// COMM
void serialEvent() {
byte buf[512];
char c;
while (Serial.available()) {
c = Serial.read();
msg += c;
if (c == ';') {
msg.getBytes(buf, 512);
Serial1.write(buf, msg.length());
msg = "";
}
else {
    // for debugging: user can select a motor to interface with
// mid = _aux::process_char(motors, c) - 1;

Serial.println(_aux::countToString(motors, -1));

if (c == '1') {
    motors[0].run(25);
}
else if (c == '2') {
    motors[1].run(25);
}
else if (c == '3') {
    motors[2].run(25);
}
else if (c == '4') {
    motors[3].run(25);
}
else if (c == '5') {
    motors[0].run(-25);
}
else if (c == '6') {
    motors[1].run(-25);
}
else if (c == '7') {
    motors[2].run(-25);
}
else if (c == '8') {
    motors[3].run(-25);
}
else if (c == '9') {
    motors[4].run(100);
    motors[5].run(-100);
}
else if (c == '0') {

motors[4].run(-100);
    motors[5].run(100);
}
else if (c == 'f') {
    motors[6].run(255);
}
else if (c == 'r') {
    motors[6].run(-255);
}
else if (c == 's') {
    motors.stopall();
}
else if (c == 'o') {
    _aux::setswitch(SW_LED, ON);
}
else if (c == 'p') {
    _aux::setswitch(SW_LED, OFF);
}

else if (c == 't') {
    Serial.println("TS[] = ["+String(tSensor[0].read())+" ","+String(tSensor[1].read())+" ","+String(tSensor[2].read())+" ","+String(tSensor[3].read())+" ]");
}
else if (c == 'q') {
    servo_claw.set_angle(90);
}
else if (c == 'a') {
    servo_claw.set_angle(45);
}
else if (c == 'z') {

servo_claw.set_angle(0);
}

else if (c == 'y') {
    wristAngle += 5;
    Serial.println("wristAngle = " + String(wristAngle));
}

else if (c == 'h') {
    servo_wrist.set_angle(wristAngle);
    Serial.println("wristAngle = " + String(wristAngle));
}

else if (c == 'n') {
    wristAngle -= 5;
    Serial.println("wristAngle = " + String(wristAngle));
}

else if (c == 'u') {
    clawAngle += 5;
    Serial.println("clawAngle = " + String(clawAngle));
}

else if (c == 'j') {
    servo_claw.set_angle(clawAngle);
    Serial.println("clawAngle = " + String(clawAngle));
}

else if (c == 'm') {
    clawAngle -= 5;
    Serial.println("clawAngle = " + String(clawAngle));
}

}

void serialEvent1 () {

char c;
static int wordCount = 0;
while(Serial1.available()) {
    c = Serial1.read();
    cmd += c;
    if (c == '/') {
        ++wordCount;
    }
    if (c == ';') {
        Serial.println("recv'd (raspi) -> " + cmd);  
        processRasPiMsg(cmd, ++wordCount);
        cmd = "";
        wordCount = 0;
    }
}

void processRasPiMsg(String buf, int wordCount)
{
    String msgWords[10];
    String op = "";
    String param = "";
    String tmp = "";
    float arg[6];
    int enc[5];

    int i, j, k, idx, c;

    // split msg into words
    j = 0; idx = 0;
    for (i = 0; i < buf.length(); ++i) {

if (buf[i] == '/ ' || buf[i] == ' ;') {
    msgWords[idx++] = buf.substring(j, i+1);
    j = i + 1;
}
}

for (c = 0; c < wordCount; ++c)
{
    for (i = 0; i < msgWords[c].length(); ++i) {
        if (msgWords[c][i] == ':' )
            break;
    }
    op = msgWords[c].substring(0, i);
    param = msgWords[c].substring(i+1);

    if (op.compareTo("init") == 0) {
        if (initialized) {
            continue;
        }
        _aux::blink();
        ESTOP = false;
        initialized = true;
    } else if (op.compareTo("state") == 0) {
        j = 0; idx = 0;
        for (i = 0; i < param.length(); ++i) {
            if (param[i] == ',' || param[i] == ';' || param[i] == '/') {
                tmp = param.substring(j, i);
                arg[idx++] = tmp.toFloat();
                j = i + 1;
            }
        }
    }
}
u = arg[0];
\n\nv = arg[1];
\n\nw = arg[2];
\n}
\nelse if (op.compareTo("encoder") == 0) {
  j = 0; idx = 0;
  for (i = 0; i < param.length(); ++i) {
    if (param[i] == ',' || param[i] == ':' || param[i] == '/') {
      tmp = param.substring(j, i);
      enc[idx++] = tmp.toInt();
      j = i + 1;
    }
  }

  for (int _i = T_MOTOR1; _i <= T_MOTOR4; ++_i) {
    eCount[_i] = enc[_i];
    spooledLen[_i] = _aux::lengthOnSpool(motors[_i].CPR(), eCount[_i]);
    prevCnt[_i] = eCount[_i];
  }
  eCount[4] = enc[4];
}

else if (op.compareTo("goto") == 0) {
  j = 0; idx = 0;
  for (i = 0; i < param.length(); ++i) {
    if (param[i] == ',' || param[i] == ':' || param[i] == '/') {
      tmp = param.substring(j, i);
      arg[idx++] = tmp.toFloat();
      j = i + 1;
    }
  }
  u_set = arg[0];
v_set = arg[1];
w_set = arg[2];
idle = false;
}
else if (op.compareTo("drive") == 0) {
    j = 0; idx = 0;
    for (i = 0; i < param.length(); ++i) {
        if (param[i] == ',' || param[i] == ':' || param[i] == '/') {
            tmp = param.substring(j,i);
            arg[idx++] = tmp.toFloat();
            j = i+1;
        }
    }
    stpt[0] = motors[D_MOTOR1].getCount();
stpt[1] = motors[D_MOTOR2].getCount();
mu[MU_ROT1] = arg[0];
mu[MU_TRAN] = -arg[1];
mu[MU_ROT2] = arg[2];
muPtr = MU_PREP;
idle = false;
}
else if (op.compareTo("drive_gain") == 0) {
    j = 0; idx = 0;
    for (i = 0; i < param.length(); ++i) {
        if (param[i] == ',' || param[i] == ':' || param[i] == '/') {
            tmp = param.substring(j,i);
            arg[idx++] = tmp.toFloat();
            j = i+1;
        }
    }
    K_mobile[0] = arg[0];
K_mobile[1] = arg[1];
K_mobile[3] = arg[3];
}
else if (op.compareTo("tendon_gain") == 0) {
    j = 0; idx = 0;
    for (i = 0; i < param.length(); ++i) {
        if (param[i] == ',' || param[i] == ':' || param[i] == '/') {
            tmp = param.substring(j, i);
            arg[idx++] = tmp.toFloat();
            j = i + 1;
        }
    }
    K_tendon[0] = arg[0];
    K_tendon[1] = arg[1];
    K_tendon[3] = arg[0];
    K_tendon[4] = arg[1];
}
else if (op.compareTo("estop") == 0) {
    motors.stopall();
    ESTOP = true;
}
else if (op.compareTo("resume") == 0) {
    ESTOP = false;
}
else if (op.compareTo("status") == 0) {
    sendRasPiMsg(RPI_STATUS);
}
else if (op.compareTo("grab") == 0) {

servo_claw.set_angle(90);

}  
else if (op.compareTo("letgo") == 0) {
    servo_claw.set_angle(0);
}
else if (op.compareTo("print_setpoint") == 0) {
    sendRasPiMsg(RPI_SETPOINT);
}
else if (op.compareTo("print_state") == 0) {
    sendRasPiMsg(RPI_STATE);
}
else if (op.compareTo("print_encoders") == 0) {
    sendRasPiMsg(RPI_ECOUNT);
}
else if (op.compareTo("print_gain") == 0) {
    sendRasPiMsg(RPI_GAIN);
}
else if (op.compareTo("poll") == 0) {
    sendRasPiMsg(RPI_LOAD);
}
else if (op.compareTo("passive") == 0) {
    _aux::setSwitch(SW_DRV, ON);
}
else if (op.compareTo("active") == 0) {
    _aux::setSwitch(SW_DRV, OFF);
}
else if (op.compareTo("wait") == 0) {
    timeWait = timeNow + param.toInt();
}
else if (op.compareTo("led_on") == 0) {
    _aux::setSwitch(SW_LED, ON);
else if (op.compareTo("led_off") == 0) {
    _aux::setswitch(SW_LED, OFF);
}

else if (op.compareTo("reset") == 0) {
    digitalWrite(RESET_PIN, LOW);
}
else if (op.compareTo("debug") == 0) {
    /* debug : 0 ; OR debug : 1 ; */
    if (param.toInt()) {
        debugPrint = true;
    } else {
        debugPrint = false;
    }
}
else {
    Serial.println("Invalid cmd..." + msgWords[c]);
}

void sendRasPiMsg(int op)
{
    byte buf[512];
    String message = "";
    switch(op) {
    case RPI_LOAD: // load
        message = (!initialized) ? "load:=" : "noop:=";
        break;
    case RPI_SAVE: // save the current state and encoder values

message = "save:\n";
message += " state:\n" + String(u,4) + "," + String(v,4) + "," + String(w,4) + ";
message += "encoder:\n" + String(eCount[0] + motors[0].getCount()) + ",";
message += String(eCount[1] + motors[1].getCount()) + ",";
message += String(eCount[3] + motors[3].getCount()) + ",";
message += String(eCount[4] + motors[4].getCount()) + ";
break;
case RPI_SETPOINT: // send the current setpoint for all
message = "print:\n";
message += "setpoint:\n" + String(u_set,4) + "," + String(v_set,4) + "," + String(w_set,4) + ";
break;
case RPI_STATE: // send current state
message = "print:\n";
message += " state:\n" + String(u,4) + "," + String(v,4) + "," + String(w,4) + ";
break;
case RPI_ECOUNT: // send encoder counts
message = "print:\n";
message += "encoder:\n" + String(eCount[0] + motors[0].getCount()) + ",";
message += String(eCount[1] + motors[1].getCount()) + ",";
message += String(eCount[3] + motors[3].getCount()) + ",";
message += String(eCount[4] + motors[4].getCount()) + ";
break;
case RPI_GAIN: // send tendon gains
message = "print:\n";
message += "tendon_gain:\n" + String(K_tendon[0],4) + ",";
message += String(K_tendon[1],4) + ",";
message += String(K_tendon[2],4)+"",";
message += String(K_tendon[3],4)+"",";
message += String(K_tendon[4],4)+"",";
message += String(K_tendon[5],4)+"/";
message = "print:/";
message += "driveGain:"+String(K_mobile[0],4)+"",";
message += String(K_mobile[1],4)+"",";
message += String(K_mobile[2],4)+"",";
message += String(K_mobile[3],4)+";";
break;

case RPI_COMPLETE:
  message = "complete:";
  break;

case RPI_STATUS:
  message = "status:";
  message += (idle) ? "idle:" : "running:";
  break;

case RPI_READY:
  message = "wait over:__ready__;";
  break;

default:
  message = "noop:";
  break;
}
message.getBytes(buf,512);
Serial1.write(buf, message.length());
}

Listing 1: Main CuRLE Arduino Code
/*
Auxillary prototypes for Lamp

AUTHOR: Zach Hawks
DATE: March 13, 2019

This header contains all of the auxillary functions
for the lamp in the namespace _aux

License: CCAv3.0 Attribution−ShareAlike (http://creativecommons.org/licenses/by−sa/3.0/)
You’re free to use this code for any venture. Attribution is greatly appreciated.

//============================================================================================
*/

 ifndef __LAMP_AUXILLARY__
define __LAMP_AUXILLARY__

#include "motor_list.h"
#include "motor.h"
#include "linear_actuator.h"
#include "tension_sensor.h"
#include "lamp_def.h"
#include "running_sum.h"
#include "running_sum.cpp"

namespace _aux {

   void setswitch(int pin, int val);
   void toggle(int pin);
}
void blink();
bool checkComplete(bool* complete, int len);
String countToString(MotorList& motors, int idx);
int process_char(MotorList& motors, char c);
float getRadius(int CPR, int32_t eCount);
float lengthOnSpool(int CPR, int32_t eCount);
int32_t deltaLengthToSetCount(int CPR, int32_t eCount, float deltaLen)
;
float tLengthAbs(MotorList& motors, int idx, int32_t eCount, int32_t
eCountZero, float s);
float tLength(int idx, float u, float v, float s, float d);
};

Listing 2: Auxillary header

#include "_aux.h"

float _aux::getRadius(int CPR, int32_t eCount)
{
    int rev = eCount/CPR;
    return SPOOL_RADIUS_BASE + (static_cast<float>(rev)/1000.0);
}

int32_t _aux::deltaLengthToSetCount(int CPR, int32_t eCount, float
deltaLen)
{
    // GOAL: to find the setCount value given a deltaLen and the current
eCount
    float setLen, boundaryLen, radius, tmpLen;
    int32_t cnt, deltaCnt;
// get the desired length
setLen = lengthOnSpool(CPR, eCount) - deltaLen;

// cnt is the iteration control variable
cnt = eCount;
if (deltaLen > 0) {
    // find the lower boundary of the current window
    boundaryLen = lengthOnSpool(CPR, (cnt/CPR)*CPR);
    // check if in the same "window" (radius is constant per window)
    if (boundaryLen > setLen) {
        // not in same window --> iterate by moving count into the next "
        window" (lower)
        cnt = ((cnt/CPR)*CPR) - 1;
    }
    else {
        // in same window --> so can determine exactly where setCount is
        radius = getRadius(CPR, cnt);
        tmpLen = lengthOnSpool(CPR, cnt);
        deltaCnt = (tmpLen - setLen) / (2*PI*radius) * CPR;
        return (cnt - deltaCnt);
    }
} else {
    while (1) {
        // find the upper boundary of the current window
        boundaryLen = lengthOnSpool(CPR, ((cnt/CPR)+1)*CPR);
        // check if in the same "window" (radius is constant per window)
        if (boundaryLen < setLen) {
            // not in same window --> iterate by moving count into next "
            // ...
window” (upper)

cnt = ( (cnt/CPR)+1 ) * CPR;

}  
else {
  // in same window -> so can determine exactly where setCount is
  radius = getRadius(CPR, cnt);
  tmpLen = lengthOnSpool(CPR, cnt);
  deltaCnt = (setLen - tmpLen) / (2*PI*radius) * CPR;
  return (cnt + deltaCnt);
}
}

return 0;

}

float _aux::lengthOnSpool(int CPR, int32_t eCount)
{
  float len = 0;
  int cnt = 0;
  float rad = SPOOL_RADIUS_BASE;
  while (cnt < eCount) {
    if (cnt + CPR < eCount) {
      len += 2*rad*PI;
      cnt += CPR;
      rad += 0.001;
    }
    else {
      len += static_cast<float>(eCount - cnt) / static_cast<float>(CPR)  
             * (2*rad*PI);
      return len;
    }
  }
float _aux::tLengthAbs (MotorList& motors, int idx, int32_t eCount,
int32_t eCountZero, float s)
{
    float lenZero = lengthOnSpool(motors[idx].CPR(), eCountZero);
    float len = lengthOnSpool(motors[idx].CPR(), eCount);
    return s + (lenZero - len);
}

float _aux::tLength(int idx, float u, float v, float s, float d)
{
    switch (idx) {
    case T_MOTOR1:
        return s - (v*d);
    case T_MOTOR2:
        return s + (u*d);
    case T_MOTOR3:
        return s + (v*d);
    case T_MOTOR4:
        return s - (u*d);
    default:
        return -1;
    }
}

bool _aux::checkComplete(bool* complete, int len)
{
    for (int i = 0; i < len; ++i) {
        if (!complete[i]) {
        }
return false;
}
}
return true;
}

void _aux::toggle(int pin)
{
    int val = digitalRead(pin);
    if (val) {
        setSwitch(pin, OFF);
    }
    else {
        setSwitch(pin, ON);
    }
}

void _aux::blink()
{
    setSwitch(SW_LED, ON);
    delay(100);
    for (int i = 0; i < 5; ++i) {
        toggle(SW_LED);
        delay(100);
    }
    setSwitch(SW_LED, OFF);
}

void _aux::setSwitch(int pin, int val)
{
    if (val == ON) {

digitalWrite(pin, HIGH);
}
else {
    digitalWrite(pin, LOW);
}

int _aux::process_char(MotorList& motors, char c) {
    int i;
    if (c >= '1' && c <= '7') {
        i = c - '0';
        return i;
    } else {
        return 0;
    }
}

String _aux::countToString(MotorList& motors, int idx) {
    String str;
    if (idx < 0) /* all */
        str = "count[    ] = [ ";
    for (int i = 0; i < M_CNT-1; ++i) {
        str = str + String(motors[i].getCount()) + ", ";
    }
    str = str + String(motors[M_CNT-1].getCount()) + "]
    
else {
    str = "count[ " + String(idx+1) + "] = [ " + String(motors[idx].

136
Listing 3: Auxillary functions

```c
getCount() + " ]");
}
return str;
}
```

// =========================HEADER
=============================================================
/*
LS7366 Quadrature Counter Interface Library
AUTHOR: Zach Hawks
DATE: March 9, 2019

This is a simple library program to read encoder counts collected by the LS7366 chip (no breakout board). This code is derived from code developed by Jason Traud. (https://github.com/SuperDroidRobots/Encoder-Buffer-Breakout)

License: CCAv3.0 Attribution−ShareAlike (http://creativecommons.org/licenses/by-sa/3.0/)
You’re free to use this code for any venture. Attribution is greatly appreciated.

//==============================================HEADER*/

#if undef __ENCODER_CHIP_H__
#define __ENCODER_CHIP_H__
#include <inttypes.h>
```
#include <Wire.h>
#include "Arduino.h"

class EncoderChip {
    public:
        EncoderChip(int ssPin);
        EncoderChip(const EncoderChip& ec);
        EncoderChip();
        EncoderChip& operator=(const EncoderChip&) = delete;

        void init();
        int32_t read();
        void clear();

        // debug method
        int getpin() const { return _ssPin; }

    private:
        static int ID;
        int _id;
        int _ssPin;
};

Listing 4: Encoder Chip Class Header
EncoderChip::EncoderChip(int ssPin) : _id(ID++), _ssPin(ssPin)
{
}

EncoderChip::EncoderChip(const EncoderChip& ec) : _id(ec._id), _ssPin(ec._ssPin)
{
}

EncoderChip::EncoderChip() : _id(-1), _ssPin(-1)
{
}

void EncoderChip::init()
{
    if (_id < 0) {
        return;
    }
    // Set slave selects as outputs
    pinMode(_ssPin, OUTPUT);

    // Raise select pins
    // Communication begins when you drop the individual select signals
    digitalWrite(_ssPin, HIGH);

    // Initialize encoder
    // Clock division factor: 0
    // Negative index input
    // free-running count mode
    // x4 quadrature count mode (four counts per quadrature cycle)
// NOTE: For more information on commands, see datasheet

digitalWrite(_ssPin, LOW); // Begin SPI conversation
SPI.transfer(0x88); // Write to MDR0
SPI.transfer(0x03); // Configure to 4 byte mode
digitalWrite(_ssPin, HIGH); // Terminate SPI conversation

int32_t EncoderChip::read()
{
    if (_id < 0) {
        return 0;
    }

    // Initialize temporary variables for SPI read
    int32_t count_1, count_2, count_3, count_4;
    int32_t cnt;

    digitalWrite(_ssPin, LOW); // Begin SPI conversation
    SPI.transfer(0x60); // Request count
    count_1 = SPI.transfer(0x00); // Read highest order byte
    count_2 = SPI.transfer(0x00);
    count_3 = SPI.transfer(0x00);
    count_4 = SPI.transfer(0x00); // Read lowest order byte
digitalWrite(_ssPin, HIGH); // Terminate SPI conversation

    // Calculate encoder count
    cnt = (count_1 << 8) + count_2;
    cnt = (cnt << 8) + count_3;
    cnt = (cnt << 8) + count_4;

    return cnt;
}
```cpp
void EncoderChip::clear()
{
    if (_id < 0) {
        return;
    }

    // Set encoder's data register to 0
    digitalWrite(_ssPin, LOW); // Begin SPI conversation
    // Write to DTR
    SPI.transfer(0x98);
    // Load data
    SPI.transfer(0x00); // Highest order byte
    SPI.transfer(0x00);
    SPI.transfer(0x00);
    SPI.transfer(0x00); // lowest order byte
    digitalWrite(_ssPin, HIGH); // Terminate SPI conversation

    delayMicroseconds(100); // provides some breathing room between SPI conversations

    // Set encoder current data register to center
    digitalWrite(_ssPin, LOW); // Begin SPI conversation
    SPI.transfer(0xE0);
    digitalWrite(_ssPin, HIGH); // Terminate SPI conversation
}
```

Listing 5: Encoder Chip Class Source

//==================================HEADER
---

/*
Definition for home+ lamp: CuRLE
*/

141
This header contains all of the definitions for the pin mappings for the lamp

License: CCAv3.0 Attribution—ShareAlike (http://creativecommons.org/licenses/by-sa/3.0/)

You’re free to use this code for any venture. Attribution is greatly appreciated.

/**
   //
   ==============================================================

#ifndef __LAMP__
#define __LAMP__

// Frequencies for PWM
#define ENC_CLK_PERIOD 20   // in 1e-8s ==> period=200ns, freq=5MHz
#define ENC_CLK_DUTY 10     // in 1e-8s ==> duty=100ns, duty_cycle =50%
#define SERVO_PWM_PERIOD 2000000 // in 1e-8s ==> period=20ms, freq=50Hz

// Other Constants
#define ON 1
#define OFF 0
#define M_CNT 7
#define SPOOL_RADIUS_BASE (0.0125) // m
#define WHEEL_RADIUS (0.042) // m
#define ROBOT_RADIUS (0.223) // m

// Action "States"
#define MU_PREP (−1)
#define MU_ROT1 0
#define MU_TRAN 1
#define MU_ROT2 2
#define MU_IDLE 3

// Opcodes for Raspberry Pi Messages
#define RPI_LOAD 0
#define RPI_SAVE 1
#define RPI_SETPOINT 2
#define RPI_STATE 3
#define RPI_ECOUNT 4
#define RPI_GAIN 5
#define RPI_COMPLETE 6
#define RPI_STATUS 7
#define RPI_READY 8

// For Motor List Indexing
#define MOTOR1 0
#define MOTOR2 1
#define MOTOR3 2
#define MOTOR4 3
#define MOTOR5 4
#define MOTOR6 5
#define MOTOR7 6
/ * ==================================================== *
/* ====== PINS ====== *
/* ==================================================== *

// Reset Pin
#define RESET_PIN A0

// Tendon Motor Pins
#define T_MOTOR1 MOTOR1
#define T_MOTOR1_PWM 11
#define T_MOTOR1_DIR 29
#define T_MOTOR1_ENC 30
#define T_MOTOR1_CPR 12659

#define T_MOTOR2 MOTOR2
#define T_MOTOR2_PWM 2
#define T_MOTOR2_DIR 35
#define T_MOTOR2_ENC 32
#define T_MOTOR2_CPR 12659

#define T_MOTOR3 MOTOR3
#define T_MOTOR3_PWM 4 /* 10 */
#define T_MOTOR3_DIR 39 /* 37 */
#define T_MOTOR3_ENC 36 /* 34 */
#define T_MOTOR3_CPR 12659

/* due to wiring, had to switch motor3 and motor4 */
#define T_MOTOR4 MOTOR4
#define T_MOTOR4_PWM 10 /* 4 */
#define T_MOTOR4_DIR 37 /* 39 */
#define T_MOTOR4_ENC 34 /* 36 */
#define T_MOTOR4_CPR 12659
// Drive Motor Pins
#define D_MOTOR1 MOTOR5
#define D_MOTOR1_PWM 12
#define D_MOTOR1_DIR 31
#define D_MOTOR1_ENC 38
#define D_MOTOR1_CPR 3200

#define D_MOTOR2 MOTOR6
#define D_MOTOR2_PWM 5
#define D_MOTOR2_DIR 41
#define D_MOTOR2_ENC 40
#define D_MOTOR2_CPR 3200

// Worm Gear Motor Pins
#define W_MOTOR MOTOR7
#define W_MOTOR_PWM 3
#define W_MOTOR_DIR 43
#define W_MOTOR_ENC 42
#define W_MOTOR_CPR (6533*30)

// Linear Actuator Pins
#define L_MOTOR 0
#define L_MOTOR_PWM 13
#define L_MOTOR_DIR 33
#define L_MOTOR_IR A5

// Gripper Servo Pin
#define WRIST_PIN 44
#define CLAW_PIN 45
Listing 6: Macros

// Tension Sensors
#define T_SENSOR1 A1
#define T_SENSOR2 A2
#define T_SENSOR3 A3 /* A2 */
#define T_SENSOR4 A4 /* A3 */

// Switches (Relay Control Pins)
#define SW_LED 50
#define SW_DRV 52

/* ============================================================== */

#endif /* __LAMP__ */
/*

#ifndef __LINEAR_ACTUATOR_H__
#define __LINEAR_ACTUATOR_H__

#include <inttypes.h>
#include <Wire.h>
#include "Arduino.h"

class LinearActuator {
public:
    LinearActuator(int dirPin, int pwmPin, int irPin);
    LinearActuator() = delete;
    LinearActuator(const LinearActuator&) = delete;

    LinearActuator& operator=(const LinearActuator&) = delete;

    void run(int val);
    int length();

private:
    int _dirPin;
    int _pwmPin;
    int _irPin;
    int _zeroPt;
    int _read();
};
*/
Listing 7: Linear Actuator Class Header

```cpp
#include "linear_actuator.h"

LinearActuator::LinearActuator(int dirPin, int pwmPin, int irPin):
    _dirPin(dirPin),
    _pwmPin(pwmPin),
    _irPin(irPin),
    _zeroPt(0.0)
{
    pinMode(_dirPin, OUTPUT);
    digitalWrite(_dirPin, LOW);

    pinMode(_pwmPin, OUTPUT);
    analogWrite(_pwmPin, 0);

    pinMode(_irPin, INPUT);
    _zeroPt = _read();
}

void LinearActuator::run(int val)
{
    // wind() unwind()
    int dir = (val <= 0) ? LOW : HIGH;
    int pwm = abs(val);

    // direction
    digitalWrite(_dirPin, dir);

    // speed
```
Listing 8: Linear Actuator Class Source

```cpp
int LinearActuator::length()
{
    int tmp = _read() - _zeroPt;

    // TODO: convert the raw analog "int" to a distance

    return tmp;
}

int LinearActuator::_read()
{
    int tmp, i, cnt;
    cnt = 5;
    tmp = 0;
    for (i = 0; i < cnt; ++i) {
        tmp += analogRead(_irPin);
    }
    return tmp / cnt;
}
```

// ===============HEADER

=================================HEADER

/*
Encoded DC Gear-Motor Library
AUTHOR: Zach Hawks
DATE: March 13, 2019
*/
This is a simple library to interface with a DC gear-motor with a quadrature encoder, controlled by a dual H-bridge (DIR and PWM).

License: CC BY 3.0 Attribution-ShareAlike (http://creativecommons.org/licenses/by-sa/3.0/)
You're free to use this code for any venture. Attribution is greatly appreciated.

//============================================

/*

#ifndef ___MOTOR_H___
#define ___MOTOR_H___

#include <inttypes.h>
#include <Wire.h>
#include "Arduino.h"
#include "encoder_chip.h"

class Motor {
public:
    Motor(int dirPin, int pwmPin, int cntPerRev, int encSlaveSelectPin);
    Motor(const Motor& m);

    Motor();
    Motor& operator=(const Motor&)= delete;

    void init();
    void run(int val);
    void stop();

#endif
*/
```cpp
int32_t getCount() { return _encoder.read(); }

int CPR() const { return _cntPerRev; }

int getid() const { return _id; }

String string() const;

private:
  static int ID;
  int _id;
  EncoderChip _encoder;
  int _cntPerRev;
  int _dirPin;
  int _pwmPin;
};

# endif // _MOTOR_H_
```

Listing 9: Motor Class Header

```cpp
#include "motor.h"

// motor index
int Motor::ID = 0;

Motor::Motor(int dirPin, int pwmPin, int cntPerRev, int encSlaveSelectPin) :
  _id(ID++),
  _encoder(encSlaveSelectPin),
  _cntPerRev(cntPerRev),
  _dirPin(dirPin),
  _pwmPin(pwmPin)
{
```

151
```cpp
Motor::Motor(const Motor& m) :
    _id(m._id),
    _encoder(m._encoder),
    _cntPerRev(m._cntPerRev),
    _dirPin(m._dirPin),
    _pwmPin(m._pwmPin)
{
}

Motor::Motor() :
    _id(-1),
    _encoder(),
    _cntPerRev(1),
    _dirPin(-1),
    _pwmPin(-1)
{
}

void Motor::init()
{
    if (_id < 0) {
        return;
    }
    pinMode(_dirPin, OUTPUT);
    digitalWrite(_dirPin, LOW);
    pinMode(_pwmPin, OUTPUT);
    analogWrite(_pwmPin, 0);
    _encoder.init();
}
encoder.clear();

String Motor::string() const
{
    if (_id < 0) {
        return "invalid";
    }
    String s = "/motor["+String(_id+1)+"] [dir="+String(_dirPin)+"
    s = s + ", pwm="+String(_pwmPin)+", cpr="+String(_cntPerRev)+", enc="+
    String(_encoder.getPin())
    return s;
}

void Motor::stop()
{
    run(0);
}

void Motor::run(int val)
{
    if (_id < 0) {
        return;
    }
    // direction
    if (val <= 0) {
        digitalWrite(_dirPin, LOW); // wind()
    }
    else {
        digitalWrite(_dirPin, HIGH); // unwind()
    }
Listing 10: Motor Class Source

```c
// speed
analogWrite(_pwmPin, abs(val));
```

// =========================HEADER

/*
* Encoded DC Gear-Motor List Library
* AUTHOR: Zach Hawks
* DATE: March 15, 2019
*
This is a simple class to hold multiple Motor objects

License: CC BY-NC-SA 3.0 Attribution—ShareAlike (http://creativecommons.org/licenses/by-sa/3.0/)
You're free to use this code for any venture. Attribution is greatly appreciated.
*/

//=================================HEADER=====================================

#ifndef __MOTOR_LIST_H__
#define __MOTOR_LIST_H__

#include <inttypes.h>
#include <Wire.h>
#include "Arduino.h"
#include "motor.h"
```
```cpp
class MotorList {
public:
    MotorList();

    MotorList(const MotorList& ml) = delete;
    MotorList& operator=(const MotorList&) = delete;

    Motor& operator[](int i);
    void init();
    void stopall();

private:
    Motor _invalid;
    Motor _tMotor1;
    Motor _tMotor2;
    Motor _tMotor3;
    Motor _tMotor4;
    Motor _dMotor1;
    Motor _dMotor2;
    Motor _wMotor;
    int _length;
};
```

Listing 11: Motor List Class Header

```bash
#include "motor_list.h"
#include "lamp_def.h"

MotorList::MotorList() :
    _invalid(),
```
MotorList::init()
{
    
    
    void MotorList::operator[](int i)
{
    switch(i) {
        case 0:
            return _tMotor1;
        case 1:
            return _tMotor2;
        case 2:
            return _tMotor3;
        case 3:
            return _tMotor4;
        case 4:
            return _dMotor1;
        case 5:
            return _dMotor2;
        case 6:
            return _wMotor;
    }
return _tMotor3;
case 3:
  return _tMotor4;
case 4:
  return _dMotor1;
case 5:
  return _dMotor2;
case 6:
  return _wMotor;
default:
  return _invalid;
}

void MotorList::stopAll()
{
  _tMotor1.stop();
  _tMotor2.stop();
  _tMotor3.stop();
  _tMotor4.stop();
  _dMotor1.stop();
  _dMotor2.stop();
  _wMotor.stop();
}

Listing 12: Motor List Class Source

/=============================HEADER

Library for running sum/average
AUTHOR: Zach Hawks
This is a simple class/library that keeps a running sum/average

License: CCAv3.0 Attribution—ShareAlike (http://creativecommons.org/licenses/by-sa/3.0/)
You’re free to use this code for any venture. Attribution is greatly appreciated.

template<class T>
class RunningSum {
public:
  RunningSum();
  RunningSum(int qty);
  ~RunningSum();

  RunningSum& operator=(const RunningSum& rs) = delete;
  RunningSum(const RunningSum<T>& rs) = delete;

  void init(int qty);
```cpp
T add(T item);  // Adds an item to the queue.

T sum() const { return _sum; }  // Returns the current sum.

int capacity() const { return _size; }  // Returns the capacity of the queue.

int size() const { return _cnt; }  // Returns the current size.

void clear();  // Clears the queue.

float avg() const;  // Returns the average of the items.

private:
    int _size;
    T* _queue;
    int _cnt;
    int _ptr;
    T _sum;
};
```

---

Listing 13: Running Sum Class Header

```cpp
#include "running_sum.h"

template <class T>
RunningSum<T>::RunningSum() : _size(-1), _queue(nullptr), _cnt(-1), _ptr(-1), _sum(0)
{
}

template <class T>
RunningSum<T>::RunningSum(int qty) : _size(qty), _queue(nullptr), _cnt(0), _ptr(0), _sum(0)
{
    _queue = (T*)calloc(_size, sizeof(T));
}
```
template <class T>
RunningSum<T>::~RunningSum()
{
    if (_size != -1) {
        free(_queue);
    }
}

template <class T>
void RunningSum<T>::init(int qty)
{
    _size = qty;
    _queue = (T*)calloc(_size, sizeof(T));
    _cnt = 0;
    _ptr = 0;
    _sum = 0;
}

template <class T>
T RunningSum<T>::add(T item)
{
    if (_size <= 0) {
        return _sum;
    }
    _sum = _sum - _queue[_ptr] + item;
    _queue[_ptr] = item;

    _ptr = (_ptr+1)%_size;
    _cnt = max(_cnt+1, _size);
    return _sum;
template <class T>
void RunningSum<T>::clear()
{
    if (_size < 0) {
        return;
    }
    memset(_queue, 0, _size*sizeof(T));
    _cnt = 0;
    _ptr = 0;
    _sum = 0;
}

template <class T>
float RunningSum<T>::avg() const
{
    if (_cnt <= 0) {
        return 0;
    }
    return (static_cast<float>(_sum) / static_cast<float>(_cnt));
}

Listing 14: Running Sum Class Source
This is a simple library program to read "tension" from a spring loaded linear potentiometer.

License: CCAv3.0 Attribution—ShareAlike (http://creativecommons.org/licenses/by-sa/3.0/)
You’re free to use this code for any venture. Attribution is greatly appreciated.

//============================================================================

*/

#ifndef __TENSIONSENSOR_H__
#define __TENSIONSENSOR_H__

#include <inttypes.h>
#include <Wire.h>
#include "Arduino.h"

class TensionSensor {

public:

    TensionSensor();
    TensionSensor(int pin);
    TensionSensor(const TensionSensor&) = delete;

    TensionSensor& operator=(const TensionSensor&) = delete;

    int init(int pin);
    int init();
    int read();
Listing 15: Tension Sensor Class Header

```cpp
#include "tension_sensor.h"

TensionSensor::TensionSensor() : _pin(-1)
{
    // pinMode(_pin, INPUT);
}

TensionSensor::TensionSensor(int pin) : _pin(pin)
{
    // pinMode(_pin, INPUT);
}

int TensionSensor::init(int pin)
{
    _pin = pin;
    pinMode(_pin, INPUT);
}

int TensionSensor::init()
{
    if (_pin == -1) {
        return -1;
    }
    pinMode(_pin, INPUT);
}
```
Listing 16: Tension Sensor Class Source

Appendix B  CuRLE Robot Software: Raspberry Pi

```python
import signal
import serial
import time
import sys
import socket
import select
import json
import math

global _serial, _socket, Kgain

def cleanup(sig, fram):
    _socket.close()
    _serial.close()
```
with open("/home/pi/lamp/gain.json", "w") as f:
    json.dump(Kgain, f, indent=4)

print("exiting ...")
sys.exit(0)

def reply_arduino(msg):
    word = msg.split('/')
    if word[0] == "save:
        _string = word[1]
        _string = _string[:-1] + ";" + word[2]
        save_state(_string)
        #
        elif word[0] == "print:
            print("arduino -> ", word[1])
        elif word[0] == "load:
            state = load_state()
            text = replyForLoad(state, Kgain)
            if text[-1] != ';':
                text = text + ';
            _serial.write(text.encode('utf-8'))
        elif word[0] == "complete:
            print("setpoint reached")
        elif word[0] == "noop:
            print("arduino goofed...")

def relay_arduino(cmd):
    if cmd[-1] != ';':
cmd = cmd + ';

tmp = cmd[:-1].split(':')

if tmp[0] == "tendon_gain":
    K = tmp[1].split(',
    i = 0
    for k in K:
        Kgain["K_tendon"][i] = float(k)
        if (len(K) == 3):
            Kgain["K_tendon"][i+3] = float(k)
            i = i+1
    elif tmp[0] == "drive_gain":
        K = tmp[1].split(',
        i = 0
        for k in K:
            Kgain["K_drive"][i] = float(k)
            i = i + 1

    print('sending -> ', cmd)
    _serial.write(cmd.encode('utf-8'))

def load_state():
    with open("/home/pi/lamp/lamp.json", "r") as f:
        data = f.read()
        #print(data)
        o = json.loads(data)
        return o

def save_state(state):
    cmds = state.split(';
    if (len(cmds) != 3):
        print("has to be 2 cmds...")
return

# "state": { "u": _, "v": _, "w": _, "raw": [_, _, _] }

tmp = cmds[0].split(' ;')
if (tmp[0] != 'state '):
    print( "state: command not present...")
return

raw = tmp[1].split( ' , ')
u = float( raw[0])
v = float( raw[1])
w = float( raw[2])
raw = [u, v, w]

# "spool_rad": [ _r1 , _r2 , _r3 , _r4 ]
tmp = cmds[1].split(' ;')
if (tmp[0] != 'encoder '):
    print("encoder: command not present...")
return
cnts = tmp[1].split(' , ')
enc = []
for c in cnts:
    enc.append(int(c))

jstring = json.dumps( { "state": { "u": u, "v": v, "w": w, "raw": raw }, "encoder": enc }, indent=4)

# print( jstring )

with open( " / home / pi / lamp / lamp . json", "w") as f:
    json.dump( { "state": { "u": u, "v": v, "w": w, "raw": raw }, "encoder": enc }, f, indent=4)

return
def load_gain():
    with open("/home/pi/lamp/gain.json", "r") as f:
        data = f.read()
        #print(data)
        o = json.loads(data)
        return o

def replyForLoad(data, gain):
    state = data["state"]
    raw = state["raw"]
    _string = "state:"+"%0.4f"%state["u"]+","
    _string += "%0.4f"%state["v"]+","
    _string += "%0.4f"%state["w"]+"/

    _string1 = "encoder:
    for r in data["encoder"]:
        _string1 = _string1 + str(r)+","
    _string1 = _string1[:-1] + '/'

    _string2 = "tendon_gain:
    for k in gain["K_tendon"]:
        _string2 = _string2 + str(k)+","
    _string2 = _string2[:-1] + '/'

    _string3 = "drive_gain:
    for k in gain["K_drive"]:
        _string3 = _string3 + str(k)+","
    _string3 = _string3[:-1] + '/'

    _string4 = "goto:"+"%0.4f"%state["u"]+","
    _string4 += "%0.4f"%state["v"]+","
_string4 += "%0.4f"%state["w"]+"/

_string5 = "init:;"

return _string+_string1+_string2+_string3+_string4+_string5

signal.signal(signal.SIGINT, cleanup)

TIMEOUT = 3

# Setup serial comm
_serial = serial.Serial(port='/dev/ttyUSB0', baudrate=9600, timeout=TIMEOUT)
msg = b''

# Setup wireless comm
PORT = 16996
_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
_socket.setblocking(0)
_socket.bind((' ',PORT))
_socket.listen(2)

Kgain = load_gain()

read_list = [_socket, sys.stdin]
#read_list = [_socket]
connected = False
while True:
    readable, writable, errored = select.select(read_list, [], [], TIMEOUT)
for s in readable:
    # new connection available
    if s is _socket:
        _client, addr = _socket.accept()
        read_list.append(_client)
        connected = True
    # existing connection has data available
    elif s is sys.stdin:
        userIn = sys.stdin.readlines()
        for line in userIn:
            print("echo -> ", line[\-1])
            if line == '\n':
                line = '; ;'
            relay_arduino(line[\-1])
    else:
        cmd = s.recv(1024)
        print("recv'd (client) -> ", cmd)
        if (cmd):
            relay_arduino(cmd.decode("utf-8"))
        else:
            s.close()
            read_list.remove(s)
            connected = False

# Read data from Arduino
while (_serial.in_waiting):
    c = _serial.read(1)
    msg += c
    if (c == b'; '):
        print("recv'd -> (arduino)", msg.decode("utf-8"))
        if (connected):
Listing 17: Python script that controls the robot running on Raspberry Pi

```python
import socket
import sys
import select
import signal

def cleanup(sigint, frame):
    _socket.close()
    print('exiting...')
    sys.exit(0)

signal.signal(signal.SIGINT, cleanup)

keepGoing = True
HOST = '198.21.183.61'
PORT = 16996
TIMEOUT = 5

_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
_socket.settimeout(TIMEOUT)
try:
    _socket.connect((HOST, PORT))
```

Appendix C  CuRLE Robot Software: Central Computer
read_list = [socket]

_socket.sendall('init;'.encode('utf-8'))

while keepGoing:
    shouldReply = False
    # read replies
    readable, writable, errored = select.select(read_list, [], [], TIMEOUT)
    for s in readable:
        msg = s.recv(1024)
        print "recv'd (raspi) -> " , msg
        shouldReply = True

    if (shouldReply):
        text = raw_input('reply -> ')
        if (text[0] == '/'):
            keepGoing = False
        if (text[-1] != ';'):
            text = text + ';'  
        _socket.sendall(text.encode('utf-8'))

except Exception as e:
    print 'exception ->', e

finally:
    _socket.close()

Listing 18: Python script to communicate RRT output to CuRLE
Appendix D  Motion Planning Software

```cpp
#ifndef __CONFIG_SPACE_H__
define __CONFIG_SPACE_H__

#include "RobotState.h"
#include <vector>

class ConfigSpace{

public:
    ConfigSpace(std::string fname, int d);
    ~ConfigSpace();

    int collisionPoints() const;
    bool collidedState(RobotState &st);
    bool collidedEdge(RobotState &st1, RobotState &st2);

private:
    const double _pi = 3.1415926535897;
    int dim;
    int *size;
    int total_size;
    uint8_t *cspace;

    ConfigSpace() = delete;
    ConfigSpace(const ConfigSpace &) = delete;
    ConfigSpace& operator=(const ConfigSpace &) = delete;
};
```

173
Listing 19: ConfigSpace Class header

```cpp
#include "ConfigSpace.h"

#include <iostream>
#include <string>
#include <fstream>

ConfigSpace::ConfigSpace(std::string fname, int d) : dim(2), size(new int[dim]), total_size(1), cspace(nullptr)
{
    int _dim;
    FILE *fptr;
    char buf[256];
    fopen_s(&fptr, fname.c_str(), "rb");
    if (fptr == nullptr) {
        std::cout << "Could not open file: " << fname << std::endl;
        return;
    }
    fgets(buf, 256, fptr);
    sscanf(buf, "%d %d %d", &_dim, &size[0], &size[1]);
    for (int i = 0; i < dim; ++i) {
        total_size *= size[i];
    }
    cspace = new uint8_t[total_size];
    fread(cspace, 1, total_size, fptr);
    fclose(fptr);
}

ConfigSpace::~ConfigSpace()
{  
    // Deallocate memory
    delete[] cspace;
    delete[] size;
}
```

174
bool ConfigSpace::collidedState(RobotState &st) {
    // since C-Space is only 2–dimensional, only worry about first 2 state variables (X = [x1, x2, ...., xn])
    double x1, x2;
    int d0, d1;  // d0 -> "row", d1 -> "col"
    st.get(0, x1);
    st.get(1, x2);
    // theta is stored in x1, and this is the "column" of the "image"
    // d1 = static_cast<int>((x1 + _pi) * 100); // resolution is 0.01
    d1 = (int)std::roundl(x1 * 100) + 314;
    // u is stored in x2, and this is the "row" of the "image"
    // d0 = static_cast<int>((x2 + _pi) * 100); // resolution is 0.01
    d0 = (int)std::roundl(x2 * 100) + 314;

    // size[1] -> COLS, size[0] -> ROWS
    return (cspace[d0*size[1] + d1] == 1);
}

bool ConfigSpace::collidedEdge(RobotState &st1, RobotState &st2) {
    // Sample along the edge from both directions
    const double DELTA = 0.001; // radians (~3 degrees)
    double dist = st1.euclidDist(st2);
    double cnt = 0.0;
    // get the unit vector
    RobotState v = ((st1) - (st2));
RobotState n = v / v.magnitude();

RobotState sample1 = (st1) - n*DELTA;
RobotState sample2 = (st2) + n*DELTA;

while (cnt <= dist / 2) {
    if (collidedState(sample1) || collidedState(sample2)) {
        return true;
    }
    sample1 = sample1 - n*DELTA;
    sample2 = sample2 + n*DELTA;
    cnt += DELTA;
}
return false;

int ConfigSpace::collisionPoints() const
{
    if (cspace == nullptr) {
        return -1;
    }
    int cnt = 0;
    for (int i = 0; i < total_size; ++i) {
        if (cspace[i] == 1) {
            cnt++;
        }
    }
    return cnt;
}

Listing 20: ConfigSpace Class source

#include "MotionPlanner.h"
```cpp
#include "RobotCom.h"
#include "StopWatch.h"

ostream& operator <<(ostream& out, const RobotState& s);

void printActionPath(ostream& out, const RobotState& s);

int main(int argc, char* argv[]) {
    MotionPlanner& planner = MotionPlanner::getInstance();
    bool unitTesting = false;
    int seed = 101;
    // std::string configFileName("./RRTPlanner/config/config2a");
    std::string configFileName("./RRTPlanner/config/config2b");
    std::string configFileName("./RRTPlanner/config/config1a");
    std::string configFileName("./RRTPlanner/config/config1b");
    std::string configFileName("./RRTPlanner/config/config1c");
    StopWatch<std::chrono::milliseconds> stopwatch;
    RobotCom comm(1011, "192.168.0.11", 50);
    std::vector<RobotAction> actions;
    std::ofstream csvfile, datfile;
    std::ifstream unittestfile;
    std::stringstream strm;
    int bytes, minNodes, lookForGoalAfter, nodeLimit;
    double deltax, thetax, thetaOpt, maxDist, minDist;
    bool success, run;
    MotionPlanner::RRTtype type;
    std::string output, buffer, fname;

    bytes = 0;
    thetax = thetaOpt = deltax = 0.0;
    if (unitTesting) {
        auto inc = [](MotionPlanner::RRTtype t) {
            return t;
        }
    }
```
if (t == MotionPlanner::RRTtype::VERTEX) { return MotionPlanner::RRTtype::EXT_VERTEX; }

if (t == MotionPlanner::RRTtype::EXT_VERTEX) { return MotionPlanner::RRTtype::EDGE; }

return MotionPlanner::RRTtype::VERTEX;
};

datfile.open("analysis/experiment_data.csv", std::ios::out | std::ios::app);
if (!datfile.is_open()) { std::cout << "ERROR!" << std::endl; }
else {

datfile << "seed,type,success,vlimit,goal_look,min_nodes,num_nodes,num_misses,t_RRT,t_A*,pathNodes,pathDist,thetaDist,thetaOpt\n";

run = true;
type = MotionPlanner::RRTtype::VERTEX;

while (run) {
    lookForGoalAfter = 50;
    minNodes = 75;
    nodeLimit = 150;
    maxDist = 30.48;
    minDist = 15.24;
    // 20 loops

    for (seed = 1111; seed < 1331; seed += 11) {
        planner.configure(configFileName);
        stopwatch.start();
        success = planner.runRRT(type, maxDist, minDist, lookForGoalAfter, nodeLimit, minNodes);
        stopwatch.stop();
        auto rrtTime = stopwatch.readback();
        stopwatch.save(std::string("rrt_time"));
        if (success) {
            datfile << "seed," << seed << "," << type << "," << success << "," << vlimit << "," << goal_look << "," << min_nodes << "," << num_nodes << "," << num_misses << "," << t_RRT << "," << t_A* << "," << pathNodes << "," << pathDist << "," << thetaDist << "," << thetaOpt << ":" << rrtTime << "," << std::endl;
        }
    }
}
59

stopwatch . s t a r t ( ) ;

60

actions = planner . runAstar () ;

61

stopwatch . stop ( ) ;

62

stopwatch . save ( std : : s t r i n g ( ”a* time ” ) ) ;

63

// theta = getTotalRotation ( actions ) ;

64

planner . optimizeActions ( actions ) ;

65

/ / thetaOpt = getTotalRotation ( actions ) ;

66

// delta = getTotalTranslation ( actions ) ;

67

}

68

d a t f i l e << s e e d << ” , ” << t y p e << ” , ” << s u c c e s s << ” , ” <<
n o d e L i m i t << ” , ” ;

69

d a t f i l e << l o o k F o r G o a l A f t e r << ’ , ’ << minNodes << ’ , ’ ;

70

d a t f i l e << p l a n n e r . getNumV ( ) << ” , ” << p l a n n e r .
getNumMissedNodes ( ) << ” , ” ;

71

d a t f i l e << r r t T i m e . c o u n t ( ) << ” , ” ;

72

d a t f i l e << s t o p w a t c h . r e a d b a c k ( ) . c o u n t ( ) << ” , ” ;

73

d a t f i l e << a c t i o n s . s i z e ( ) << ” , ” << d e l t a << ” , ” << t h e t a << ”
, ” << t h e t a O p t << s t d : : e n d l ;

74

planner . resetGraph () ;

75

actions . clear () ;

76
77

}

78

type = inc ( type ) ;

79

r u n = ! ( t y p e == M o t i o n P l a n n e r : : RRTtype : : VERTEX) ;

80

}

81

d a t f i l e . close () ;
}

82
83
84

}

85

else {

86

a u t o r a d i a n s = [ ] ( d o u b l e deg )−>d o u b l e { c o n s t d o u b l e

179

pi =


3.14159265; return (deg / 180.0 * _pi); }
planner.configure(configFileName);

stopwatch.start();
planner.runRRT(MotionPlanner::RRTtype::EXT_VERTEX, radians(40), radians(10), 25, 100, 30);
stopwatch.stop();
stopwatch.save(std::string("rrt_time"));

stopwatch.start();
actions = planner.runAstar();
stopwatch.stop();
stopwatch.save(std::string("a*_time"));

planner.optimizeActions(actions);

output = planner.buildcsv();
strm.str(std::string()); strm.clear();
strm << "analysis/path.csv";
fname = strm.str();
csvfile.open(fname.c_str(), std::ios::out | std::ios::trunc);
if (!csvfile.is_open())
    std::cout << "ERROR!" << std::endl;
else {
    csvfile << output << std::endl;
    csvfile.close();
}
/* csvfile.open("analysis/experiment_log.csv", std::ios::out | std::ios::app);
if (!csvfile.is_open()) { std::cout << "ERROR!" << std::endl; }"
else {
    csvfile << "Experiment.seed," << seed << ", configFile," << configFileName << std::endl;
    csvfile << "type,maxDistThresh,minDistThresh,nodesBeforeGoal,,
    nodeLimit,minNodes" << std::endl;
    csvfile << planner.getRRTdescriptor() << std::endl;
    csvfile << "nodes," << planner.getNumV();
    csvfile << ",misses," << planner.getNumMissedNodes() << std::endl;
    csvfile << ",time[ms].RRT," << stopwatch.recall(std::string("" 
    rrt_time\"\")).count();
    csvfile << ",A*," << stopwatch.recall(std::string("" 
    a*_time\"\")).count() << std::endl;
    csvfile << "file," << fname << std::endl << std::endl;
    csvfile.close();
} /*
strm.str(std::string()); strm.clear();

// printActionPath(actions, true);
//comm.connectToRobot();

// send to the robot
if (comm.isConnected()) {
    // strm << ",init\:\n\n];
    // bytes = comm.sendMsg(strm.str());
    // if (bytes < 0) { std::cout << "error sending on init" << std::endl; 
    // return 1; } } 
    // strm.str(std::string()); strm.clear();
    // //bytes = comm.recvMsg(buffer);
    // //std::cout << buffer << std::endl;
    // std::size_t i = 0;
    // std::chrono::milliseconds tnow = stopwatch.readclock();
141 // std::chrono::milliseconds alarm = tnow + std::chrono::milliseconds(500);
142 // while (tnow < alarm) { tnow = stopwatch.readclock(); }
143 // while (bytes > 0 && i < actions.size()) {
144 // // compose
145 // tnow = stopwatch.readclock();
146 // if (tnow > alarm) {
147 // strm << "action:" << actions[i].getRotation1() << ",";
148 // strm << actions[i].getTranslation() << ",";
149 // strm << actions[i].getRotation2() << "\n";
150 // std::cout << strm.str();
151 // bytes = comm.sendMsg(strm.str());
152 // if (bytes < 0) { std::cout << "error sending on index " << i
153 << std::endl; break; }
154 // buffer = (std::string());
155 // bytes = comm.recvMsg(buffer);
156 // if (bytes >= 0)
157 // std::cout << buffer << " [" << bytes << "]" << std::endl;
158 // strm.str(std::string()); strm.clear();
159 // ++i;
160 // alarm = tnow + std::chrono::milliseconds(1000);
161 // }
162 // }
163 // comm.closeConnection();
164 }
165 else {
166 std::cout << "comm error" << std::endl;
167 }
168 }
169 std::cout << "complete...
getchar();
```cpp
return 0;
}

double getTotalTranslation(std::vector<RobotAction> actions) {
    double delta = 0.0;
    for (std::size_t i = 0; i < actions.size(); ++i) {
        // delta += actions[i].getTranslation();
    }
    return delta;
}

double getTotalRotation(std::vector<RobotAction> actions) {
    double theta = 0.0;
    for (std::size_t i = 0; i < actions.size(); ++i) {
        // theta += std::fabs(actions[i].getRotation1()) + std::fabs(actions[i].getRotation2());
    }
    return theta;
}

// print action vector
void printActionPath(std::vector<RobotAction> actions, bool toFile) {
    std::stringstream strm;
    std::ofstream file;
    if (toFile) {
        file.open("analysis/action_list.csv");
        if (!file.is_open()) {
            std::cout << "error opening file... using std::cout instead" << std::endl;
            toFile = false;
        }
    }
```
if (toFile) {
    for (std::size_t i = 0; i < actions.size(); ++i) {
        // strm << actions[i].getRotation1() << ",";
        // strm << actions[i].getTranslation() << ",";
        // strm << actions[i].getRotation2() << "\n";
        // file << strm.str();
        // strm.str(std::string()); strm.clear();
    }
}

else {
    for (std::size_t i = 0; i < actions.size(); ++i) {
        // strm << "action:" << actions[i].getRotation1() << "\n";
        // strm << actions[i].getTranslation() << "\n";
        // strm << actions[i].getRotation2() << "\n";
        // std::cout << strm.str();
        // strm.str(std::string()); strm.clear();
    }
}

if (toFile) {
    file.close();
}

Listing 21: Main Execution Function

#ifndef __MOTION_PLANNER_H__
#define __MOTION_PLANNER_H__
#include <boost/config.hpp>
#include <fstream>
#include <iostream>
#include <string>
```cpp
#include <sstream>
#include <random>
#include <functional>
#include <utility>
#include <deque>
#include <chrono>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_utility.hpp>
#include <boost/property_map/property_map.hpp>
#include <boost/tuple/tuple.hpp>

#include "RobotAction.h"
#include "ConfigSpace.h"
#include "RobotState.h"
#include "PriorityQueue.h"

class RobotAction;
class ConfigSpace;
class RobotState;
class PriorityQueue;
class MotionPlanner;

namespace RRT {
    const int DIM = 3;

    struct VertexProperties {
        std::size_t index;
        RobotState *state;
        double heuristic;
        double cost;
    };
}
struct EdgeProperties {
    double dist;
    RobotAction *action;
};
typedef boost::adjacency_list<boost::vecS, boost::listS, boost::directedS, VertexProperties, EdgeProperties> Graph;
typedef boost::graph_traits<Graph>::vertex_descriptor Vertex;
typedef boost::graph_traits<Graph>::edge_descriptor Edge;
typedef boost::graph_traits<Graph>::vertex_iterator V_iter;
typedef boost::graph_traits<Graph>::out_edge_iterator E_iter;
typedef std::map<std::pair<std::size_t, std::size_t>, RobotState> EdgeMeasured;

class MotionPlanner { /* Meyers Singleton Class */
public:
    // Functor
    class LowerCost {
    public:
        bool operator()(RRT::Vertex v1, RRT::Vertex v2) {
            double tmp = MotionPlanner::getInstance().getCost(v1);
            double tmp2 = MotionPlanner::getInstance().getCost(v2);
            return (tmp <= tmp2);
        }
    };
    enum RRTtype { VERTEX=0, EXT_VERTEX, EDGE };
    static MotionPlanner& getInstance();
    ~MotionPlanner();
void configure(std::string configFile);
std::string buildcsv();

MotionPlanner(const MotionPlanner&) = delete;
MotionPlanner& operator=(const MotionPlanner&) = delete;

double getCost(RRT::Vertex v) { if (configured) { return v_cost[v]; } else { return 0.0; } }
int getNumV() const { return numv; }
int getNumMissedNodes() const { return missedNodes; }
std::string getRRTdescriptor() const { return descriptor; }

bool runRRT(RRTtype type, double maxDistThresh, double minDistThresh,
            int numBeforeLookForGoal, int vLimit, int minNodes);
void resetGraph();
std::vector<RobotAction> runAstar(/*params*/);
void optimizeActions(std::vector<RobotAction>& actions);

private:
  MotionPlanner();

// Scenario parameters
RobotState startState;
RobotState goalState;
std::size_t goalId;
ConfigSpace *cSpace;

// Random generator params
std::random_device rd;
std::mt19937 gen;
std::uniform_real_distribution<> disTh;
std::uniform_real_distribution<> disU;
std::uniform_real_distribution<> disV;

// Booleans (state vars)
bool goalIsConnected;
bool failedToConnectToGoal;
bool inserted;

// Graph variables
RRT::Graph rrt;
std::string descriptor;
int numv;
int missedNodes;
bool configured;
RRT::Vertex vptr, vptr2, vstart;
RRT::Vertex src, tgt;
RRT::Edge eptr, eptr2, estart;
boost::property_map<RRT::Graph, std::size_t RRT::VertexProperties::*>::type v_id;
boost::property_map<RRT::Graph, RobotState* RRT::VertexProperties::*>::type v_state;
boost::property_map<RRT::Graph, double RRT::VertexProperties::*>::type v_heur;
boost::property_map<RRT::Graph, double RRT::VertexProperties::*>::type v_cost;
boost::property_map<RRT::Graph, RobotAction* RRT::EdgeProperties::*>::type e_action;
RRT::V_iter vi, viend;
RRT::E_iter ei, eiend;
// A* variables
bool* closedList;
PriorityQueue<RRT::Vertex, std::deque<RRT::Vertex>, LowerCost>
  openList;
PriorityQueue<RRT::Vertex, std::deque<RRT::Vertex>, LowerCost>::
  const_iterator pqPtr;
std::vector<RRT::Edge> path;
std::vector<RRT::Edge> pathOptimal;
std::vector<RRT::Edge>::const_iterator pathPtr;

// Methods
private:
double extVertDelta(double min_dist, double thresh);
friend class LowerCost;
};

Listing 22: MotionPlanner Class header

#include "MotionPlanner.h"

// Meyers Singleton
MotionPlanner& MotionPlanner::getInstance() {
  static MotionPlanner instance;
  return instance;
}

// disTh(std::Pi, std::Pi), disU(std::Pi, std::Pi), disV(-std::Pi, std::Pi),
MotionPlanner::MotionPlanner() :
  startState(RRT::DIM), goalState(RRT::DIM),
MotionPlanner::~MotionPlanner() {
  // Clean up
  if (cSpace) {
    delete cSpace;
  }
}
if (closedList) {
    delete [] closedList;
}

for (boost::tie(vi, viend) = boost::vertices(rrt); vi != viend; ++vi) {
    for (boost::tie(ei, eiend) = boost::out_edges(*vi, rrt); ei != eiend;
         ++ei) {
        // check that edge not already deleted (v_id[0] will have edge
        0->3, so at v_id[3], edge already deleted)
        if (e_action[*ei]) {
            delete (e_action[*ei]);
            e_action[*ei] = nullptr;
        }
    }
    delete v_state[*vi];
}

bool MotionPlanner::runRRT(RRTtype type, double maxDistThresh, double
minDistThresh,
        int numBeforeLookForGoal, int vLimit, int
minNodes) {
    auto discrete = [](double d)->double { int i = (int)d; return (double)
i / 100.0; }; 
    RobotState qr(3); // random node
    RobotState qGoalLine(3);
    RobotState vEdge(3);
    bool collided, edgeUsed, usingGoal, goalLineConnected;
    int iterations, indexToUse;
    double min_dist;
double omega;

// create the descriptor
std::stringstream strm; strm.str(std::string()); strm.clear();
if (type == VERTEX)   { strm << "VERTEX";   }
else if (type == EDGE) { strm << "EDGE";     }
else                   { strm << "EXT_VERTEX"; }
strm << "," << maxDistThresh << "," << minDistThresh << ",";
strm << numBeforeLookForGoal << "," << vLimit << "," << minNodes;
descriptor = strm.str();

// Add the start node to the graph
vstart = boost::add_vertex(rrt);
vid[vstart] = 0;
ystate[vstart] = new RobotState(startState);
heur[vstart] = startState.dist(goalState);
cost[vstart] = 0.0;
goalLineConnected = false;

numv = 1;
iterations = missedNodes = 0;
while (numv <= vLimit) {
  ++iterations;
  if (iterations >= 3000000) return false;
  // generate new random node
  if (numv % numBeforeLookForGoal == 1 && !goalIsConnected) {
    if (failedToConnectToGoal) {
      double st[] = { discrete(disTh(gen)), discrete(disU(gen)),
                      discrete(0) };
      //double st[] = { disTh(gen), disU(gen), disV(gen) };
}
qr = RobotState(3, st);
usingGoal = false;
indexToUse = -1;
}
else {
    qr = goalState;
    usingGoal = true;
    indexToUse = 0; // theta = 0, u = 1, v = 2
}
}
else {
    double st[] = { discrete(disTh(gen)), discrete(disU(gen)),
discrete(0) };
    //double st[] = { disTh(gen), disU(gen), disV(gen) };
    qr = RobotState(3, st);
    failedToConnectToGoal = false;
    usingGoal = false;
    indexToUse = -1;
}

// check for qr collisions
collided = cSpace->collidedState(qr);
if (collided) { ++missedNodes; continue; }

// find closest vertex
min_dist = -1;
vptr2 = vstart;
edgeUsed = false;
for (boost::tie(vi, viend) = boost::vertices(rrt); vi != viend; ++vi )
{
    double distanceV = qr.dist(v_state[*vi], indexToUse);
if (distanceV < min_dist || min_dist < 0) {
    min_dist = distanceV;
    vptr2 = *vi;
    edgeUsed = false;
}

if (type == RRTtype::EDGE) {
    std::pair<std::size_t, std::size_t> e1, e2;
    RRT::EdgeMeasured emap;
    RRT::EdgeMeasured::iterator pos;
    // find closest edge
    for (boost::tie(ei, eend) = boost::out_edges(*vi, rrt); ei != eend; ++ei) {
        // grab the edge (defined by the 2 vertex ids)
        e1 = std::make_pair(v_id[boost::source(*ei, rrt)], v_id[boost::target(*ei, rrt)]);
        // grab its "mirror" ( <1.0> is same edge as <0.1> )
        e2 = std::make_pair(e1.second, e1.first);

        // check if the edge has already been used
        if (emap.find(e1) == emap.end()) {
            // check if the "mirror" of the edge has been used
            if (emap.find(e2) == emap.end()) {
                // insert the edge into the map
                boost::tie(pos, inserted) = emap.insert(std::make_pair(e1, RobotState()));
                // calculate the distance (normal intersect)
                pos->second = qr.intersect(v_state[boost::source(*ei, rrt)], v_state[boost::target(*ei, rrt)]);
                double distanceE = qr.dist(pos->second);
                if (pos->second.onTheLine(v_state[boost::source(*ei, rrt)], v_state[boost::target(*ei, rrt)]))
if (distanceE < min_dist || min_dist < 0) {
    min_dist = distanceE;
    edgeUsed = true;
    vEdge = pos->second;
    eptr2 = *ei;
}
}

// means the mirror was found (edge used)
} // means the edge was found (edge used)

} // end looping for edges for node(i)

/* RRTtype == EDGE */

} // end find closest distance for all nodes

// check if the new edge will pass through an obstacle (between * v_state[vptr2] and qr)
if (type == RRTtype::EDGE && edgeUsed) {
    collided = cSpace->collidedEdge(vEdge, qr);
}
else {
    // modify the desired node to the one that is one the goalLine
    if (usingGoal && min_dist >= 0.005 && !goalLineConnected) {
        qGoalLine = RobotState(*v_state[vptr2]);
        goalState.get(0, omega);
        qGoalLine.set(0, omega);
        usingGoal = false;
        goalLineConnected = true;
    }
}
qr = qGoalLine;

}
collided = cSpace->collidedEdge(*(v.state[vptr2]), qr);
}

if (collided) {
    if (qr == goalState) {
        // set flag to know try to select more random nodes
        // otherwise will continuously try (and fail) to connect to the
        // goal node
        failedToConnectToGoal = true;
    }
    ++missedNodes;
    continue;
}

if (usingGoal && (goalLineConnected || min_dist <= 0.001)) {
    min_dist = goalState.dist(qGoalLine, -1);
}

// add qr to the graph if greater than minimum threshold distance
// if qr is the goal node, we don't care if it's "too" close
// this accounts for the case where we randomly picked a node that
// is basically the goal location
if (min_dist <= minDistThresh && !(qr == goalState)) { ++missedNodes
    ; continue; }

// make sure that if using edges, the new edges aren't less than the
// minimum distance
if (type == RRTtype::EDGE && edgeUsed && !(qr == goalState)) {
    src = boost::source(eptr2, rrt);
    tgt = boost::target(eptr2, rrt);
if ((v_state[src]->dist(vEdge) <= minDistThresh ||
    v_state[tgt]->dist(vEdge) <= minDistThresh))
{
    ++missedNodes; continue;
}

// if we aren’t using edges but the new distance is “far”, let’s
break it up into multiple nodes/edges
// this can be faster for checking “closest” node, rather than using
edges (at higher resource cost)
if (type == RRTtype::EXT_VERTEX && min_dist >= maxDistThresh) {
    // Try to keep new distances less than 1ft (30.5 cm)
    RobotState q = qr - (*v_state[vptr2]);
    RobotState n = q / q.magnitude();
    double delta = extVertDelta(min_dist, maxDistThresh);
    double d = 0;
    while (d < (min_dist - delta)) {
        vptr = boost::add_vertex(rrt);
        v_id[vptr] = numv++;
        v_state[vptr] = new RobotState((*v_state[vptr2]) + n*delta);
        v_heur[vptr] = v_state[vptr]->dist(goalState);
        boost::tie(eptr, inserted) = boost::add_edge(vptr2, vptr, rrt);
        if (inserted) {
            e_dist[eptr] = delta;
            e_action[eptr] = new RobotAction(v_state[vptr2], v_state[vptr]);
        }
        vptr2 = vptr;
        d += delta;
    }
    min_dist = delta;
if (qr == goalState) {
    // at this point, the goal can now be connected
    // the loop will most likely still run, since having more nodes in
    // the graph
    // helps plan paths
    goalIsConnected = true;
    goalId = numv;
}

// add new node to the tree
vptr = boost::add_vertex(rrt);
vid[vptr] = numv;
state[vptr] = new RobotState(qr);
heur[vptr] = qr.dist(goalState);

// check if edge or vertex used
if (type == RRTtype::EDGE && edgeUsed) {
    ++numv;
    vptr2 = boost::add_vertex(rrt);
    vid[vptr2] = numv;
    state[vptr2] = new RobotState(vEdge);
    heur[vptr2] = vEdge.dist(goalState);
    // grab vertex end points
    src = boost::source(eptr2, rrt);
    tgt = boost::target(eptr2, rrt);
    // remove old edge (and delete the action)
    delete e_action[eptr2];
    e_action[eptr2] = nullptr;
    boost::remove_edge(eptr2, rrt);
    // add new edges
boost::tie(eptr, inserted) = boost::add_edge(src, vptr2, rrt);
    if (inserted) {
        e_dist[eptr] = v_state[src]->dist(v_state[vptr2]);
        e_action[eptr] = new RobotAction(v_state[src], v_state[vptr2]);
    }
boost::tie(eptr, inserted) = boost::add_edge(vptr2, tgt, rrt);
    if (inserted) {
        e_dist[eptr] = v_state[vptr2]->dist(v_state[tgt]);
        e_action[eptr] = new RobotAction(v_state[vptr2], v_state[tgt]);
    }
boost::tie(eptr, inserted) = boost::add_edge(vptr2, vptr, rrt);
    if (inserted) {
        e_dist[eptr] = min_dist;
        e_action[eptr] = new RobotAction(v_state[vptr2], v_state[vptr]);
    }
    // increment vLimit, since we want to add 10 new nodes, not
    // including ones I added into edges
    ++vLimit;
}
else {
    boost::tie(eptr, inserted) = boost::add_edge(vptr2, vptr, rrt);
    if (inserted) {
        e_dist[eptr] = min_dist;
        e_action[eptr] = new RobotAction(v_state[vptr2], v_state[vptr]);
    }
    else {
        std::cout << "error" << std::endl;
    }
}
++numv;
if (goalsIsConnected && numv >= minNodes) {
break;
    } /* quit the loop */
    }
    return goalsIsConnected;
    }

void MotionPlanner::resetGraph() {
    goalId = -1;
    goalsIsConnected = false;
    failedToConnectToGoal = false;
    // Delete all nodes and edges (along with their properties)
    int i = numv−1;
    //while (i >= 0) {
    // for (boost::tie(ei, eiend) = boost::out_edges(boost::vertex(i, rrt), rrt); ei != eiend; ++ei) {
    //    if (e_action[*ei]) {
    //        delete (e_action[*ei]);
    //    e_action[*ei] = nullptr;
    //    }
    // }
    // delete v_state[boost::vertex(i, rrt)];
    // v_state[boost::vertex(i, rrt)] = nullptr;
    // // boost::clear_vertex(boost::vertex(i, rrt), rrt);
    // // boost::remove_vertex(boost::vertex(i, rrt), rrt);
    // −−i;
    //}
    rrt.clear();
    
    std::vector<RobotAction> MotionPlanner::runAstar() {
        std::vector<RobotAction> actions;
pathOptimal.clear();
path.clear();
path.reserve(numv);

if (goalsIsConnected) {
    // initialize the closedList to empty
    if (!closedList) { closedList = new bool[numv]; }
    memset(closedList, 0, numv);

    // initialize the openList with the start node
    openList.push(vstart);

    bool keepPlanning = true;
    while (keepPlanning) {
        vptr = openList.top();
        openList.pop();
        // add node to the closed list
        closedList[v.id[vptr]] = true;
        // for all neighbors
        for (boost::tie(ei, eiend) = boost::out_edges(vptr, rrt); ei !=
eiend; ++ei) {
            tgt = boost::target(*ei, rrt);
            std::size_t smacky = v.id[tgt];
            // check if neighbor (tgt) is in the closed list
            if (!closedList[v.id[tgt]]) {
                // find cost of the neighbor
                // \( f(n') = (g(n) + g(n')) + h(n') \)
                double cost = (v.cost[vptr] + v.state[vptr]->dist(v.state[tgt]
                              )) + v.heur[tgt];

                // check if tgt is currently in the openList
                pqPtr = openList.find(tgt);

                // add tgt to the openList
                if (pqPtr == openList.end()) {
                    pqPtr = openList.push(tgt);
                    pqPtr->second = cost;
                    pqPtr->first = tgt;
                } else {  // tgt is already in the openList
                    if (pqPtr->second < cost) continue;
                    pqPtr->second = cost;
                    pqPtr->first = tgt;
                }
            }
        }
    }
}

// initialize the openList with the start node
openList.push(vstart);

bool keepPlanning = true;
while (keepPlanning) {
    vptr = openList.top();
    openList.pop();
    // add node to the closed list
    closedList[v.id[vptr]] = true;
    // for all neighbors
    for (boost::tie(ei, eiend) = boost::out_edges(vptr, rrt); ei !=
eiend; ++ei) {
        tgt = boost::target(*ei, rrt);
        std::size_t smacky = v.id[tgt];
        // check if neighbor (tgt) is in the closed list
        if (!closedList[v.id[tgt]]) {
            // find cost of the neighbor
            // \( f(n') = (g(n) + g(n')) + h(n') \)
            double cost = (v.cost[vptr] + v.state[vptr]->dist(v.state[tgt]
                              )) + v.heur[tgt];

            // check if tgt is currently in the openList
            pqPtr = openList.find(tgt);

            // add tgt to the openList
            if (pqPtr == openList.end()) {
                pqPtr = openList.push(tgt);
                pqPtr->second = cost;
                pqPtr->first = tgt;
            } else {  // tgt is already in the openList
                if (pqPtr->second < cost) continue;
                pqPtr->second = cost;
                pqPtr->first = tgt;
            }
        }
    }
}

// initialize the openList with the start node
openList.push(vstart);

bool keepPlanning = true;
while (keepPlanning) {
    vptr = openList.top();
    openList.pop();
    // add node to the closed list
    closedList[v.id[vptr]] = true;
    // for all neighbors
    for (boost::tie(ei, eiend) = boost::out_edges(vptr, rrt); ei !=
eiend; ++ei) {
        tgt = boost::target(*ei, rrt);
        std::size_t smacky = v.id[tgt];
        // check if neighbor (tgt) is in the closed list
        if (!closedList[v.id[tgt]]) {
            // find cost of the neighbor
            // \( f(n') = (g(n) + g(n')) + h(n') \)
            double cost = (v.cost[vptr] + v.state[vptr]->dist(v.state[tgt]
                              )) + v.heur[tgt];

            // check if tgt is currently in the openList
            pqPtr = openList.find(tgt);

            // add tgt to the openList
            if (pqPtr == openList.end()) {
                pqPtr = openList.push(tgt);
                pqPtr->second = cost;
                pqPtr->first = tgt;
            } else {  // tgt is already in the openList
                if (pqPtr->second < cost) continue;
                pqPtr->second = cost;
                pqPtr->first = tgt;
            }
        }
    }
}
if (pqPtr == openList.end()) {
    v_cost[tgt] = cost;
    openList.push(tgt);
    // add edge to the path
    path.push_back(*ei);
}
else if (cost < v_cost[*pqPtr]) {
    // delete the old edge in the path, since we found a better one
    RRT::E_iter ei2, eiend2;
    for (boost::tie(ei2, eiend2) = boost::out_edges(*pqPtr, rrt); ei2 != eiend2; ++ei2) {
        pathPtr = std::find(path.begin(), path.end(), *ei2);
        if (pathPtr != path.end()) {
            path.erase(pathPtr);
            break;
        }
    }
    // update the cost & add new edge to the path
    v_cost[*pqPtr] = cost;
    path.push_back(*ei);
}
}
RobotState q = *v_state[openList.top()];
int id = v_id[openList.top()];
keepPlanning = !(v_state[openList.top()] == goalState);

// Grab optimal path from the
int pathLen = 0;
```cpp
std::size_t srcId, tgtId;

for (std::size_t i = path.size() - 1; i >= 0 && i < path.size(); --i) {
    srcId = (pathLen == 0) ? goalId : v_id[boost::source(pathOptimal[pathLen - 1], rrt)];
    tgtId = v_id[boost::target(path[i], rrt)];
    if (srcId == tgtId) {
        pathOptimal.push_back(path[i]);
        actions.push_back(*e_action[path[i]]);
        ++pathLen;
    }
}

openList.clear();
// while (!openList.empty())
// openList.pop();
return actions;
}

void MotionPlanner::optimizeActions(const std::vector<RobotAction>& actions) {
    auto min_angle = [](double delta) -> double {
        if ((delta) > std::Pi) return (delta - (2 * std::Pi));
        if ((delta) < -std::Pi) return ((delta) + (2 * std::Pi));
        return (delta);
    };
    // reverse the path
    RobotAction tmp;
    std::size_t i = 0;
    std::size_t j = actions.size() - 1;
    while (i <= j) {
        // code
    }
```
tmp = actions[i];
actions[i] = actions[j];
actions[j] = tmp;
if (j == 0) {
    break;
}
++i;
--j;
}

// optimize by removing the second rotation action
// assumptions: start node is first, goal node is last
/** for (i = 0; i < actions.size() - 1; ++i) {
    delta = actions[i].getRotation2() + actions[i + 1].getRotation1();
    if (fabs(delta) <= epsilon) {
        delta = 0.0;
    }
    actions[i].setRotation2(min_angle(delta));
    actions[i + 1].setRotation1(0.0);
} */

/* Config File Format
   (1) seed
   (2) dim
   (3) c_space.conf
   (4) start
   (5) goal_cnt (2)
   (6) goal_1
   (7) goal_2 */
*/

void MotionPlanner::configure(std::string configfile) {
    FILE *fptr;
    char buf[256];
    char cspaceFname[80];
    int seed;
    int cspace_dim;
    double st[3];
    int goal_cnt;

    fopen_s(&fptr, configfile.c_str(), "r");
    if (fptr == NULL) {
        std::cout << "Error! couldn't open file " << configfile << std::endl;
        return;
    }

    // seed
    fgets(buf, 256, fptr);
    sscanf_s(buf, "%d", &seed);

    // dimension of cspace
    fgets(buf, 256, fptr);
    sscanf_s(buf, "%d", &cspace_dim);

    // cspace file name
    fgets(buf, 256, fptr);
    sscanf_s(buf, "%s", cspaceFname, 80);

    // start state
    fgets(buf, 256, fptr);
    sscanf_s(buf, "%lf %lf %lf", st + 0, st + 1, st + 2);
    startState = RobotState(3, st);

    // number of goals
    fgets(buf, 256, fptr);
sscanf_s(buf, "%d", &goal_cnt);

// goal state: 1
fgets(buf, 256, fptr);
sscanf_s(buf, "%lf %lf %lf", st + 0, st + 1, st + 2);
goalState = RobotState(3, st);
// goal state: 2
// fgets(buf, 256, fptr);
// sscanf_s(buf, "%lf %lf %lf", st + 0, st + 1, st + 2);
// goalState2 = RobotState(dim, st);
fclose(fptr);

// Set up the random number generator (re-seed it)
if (seed <= -1) { gen.seed(rd()); } else { gen.seed(seed); }

if (cSpace) {
    delete cSpace;
    cSpace = nullptr;
}
cSpace = new ConfigSpace(std::string(cspaceFname), cspace_dim);
configured = true;

std::string MotionPlanner::buildsv() {
    std::stringstream csvfile;
csvfile.str(std::string());
csvfile.clear();
double x[3];
    // Output the tree
csvfile << "Id, state.th, state.u, state.v, e_dist, e_id," << std::
for (boost::tie(vi, viend) = boost::vertices(rrt); vi != viend; ++vi)
{
    v_state[*vi]->get(0, x[0]);
    v_state[*vi]->get(1, x[1]);
    v_state[*vi]->get(2, x[2]);
    csvfile << v_id[*vi] << "", " << x[0] << "", " << x[1] << "", " << x[2] << "", ";
    for (boost::tie(ei, eiend) = boost::out_edges(*vi, rrt); ei != eiend; ++ei) {
        csvfile << e_dist[*ei] << "", " << v_id[boost::target(*ei, rrt)] << "", ";
    }
    csvfile << std::endl;
}

// Output the path (in nodes)
for (std::size_t i = pathOptimal.size() - 1; i >= 0 && i < pathOptimal.size(); --i) {
    csvfile << v_id[boost::source(pathOptimal[i], rrt)] << "", ";
}
if (pathOptimal.size()) {
    csvfile << v_id[boost::target(pathOptimal[0], rrt)] << std::endl;
}

// Output the path (in actions)
// TODO

return csvfile.str();

// Private methods
double MotionPlanner::extVertDelta(double min_dist, double thresh) {

double delta;
if (min_dist >= 5 * thresh) { delta = min_dist / 6; }
else if (min_dist >= 4 * thresh) { delta = min_dist / 5; }
else if (min_dist >= 3 * thresh) { delta = min_dist / 4; }
else if (min_dist >= 2 * thresh) { delta = min_dist / 3; }
else { delta = min_dist / 2; }
return delta;

Listing 23: MotionPlanner Class source

#ifndef _PRIORITY_QUEUE_H__
define _PRIORITY_QUEUE_H__

#include <queue>

template<
class T,
class Container = std::vector<T>,
class Compare = std::less<typename Container::value_type>
> class PriorityQueue : public std::priority_queue<T, Container, Compare>
{
public:

typedef typename

std::priority_queue<

T,

Container,

Compare>::container_type::const_iterator const_iterator;

const_iterator find(const T&val) const {
    auto first = this->c.begin();
    auto last = this->c.end();

    208
```
while (first != last) {
    if (*first == val) return first;
    ++first;
}
return last;
}
const_iterator end() const {
    return this->c.end();
}
void clear() {
    while (!this->empty()) {
        this->pop();
    }
};
#endif /*_PRIORITY_QUEUE_H__*/

Listing 24: PriorityQueue Class header

#ifndef __ROBOT_ACTION_H__
#define __ROBOT_ACTION_H__

#include <random>
#include "RobotState.h"
#include <algorithm>

class RobotAction {
public:
    RobotAction();
    RobotAction(int d);
    RobotAction(int d, double *act);
```
RobotAction(const RobotAction& act);
RobotAction(const RobotState* s1, const RobotState* s2);
~RobotAction();

// Operator Overloads
RobotAction& operator=(const RobotAction& act)
{
    if (dim == act.dim) {
        memcpy(action, act.action, dim * sizeof(double));
    }
    return (*this);
}

int get(int idx, double &val) const;
int set(int idx, double val);

private:
    int dim;
    double *action;
};

Listing 25: RobotAction Class header
```cpp
RobotAction::RobotAction(int d) : dim(d), action(new double[dim]) {
    std::memset(action, 0, dim * sizeof(double));
}

RobotAction::RobotAction(int d, double *act) : dim(d), action(new double[dim]) {
    std::memcpy(action, act, dim * sizeof(double));
}

RobotAction::RobotAction(const RobotAction& act) : dim(act.dim), action(new double[dim]) {
    std::memcpy(action, act.action, dim * sizeof(double));
}

RobotAction::RobotAction(const RobotState* s1, const RobotState* s2) :
    RobotAction(s1->dimension()) {
    const double epsilon = 0.0001;
    double src, dst;
    auto min_angle = [] (double dst, double src) -> double {
        if ((dst - src) > std::Pi) return ((dst - src) - (2 * std::Pi));
        if ((dst - src) < -std::Pi) return ((dst - src) + (2 * std::Pi));
        if ((dst - src) == std::Pi || (dst - src) == -std::Pi) return (0);
        return (dst - src);
    };
    for (int i = 0; i < dim; ++i) {
        (void)s1->get(i, src);
        (void)s2->get(i, dst);
        action[i] = min_angle(dst, src);
    }
```
Listing 26: RobotAction Class source

```cpp
// unused (so restrict)
RobotCom() = delete;
RobotCom(const RobotCom&) = delete;
RobotCom& operator=(const RobotCom&) = delete;
```
bool isConnected() const { return connected; }
int getPortNumber() const { return portNo; }
std::string getIpAddr() const { return ipAddr; }

bool connectToRobot();
void closeConnection();
int sendMsg(const std::string msg);
int recvMsg(std::string& msg);

private:
    SOCKET _socket;
    bool connected;
    int portNo;
    std::string ipAddr;
    const int msgLen;
    char* buffer;
};

Listing 27: RobotCom Class header

#include "RobotCom.h"

bool RobotCom::connectToRobot() {
    // start up Winsock...
    WSADATA wsaData;

    int error = WSAStartup(0x0202, &wsaData);
    if (error) {
        // handle error
    } else {
        // proceed with connection
    }
}
connected = false;
return false;
}

// verify version
if (wsadata.wVersion != 0x0202) {
    WSACleanup();
    connected = false;
    return false;
}

SOCKADDR_IN target;
target.sin_family = AF_INET;
target.sin_port = htons(portNo);
target.sin_addr.s_addr = inet_addr(ipAddr.c_str());

_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); // create socket
if (_socket == INVALID_SOCKET) {
    connected = false;
    return false;
}

// try connecting...
connected = !(connect(_socket, (SOCKADDR*)&target, sizeof(target)) ==
    SOCKET_ERROR);
return connected;

void RobotCom::closeConnection() {
    if (_socket) {
        closesocket(_socket);
    }
}
```cpp
WSACleanup();

int RobotCom::sendMsg(const std::string msg) {
    int bytes = -1;
    strcpy(buffer, msgLen, msg.c_str());
    if (connected) {
        bytes = send(_socket, buffer, strlen(buffer), 0);
        if (bytes == SOCKET_ERROR)
            return WSAGetLastError();
    }
    return bytes;
}

int RobotCom::recvMsg(std::string& msg) {
    int bytes = recv(_socket, buffer, msgLen, 0);
    if (bytes < 0)
        return WSAGetLastError();

    // return by reference
    msg = (buffer);
    return bytes;
}
```

Listing 28: RobotCom Class source

```cpp
#include <iostream>

class RobotState {
```
public:
    RobotState();
    RobotState(int d);
    RobotState(int d, double *st);
    RobotState(const RobotState& st);
    ~RobotState();

    // Operator Overloads
    RobotState& operator=(const RobotState& st)
    {
        if (dim == st.dim)
        {
            memcpy(state, st.state, dim * sizeof(double));
        }
        return (*this);
    }

    RobotState operator-(const RobotState& st) const;
    RobotState operator+(const RobotState& st) const;
    RobotState operator*(double a) const;
    RobotState operator/(double a) const;
    bool operator==(const RobotState& st) const;

    // Get/Set functions
    int get(int idx, double &val) const;
    int set(int idx, double val);
    int dimension() const { return dim; }

    // Other operations
    double dot(const RobotState& st) const;
    double magnitude() const;

// distance function
RobotState intersect(const RobotState* st1, const RobotState* st2)
const;

bool onTheLine(const RobotState* st1, const RobotState* st2) const;

// distance function
double dist(const RobotState* st, int indexToUse = −1) const;
double dist(const RobotState st, int indexToUse = −1) const;
double euclidDist(const RobotState st) const;

private:
const double _pi = 3.1415926535897;
int dim;
double *state;
double min_angle(double dst, double src) const;

// implementation specific
int weights(double *a) const;
}

Listing 29: RobotState Class header

#include "RobotState.h"
#include <algorithm>

// Constructors
RobotState::RobotState() : dim(2), state(new double[dim])
{
    std::memset(state, 0, dim * sizeof(double));
}
RobotState::RobotState(int d) : dim(d), state(new double[dim])
RoboState::RoboState(int d, double *st) : dim(d), state(new double[dim])
{
    std::memset(state, 0, dim * sizeof(double));
}

RoboState::RoboState(const RoboState& st) : dim(st.dim), state(new double[dim])
{
    std::memcpy(state, st.state, dim * sizeof(double));
}

// Destructor
RoboState::~RoboState()
{
    if (state != nullptr) {
        delete[] state;
    }
}

int RoboState::get(int idx, double &val) const
{
    if (idx < 0 || idx >= dim) {
        return -1;
    }
    val = state[idx];
    return 0;
}
```cpp
int RobotState::set(int idx, double val)
{
    if (idx < 0 || idx >= dim) {
        return -1;
    }
    state[idx] = val;
    return 0;
}

// Operator Overloads

bool RobotState::operator==(const RobotState& st) const
{
    if (dim != st.dim) {
        return false;
    }
    for (int i = 0; i < dim; ++i) {
        if (state[i] != st.state[i]) {
            return false;
        }
    }
    return true;
}

RobotState RobotState::operator-(const RobotState& st) const
{
    RobotState rs(dim);
    for (int i = 0; i < dim; ++i) {
        rs.state[i] = this->state[i] - st.state[i];
    }
    return rs;
}
```
RobotState RobotState::operator+(const RobotState& st) const
{
    RobotState rs(dim);
    for (int i = 0; i < dim; ++i) {
        rs.state[i] = this->state[i] + st.state[i];
    }
    return rs;
}

RobotState RobotState::operator*(double a) const
{
    RobotState rs(dim);
    for (int i = 0; i < dim; ++i) {
        rs.state[i] = this->state[i] * a;
    }
    return rs;
}

RobotState RobotState::operator/(double a) const
{
    RobotState rs(dim);
    for (int i = 0; i < dim; ++i) {
        rs.state[i] = this->state[i] / a;
    }
    return rs;
}

// Other operations
double RobotState::dot(const RobotState& st) const
{
    double result = 0.0;
    for (int i = 0; i < dim; ++i) {
        result += this->state[i] * st.state[i];
    }
    return result;
}
```cpp
double RobotState::magnitude() const {
    double result = 0.0;
    for (int i = 0; i < dim; ++i) {
        result += this->state[i] * this->state[i];
    }
    return std::sqrt(result);
}

// return the the node that is the projection onto the line segment defined by 2 node parameters
RobotState RobotState::intersect(const RobotState* st1, const RobotState* st2) const {
    return RobotState(dim);
}

bool RobotState::onTheLine(const RobotState* st1, const RobotState* st2) const {
    return false;
}

// distance function
double RobotState::dist(const RobotState* st, int indexToUse) const {
    auto SQR = [](double x)->double { return x*x; }; 
    // implementation specific
```
double *a = new double[dim];
double tmp;

// set the values of a (implementation specific)
(void)weights(a);
double d = 0;
if (indexToUse >= 0) {
    (void)st->get(indexToUse, tmp);
    d = a[indexToUse] * SQR(state[indexToUse] - tmp);
}
else {
    for (int i = 0; i < dim; ++i) {
        (void)st->get(i, tmp);
        d += a[i] * SQR(state[i] - tmp);
    }
}
delete[] a;
return std::sqrt(d);

double RobotState::dist(const RobotState st, int indexToUse) const
{
    auto SQR = [](double x) -> double { return x*x; };
    // implementation specific
double *a = new double[dim];
double tmp;
    // set the values of a (implementation specific)
    (void)weights(a);
double d = 0;
    if (indexToUse >= 0) {
        (void)st->get(indexToUse, tmp);
        d = a[indexToUse] * SQR(state[indexToUse] - tmp);
    }
```
} else {
    for (int i = 0; i < dim; ++i) {
        (void)st.get(i, tmp);
        d += a[i] * SQR(state[i] - tmp);
    }
}

delete[] a;
return std::sqrt(d);
}

double RobotState::euclidDist(const RobotState &st) const
{
    auto SQR = [](double x)->double { return x*x; };
    double tmp, d;
    d = 0;
    for (int i = 0; i < dim; ++i) {
        (void)st.get(i, tmp);
        d += SQR(state[i] - tmp);
    }
    return std::sqrt(d);
}

double RobotState::min_angle(double dst, double src) const
{
    if ((dst - src) > -pi) return ((dst - src) - (2 * -pi));
    if ((dst - src) < -pi) return ((dst - src) + (2 * -pi));
    return (dst - src);
}

int RobotState::weights(double *a) const
{

}
```
// dim = 3;

a[0] = 0.3;
a[1] = 0.7;
a[2] = 0.0;

return 1;

Listing 30: RobotState Class source
Appendix E  Continuum Element Simulation Software

```matlab
function [L] = all_lengths( X )

d = 0.0438;
alpha = d*sqrt(3)/2;
A = [ 0, -2, 2/(d*sqrt(3)), 0;
     1, 1, 2/(d*sqrt(3)), 0;
    -1, 1, 2/(d*sqrt(3)), 0;
     0, 0, 0, 1 ];
inv(A)
l = (alpha*A*X);
l1 = l(1); l2 = l(2); l3 = l(3);
end
```

Listing 31: Function to retrieve tendon lengths from configuration variables

```matlab
function col = check_collision(arm, off, gripper, all_obs)

col = false;
X = 1; Y = 2; Z = 3;
rad = 0.025;
[r,~] = size(arm);
c = max(size(all_obs));

% for each obstacle
for j = 1:c
    obs = all_obs{j};
    % check gripper
    for g = 1:3
        grp = gripper{g};
        col = is_inside(grp.p, obs);
        if (col)
            return;
        end
    end
end
```

225
end
col = is.inside(obs.p, grp);
if (col)
    return;
end

% check along the backbone, so inflate by backbone radius
obs.w = obs.w + 2*rad;
obs.l = obs.l + 2*rad;
obs.h = obs.h + 2*rad;
col = is.inside(arm + [zeros(r, 1), zeros(r, 1), off*ones(r, 1)], obs);
if (col)
    return;
end
end

Listing 32: Function to check for collisions

function execute_timestep_ind(obj, ~)
S = 1; U = 2; V = 3; W = 4;
data = get(obj, 'UserData');
data.t = data.t + data.tau;
data.t;
% data has the following properties
% t:
% tau:
% state: [s, u, v, w]
% setpt: [s, u, v, w]
% K: [kp, kd, ki]
% ravg: [ras, rau, rav, raw]
samples:

elast: [es, eu, ev, ew]

s_0 = 0.95; d = 0.022;

u = data.state(U); v = data.state(V); s = data.state(S);

l = [ s_0 + (-2)*d*v ;
      s_0 + ( 2)*d*u ;
      s_0 + ( 2)*d*v ;
      s_0 + (-2)*d*u ];

u = data.setpt(U); v = data.setpt(V); s = data.setpt(S);

l_set = [ s_0 + (-2)*d*v ;
          s_0 + ( 2)*d*u ;
          s_0 + ( 2)*d*v ;
          s_0 + (-2)*d*u ];

out = zeros(1,4);

err = l_set - 1;

for i = 1:4
    data.ravg(i) = (data.ravg(i)*data.samples+err(i))/(data.samples+1);
    out(i) = err(i) * data.K(1) + ... 
              (err(i) - data.elast(i))/data.t * data.K(2) + ... 
              (data.ravg(i)) * data.K(3);

    if (out(i) > 12)
        out(i) = 12;
    elseif (out(i) < -12)
        out(i) = -12;
    end
end

% simulate
The code snippet is a function that executes a time-step in a simulation. It includes calculations and operations that simulate a process involving variables such as `W`, `U`, `V`, `v`, and `s`. The snippet demonstrates how to handle states and data samples within a simulation framework. The function iterates through a grid defined by `W`, `U`, and `V`, simulating collisions, and saves the relevant data points. The listing also includes clear statements and initializations to set up the simulation environment.
Listing 34: Script that explores the space to generate the occupancy map

clear; clc;

% load('cspace_shelf.grp.mat');
load('cspace_shelf.grp2.mat');
cspace = uint8(zeros(629,629,629));
[len,~] = size(points);

for i=1:len
    r = int32((points(i,1) + pi) * 100)+1;
    c = int32((points(i,2) + pi) * 100)+1;
    d = int32((points(i,3) + pi) * 0) + 1;  \% v should always be 0
    cspace(r,c,d) = 1;
end
Listing 35: Script that generates the occupancy map file

```matlab
function b = is_inside(points, obj)
    xyz = [false false false];
    X = 1; Y = 2; Z = 3;
    upperb = zeros(3);
    lowerb = zeros(3);
    
    for i = X:Z
        upperb(i) = max(obj.p(:,i));
        lowerb(i) = min(obj.p(:,i));
    end
    
    [r,~] = size(points);
    for i = 1:r
        for j = X:Z
            xyz(j) = (points(i,j) >= lowerb(j) && points(i,j) <= upperb(j));
        end
        b = xyz(X) && xyz(Y) && xyz(Z);
```
Listing 36: Function that checks if a point is inside a polygon

```matlab
function varargout = KinematicGui(varargin)

% KINEMATICGUI MATLAB code for KinematicGui.fig
% KINEMATICGUI, by itself, creates a new KINEMATICGUI or raises the existing
% singleton*.
%
% H = KINEMATICGUI returns the handle to a new KINEMATICGUI or the handle to
% the existing singleton*.
%
% KINEMATICGUI('CALLBACK', hObject, eventData, handles, ...) calls the local
% function named CALLBACK in KINEMATICGUI.M with the given input arguments.
%
% KINEMATICGUI('Property', 'Value', ...) creates a new KINEMATICGUI or raises the
% existing singleton*. Starting from the left, property value pairs are
% applied to the GUI before KinematicGui_OpeningFcn gets called. An
% unrecognized property name or invalid value makes property
```

231
application

% stop. All inputs are passed to KinematicGui_OpeningFcn via varargin.

% *See GUI Options on GUIDE’s Tools menu. Choose “GUI allows only one instance to run (singleton)”.

% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help KinematicGui

% Last Modified by GUIDE v2.5 13–Feb–2019 13:58:08

% Begin initialization code – DO NOT EDIT

gui_Singleton = 1;

gui_State = struct ('gui_Name', mfilename, ...
    'gui_Singleton', gui_Singleton, ...
    'gui_OpeningFcn', @KinematicGui_OpeningFcn, ...
    'gui_OutputFcn', @KinematicGui_OutputFcn, ...
    'gui_LayoutFcn', [], ...
    'gui_Callback', []);

if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargs
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code – DO NOT EDIT

% —— Executes just before KinematicGui is made visible.

function KinematicGui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved – to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to KinematicGui (see VARARGIN)

% Choose default command line output for KinematicGui
handles.output = hObject;
handles.S = 1;
handles.U = 2;
handles.V = 3;
handles.W = 4;
handles.X = 6;
handles.Y = 7;
handles.Z = 8;

handles.UV = 3;
handles.UW = 2;
handles.VW = 1;

handles.state = [1.03, 0.0001, 0.0001, 0];
handles.rigid = 0;

handles.delta = 2 * (pi / 180);
handles.max_angle = 4*pi;
handles.min_angle = -4*pi;
handles.cpoints = zeros(1,4);
handles.collided = false;

handles.sim = false;
handles.node = false;

% Update handles structure
guidata(hObject, handles);

set(handles.axesLamp, 'Units', 'pixels', 'Position',
     [-1275, -1250, 3000, 3000]);
set(handles.axesCspace3, 'Units', 'pixels', 'Position', [-850, -1250, 3000, 3000]);
set(handles.axesCspace, 'Units', 'pixels', 'Position', [-475, -1250, 3000, 3000]);

% decent values....

set(handles.axesLamp, 'Units', 'pixels', 'Position',
     [-1150, -1200, 3000, 3000]);
set(handles.axesCspace, 'Units', 'pixels', 'Position', [-450, -950, 3000, 3000]);
set(handles.axesCspace3, 'Units', 'pixels', 'Position', [-450, -1300, 3000, 3000]);

axes(handles.axesLamp);
rotate3d(handles.axesLamp, 'on');

axes(handles.axesCspace);

axes(handles.axesCspace3);

rotate3d(handles.axesCspace3, 'on');
view (handles.axesCspace,0,−90)
view (handles.axesCspace3,30,22)
view (handles.axesLamp,−46,22)

if (handles.rigid == 0)
    Lamp_Sim_GUI(1.03, 0.0001, 0.0001, 0, handles);
else
    rigidLamp_Sim_GUI(1.03, 0.0001, 0.0001, 0, handles);
end

handles.configTable.Data(handles.S) = {1.03};
handles.configTable.Data(handles.U) = {0.0001};
handles.configTable.Data(handles.V) = {0.0001};
handles.configTable.Data(handles.W) = {0.0000};

updatePlot(handles);

% set (handles.rotThVal,'string',0.0000)
% set (handles.rotUVal,'string',0.0001)
% set (handles.rotVVal,'string',0.0001)

% UIWAIT makes KinematicGui wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% Outputs from this function are returned to the command line.
function varargout = KinematicGui_OutputFcn(hObject, eventdata, handles)
% varargout cell array for returning output args (see VARARGOUT);
% hObject handle to figure
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

% —— Executes on button press in viewPoint.
function viewPoint_Callback(hObject, eventdata, handles)
% hObject handle to viewPoint (see GCBO)
% eventdata reserved – to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
v_en = get(handles.viewPoint,'Value');

[az, el] = view;
if v_en == 1
    omega = handles.configTable.Data{handles.W};
    view_az = 180*(omega)/(pi)-30;
    view(handles.axesLamp,view_az,30);
    % rotate3d off;
else
    rotate3d(handles.axesLamp);
end

% Hint: get(hObject,'Value') returns toggle state of viewPoint

function assignValuesAllen(handles)
s = handles.configTable.Data{handles.S};
u = handles.configTable.Data{handles.U};
v = handles.configTable.Data{handles.V};
w = handles.configTable.Data{handles.W};
cla(handles.axesLamp)
```matlab
% handles.collided = Lamp_Sim_GUI(s, u, v, w, handles);
if (handles.rigid == 0)
    handles.collided = Lamp_Sim_GUI(s, u, v, w, handles);
else
    handles.collided = rigidLamp_Sim_GUI(s, u, v, w, handles);
end
handles.collided = false;
updatePlot(handles);

function updatePlot(handles)
    u = handles.configTable.Data{handles,U};
    v = handles.configTable.Data{handles,V};
    w = handles.configTable.Data{handles,W};
    i = getDrawPlane(handles);

    % 3D Plot
    cla(handles.axesCspace3)
    axis(handles.axesCspace3, 18*[−1, 1, −1, 1, −1, 1, 'square']) % set axis to square
    plot_torus(u, v, w, handles.axesCspace3);
    plot_donut(handles.axesCspace3, u, v, w, i);
    axis(handles.axesCspace3, 18*[−1, 1, −1, 1, −1, 1, 'square']) % set axis to square
    set(handles.axesCspace3, 'XTick', [], 'YTick', [], 'ZTick', []);

    % 2D Plot
    fig = handles.axesCspace;
    if (~handles.sim)
        cla(fig)
    end
```
hold(fig, 'on')
grid(fig, 'on')
axis(fig,1.75*[−pi −pi −pi −pi], 'square')
if (i == handles.UW)
    axis(fig,1.75*[−pi −pi −pi −pi], 'square')
    if (handles.sim)
        if (handles.node)
            plot3(fig, w, u, v, 'ko', 'MarkerSize', 3, 'MarkerFaceColor', [1.0, 1.0, 0.0]);
        else
            plot3(fig, w, u, v, 'ko', 'MarkerSize', 1, 'MarkerFaceColor', [0.0, 0.0, 0.0]);
        end
    else
        plot3(fig, w, u, v, 'ko', 'MarkerSize', 3, 'MarkerFaceColor', [1.0, 1.0, 0.0]);
    end
elseif (i == handles.UV)
    axis(fig,1.75*[−pi −pi −pi −pi], 'square')
    if (handles.sim)
        if (handles.node)
            plot3(fig, v, u, w, 'ko', 'MarkerSize', 3, 'MarkerFaceColor', [1.0, 1.0, 0.0]);
        else
            plot3(fig, v, u, w, 'ko', 'MarkerSize', 1, 'MarkerFaceColor', [0.0, 0.0, 0.0]);
        end
    else
        plot3(fig, v, u, w, 'ko', 'MarkerSize', 3, 'MarkerFaceColor', [0.0,
0.0, 0.0));

end

%plot3(fig, v, u, 'ko', 'MarkerSize', 3, 'MarkerFaceColor', [0.0, 0.0, 0.0]);
xlabel(fig,'v')
ylabel(fig,'u')

else
    axis(fig,1.75*[-pi pi -pi pi], 'square')
    if (handles.sim)
        if (handles.node)
            plot3(fig, w, v, u, 'ko', 'MarkerSize', 3, 'MarkerFaceColor', [1.0, 1.0, 0.0]);
        else
            plot3(fig, w, v, u, 'ko', 'MarkerSize', 1, 'MarkerFaceColor', [0.0, 0.0, 0.0]);
        end
    else
        plot3(fig, w, v, u, 'ko', 'MarkerSize', 3, 'MarkerFaceColor', [0.0, 0.0, 0.0]);
    end
%plot3(fig, w, v, u, 'ko', 'MarkerSize', 3, 'MarkerFaceColor', [0.0, 0.0, 0.0]);
xlabel(fig, '\omega')
ylabel(fig, 'v')
end
hold(fig, 'off')

function idx = getDrawPlane(handles)
if (handles.uibuttongroup1.SelectedObject == handles.uwPlaneButton)
    idx = handles.UW;

end
elseif (handles.uibuttongroup1.SelectedObject == handles.uvPlaneButton)
    idx = handles.UV;
else
    idx = handles.VW;
end

% —— Executes on button press in plotButton.
function plotButton_Callback(hObject, eventdata, handles)
    % hObject    handle to plotButton (see GCBO)
    % eventdata reserved — to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
    [r, c] = size(handles.cpoints);
    u = handles.configTable.Data{handles.U};
    v = handles.configTable.Data{handles.V};
    w = handles.configTable.Data{handles.W};
    c = handles.collided;
    handles.cpoints(r+1,:) = [w, u, v, c];
    guidata(hObject, handles)

% —— Executes on button press in saveButton.
function saveButton_Callback(hObject, eventdata, handles)
    % hObject    handle to saveButton (see GCBO)
    % eventdata reserved — to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)

% —— Executes when entered data in editable cell(s) in configTable.
function configTable_CellEditCallback(hObject, eventdata, handles)
    % hObject    handle to configTable (see GCBO)
    % eventdata structure with the following fields (see MATLAB.UI.CONTROL.
TABLE)

% Indices: row and column indices of the cell(s) edited
% PreviousData: previous data for the cell(s) edited
% EditData: string(s) entered by the user
% NewData: EditData or its converted form set on the Data property.
% Empty if Data was not changed
% Error: error string when failed to convert EditData to appropriate
% value for Data
% handles structure with handles and user data (see GUIDATA)

C = eventdata.Indices;
c = C(2);
data = eventdata.NewData;
if (not(isempty(data)))
    if (c == handles.S)
        %
    elseif (c == handles.U)
        if (data > handles.max_angle)
            data = handles.max_angle;
        elseif (data < handles.min_angle)
            data = handles.min_angle;
        end
        handles.configTable.Data(c) = {data};
    elseif (c == handles.V)
        if (data > handles.max_angle)
            data = handles.max_angle;
        elseif (data < handles.min_angle)
            data = handles.min_angle;
        end
        handles.configTable.Data(c) = {data};
    elseif (c == handles.W)
        % if (data > pi)
data = pi;
elseif (data < -pi)
data = -pi;
end

data = mod(data, 2*pi);
handles.configTable.Data(c) = {data};

assignValuesAllen(handles)

% —— Executes on button press in sEditToggle.

function sEditToggle_Callback(hObject, eventdata, handles)

v = get(hObject, 'Value');

if (v == 0)
    handles.s_minus.Enable = 'off';
    handles.s_plus.Enable = 'off';
else
    handles.s_minus.Enable = 'on';
    handles.s_plus.Enable = 'on';
end

% Hint: get(hObject,'Value') returns toggle state of sEditToggle

% —— Executes on button press in uEditToggle.

function uEditToggle_Callback(hObject, eventdata, handles)

% hObject handle to uEditToggle (see GCBO)
% eventdata reserved – to be defined in a future version of MATLAB

% handles structure with handles and user data (see GUIDATA)

v = get(hObject, 'Value');

handles.configTable.ColumnEditable(handles.U) = v;

if (v == 0)
    handles.u_minus.Enable = 'off';
    handles.u_plus.Enable = 'off';
else
    handles.u_minus.Enable = 'on';
    handles.u_plus.Enable = 'on';
end

% Hint: get(hObject,'Value') returns toggle state of uEditToggle

% --- Executes on button press in vEditToggle.

function vEditToggle_Callback(hObject, eventdata, handles)

% hObject handle to vEditToggle (see GCBO)

% eventdata reserved – to be defined in a future version of MATLAB

% handles structure with handles and user data (see GUIDATA)

v = get(hObject, 'Value');

handles.configTable.ColumnEditable(handles.V) = v;

if (v == 0)
    handles.v_minus.Enable = 'off';
    handles.v_plus.Enable = 'off';
else
    handles.v_minus.Enable = 'on';
    handles.v_plus.Enable = 'on';
end

% Hint: get(hObject,'Value') returns toggle state of vEditToggle
function wEditToggle_Callback(hObject, eventdata, handles)

v = get(hObject, 'Value');

handles.configTable.ColumnEditable(handles.W) = v;
if (v == 0)
    handles.w_minus.Enable = 'off';
    handles.w_plus.Enable = 'off';
else
    handles.w_minus.Enable = 'on';
    handles.w_plus.Enable = 'on';
end

% Hint: get(hObject,'Value') returns toggle state of wEditToggle

function configTable_CreateFcn(hObject, eventdata, handles)

set(hObject, 'Data', cell(1,8));
set(hObject, 'ColumnName', {'S', 'U', 'V', 'Omega (W)', ' ', 'X', 'Y', 'Z'});

function s_plus_Callback(hObject, eventdata, handles)

% hObject handle to s_plus (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

s = handles.configTable.Data{handles.S};
handles.configTable.Data(handles.S) = {s + 0.5};
assignValuesAllen(handles)

% —— Executes on button press in u_plus.
function u_plus_Callback(hObject, eventdata, handles)
% hObject handle to u_plus (see GCBO)
% eventdata reserved — to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

u = handles.configTable.Data{handles.U} + handles.delta;

if (u > handles.max_angle)
  u = handles.max_angle;
end

handles.configTable.Data(handles.U) = {u};
assignValuesAllen(handles)

% —— Executes on button press in v_plus.
function v_plus_Callback(hObject, eventdata, handles)
% hObject handle to v_plus (see GCBO)
% eventdata reserved — to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

v = handles.configTable.Data{handles.V} + handles.delta;

if (v > handles.max_angle)
  v = handles.max_angle;
end

handles.configTable.Data(handles.V) = {v};
assignValuesAllen(handles)

% —— Executes on button press in w_plus.
function w_plus_Callback(hObject, eventdata, handles)
w = handles.configTable.Data{handles.W} + handles.delta;

w = mod(w, 2*pi);

handles.configTable.Data(handles.W) = {w};

assignValuesAllen(handles)

% —— Executes on button press in s_minus.
function s_minus_Callback(hObject, eventdata, handles)

s = handles.configTable.Data{handles.S};

handles.configTable.Data(handles.S) = {s - 0.05};

assignValuesAllen(handles)

% —— Executes on button press in u_minus.
function u_minus_Callback(hObject, eventdata, handles)

u = handles.configTable.Data{handles.U} - handles.delta;

if (u < handles.min_angle)
    u = handles.min_angle;
end

handles.configTable.Data(handles.U) = {u};

assignValuesAllen(handles)

% —— Executes on button press in v_minus.
function v_minus_Callback(hObject, eventdata, handles)
    % hObject    handle to v_minus (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
    v = handles.configTable.Data{handles.V} - handles.delta;
    if (v < handles.min_angle)
        v = handles.min_angle;
    end
    handles.configTable.Data(handles.V) = {v};
    assignValuesAllen(handles)
end

% —— Executes on button press in w_minus.
function w_minus_Callback(hObject, eventdata, handles)
    % hObject    handle to w_minus (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
    w = handles.configTable.Data{handles.W} - handles.delta;
    w = mod(w, 2*pi);
    handles.configTable.Data(handles.W) = {w};
    assignValuesAllen(handles)
end

% —— Executes during object creation, after setting all properties.
function axesLamp_CreateFcn(hObject, eventdata, handles)
    % hObject    handle to axesLamp (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    empty - handles not created until after all CreateFcns
called

% Hint: place code in OpeningFcn to populate axesLamp
% —— Executes when selected object is changed in uibuttongroup1.
function uibuttongroup1_SelectionChangedFcn(hObject, eventdata, handles)
% hObject handle to uibuttongroup1
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
updatePlot(handles);

% —— Executes on button press in continuumSelectButton.
function continuumSelectButton_Callback(hObject, eventdata, handles)
% hObject handle to continuumSelectButton (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of continuumSelectButton

% —— Executes when selected object is changed in uibuttongroupRobot.
function uibuttongroupRobot_SelectionChangedFcn(hObject, eventdata, handles)
% hObject handle to uibuttongroupRobot
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
if (hObject == handles.continuumSelectButton)
    handles.rigid = 0;
else
    handles.rigid = 1;
end
assignValuesAllen(handles)
guidata(hObject, handles)
% —— Executes on button press in sim_enable.

function sim_enable_Callback(hObject, eventdata, handles)

% hObject    handle to sim_enable (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of sim_enable
val = get(hObject, 'Value');
if (val)
    % enable the "load sim file" button
    set(handles.file_button, 'Enable', 'on');

    handles.sim = true;

    % disable "interactive mode"
    set(handles.sEditToggle, 'Value', 0);
    set(handles.uEditToggle, 'Value', 0);
    set(handles.vEditToggle, 'Value', 0);
    set(handles.wEditToggle, 'Value', 0);
    set(handles.sEditToggle, 'Enable', 'off');
    set(handles.uEditToggle, 'Enable', 'off');
    set(handles.vEditToggle, 'Enable', 'off');
    set(handles.wEditToggle, 'Enable', 'off');
    set(handles.s_minus, 'Enable', 'off');
    set(handles.s_plus, 'Enable', 'off');
    handles.configTable.ColumnEditable(handles.S) = 0;
    set(handles.u_minus, 'Enable', 'off');

end
```plaintext
set(handles.u_plus, 'Enable', 'off');

handles.configTable.ColumnEditable(handles.U) = 0;
set(handles.v_minus, 'Enable', 'off');
set(handles.v_plus, 'Enable', 'off');

handles.configTable.ColumnEditable(handles.V) = 0;
set(handles.w_minus, 'Enable', 'off');
set(handles.w_plus, 'Enable', 'off');

handles.configTable.ColumnEditable(handles.W) = 0;

else
    % disable the "load sim file" button
    set(handles.file_button, 'Enable', 'off');

    handles.sim = false;

    % enable "interactive mode"
    set(handles.sEditToggle, 'Enable', 'on');
    set(handles.uEditToggle, 'Enable', 'on');
    set(handles.vEditToggle, 'Enable', 'on');
    set(handles.wEditToggle, 'Enable', 'on');

    set(handles.sEditToggle, 'Value', 0);
    set(handles.uEditToggle, 'Value', 1);
    set(handles.vEditToggle, 'Value', 1);
    set(handles.wEditToggle, 'Value', 1);

    set(handles.s_minus, 'Enable', 'off');
    set(handles.s_plus, 'Enable', 'off');

    handles.configTable.ColumnEditable(handles.S) = 0;
    set(handles.u_minus, 'Enable', 'on');
    set(handles.u_plus, 'Enable', 'on');

    handles.configTable.ColumnEditable(handles.U) = 1;
```

250
```matlab
set(handles.v_minus, 'Enable', 'on');
set(handles.v_plus, 'Enable', 'on');
handles.configTable.ColumnEditable(handles.V) = 1;
set(handles.w_minus, 'Enable', 'on');
set(handles.w_plus, 'Enable', 'on');
handles.configTable.ColumnEditable(handles.W) = 1;
end

guidata(hObject, handles)

function file_button_Callback(hObject, eventdata, handles_in)

% hObject handle to file_button (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles_in structure with handles and user data (see GUIDATA)
handles = guidata(hObject);

% get the path
[file, path] = uigetfile('simfiles/*.csv');
handles.rawpath = csvread([path, file], 1, 0);
[r,˜] = size(handles.rawpath);
path_len = handles.rawpath(r,1);
path_idx = handles.rawpath(r,2:(path_len+1)) + ones(1,path_len);
handles.path = zeros(path_len, 4);
for i = 1:path_len
    handles.path(i,:) = [handles.state(1), handles.rawpath(path_idx(i),3),...
                        handles.rawpath(path_idx(i),4), handles.rawpath(path_idx(i),2)];
end

% User Data for the Timer Object
```
data.t = 0;
data.tau = 0.01;
data.state = handles.state;
data.path = handles.path;
data.path_idx = 1;
data.path_len = path_len;
data.K = [100, 0, 0; 200, 0, 0];
data.ravg = zeros(5,20);
data.samples = 1;
data.elast = [0, 0, 0, 0, 0];
data.gui = hObject.Parent;
data.handles = handles;

xtimer = timer;
xtimer.TimerFcn = @execute_timestep;
xtimer.StopFcn = @cleanup_timer;
xtimer.ExecutionMode = 'fixedSpacing';
xtimer.Period = 0.01;
xtimer.StartDelay = 3;
xtimer.Tag = 'exe_timer';
xtimer.UserData = data;
guida(hObject, handles);

function start(xtimer);

function execute_timestep(obj, ~)
S = 1; U = 2; V = 3; W = 4;
epsilon = 0.01;
data = get(obj, 'UserData');
data.t = data.t + data.tau;
data.t;
setpt = data.path(data.path_idx,:);
% data has the following properties
% t :
% tau :
% state : [s, u, v, w]
% K : [kp, kd, ki]
% ravg : [ [], [], [], [], []]
% samples :
% elast : [el1, el2, el3, el4]
s_0 = 0.95; d = 0.022;
u = data.state(U); v = data.state(V); s = data.state(S);
l = [0, -2*d, 1; 2*d, 0, 1; 0, 2*d, 1; -2*d, 0, 1] *[u;v;s];
u = setpt(U); v = setpt(V); s = setpt(S);
l_set = [0, -2*d, 1; 2*d, 0, 1; 0, 2*d, 1; -2*d, 0, 1] *[u;v;s];
out = zeros(1,5);
err = [l_set - l; setpt(W)-data.state(W)];
for i = 1:5
    if (data.samples < 20)
        data.ravg(i,data.samples) = err(i);
        data.samples = data.samples + 1;
    else
        data.ravg(i,:) = [data.ravg(i,2:20), err(i)];
    end
    if (i < 5) % tendons
        out(i) = err(i) * data.K(1,1) + ... ( err(i) - data.elast(i))/data.t * data.K(1,2) + ... (sum(data.ravg(i))/data.samples) * data.K(1,3);
    else % turntable
        out(i) = err(i) * data.K(2,1) + ...
(err(i) - data.elast(i))/data.t * data.K(2,2) + ... 
(sum(data.ravg(i))/data.samples) * data.K(2,3);
end
% saturate
if (out(i) > 12)
  out(i) = 12;
elseif (out(i) < -12)
  out(i) = -12;
end
end

% simulate
% tendon lengths
l_next = l + (out(1:4))'/(12*(32/60)*data.tau*0.44*pi);
state_next = (1/(4*d)) * [0, 1, 0, -1; -1, 0, 1, 0; d, d, d, d] * l_next;

% turntable (pi/9) rad/s at 12V
w_next = data.state(W) + (out(5)/12)*(pi/9)*data.tau*15;
data.state = [state_next(3), state_next(1), state_next(2), w_next];

% update the GUI
handles = guidata(data.gui);
handles.state = data.state;
handles.configTable.Data(S) = {handles.state(S)};
handles.configTable.Data(U) = {handles.state(U)};
handles.configTable.Data(V) = {handles.state(V)};
handles.configTable.Data(W) = {handles.state(W)};
delta = abs(data.state - setpt);

b = (delta < epsilon*ones(1,4));

if (b)
    data.path_idx = data.path_idx + 1;
    handles.node = true;
    if (data.path_idx > data.path_len)
        stop(obj)
    end
else
    handles.node = false;
end

guida(data.gui, handles);
assignValuesAllen(handles);

obj.UserData = data;

function cleanup_timer(obj, ~)
delete(obj);

Listing 37: Kinematic GUI Source

function collided = Lamp.Sim_GUI(s, u, v, w, handles)

num_spheres = 29;
big_rad = 30;
lil_rad = 15;
point_a = zeros(num_spheres,3);

% Rotation matrix (by theta)
T = [ cos(w), -sin(w), 0, 0;
     sin(w),  cos(w), 0, 0;
     0,   0,   1,  0;
     0,   0,   0,  1 ];

for i = 1: num_spheres
    scale = i / num_spheres;
    H_a = T * tranMatrixA(u * scale, v * scale, s * scale);
    point_a(1,1,i) = H_a(1,4);
    point_a(1,2,i) = H_a(2,4);
    point_a(1,3,i) = H_a(3,4);
    point_a(i,1) = H_a(1,4);
    point_a(i,2) = H_a(2,4);
    point_a(i,3) = H_a(3,4);
end

fig = handles.axesLamp;

% Update position of end effector
handles.configTable.Data(handles.X) = {H_a(1,4)};
handles.configTable.Data(handles.Y) = {H_a(2,4)};
handles.configTable.Data(handles.Z) = {H_a(3,4)};

hold(fig, 'on')
grid(fig, 'on')
axis(fig,[−1 1 −1 1 0 2])
xlabel(fig,'x')
ylabel(fig,'y')

R = [ cos(w), -sin(w), 0;
     sin(w), cos(w), 0;
     0, 0, 1; ];

% plot base
ext = 0.12; %read_la
% "read in" points for objects
%points;
[h; o; la; point_base; point_base_la; point_arm_la; point_orient;
[offset, point_base, point_orient,...
    point_grp1, point_grp2, point_grp3, ...]
point_base_la, point_arm_la, ...
point_shelf1, point_cup1, point_cup2, point_cup3] = points(ext,
    point_a(num_spheres,:),H_a);

% [offset, point_base, point_orient,...
%    point_grp1, point_grp2, point_grp3, ...
%    point_base_la, point_arm_la, ...
%    point_shelf1, point_shelf2, point_cup1, point_cup2, point_cup3] =
    points2(ext, point_a(num_spheres,:),H_a);

% Lamp box/base
V = zeros(8,3);
for i=1:8
    V(i,:) = (R*point_base.p(i,:))';
end

F = [1,2,4,3; 5,6,8,7; 1,2,6,5; 2,4,8,6; 3,4,8,7; 1,3,7,5];
patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [1.0 0 0], 'EdgeColor'
    , [0 0 0]);

%p = patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [1.0 0 0],'
EdgeColor', [0 0 0], ...  
% 'FaceVertexAlphaData', 0.2, 'FaceAlpha', 'flat');  
% Lamp orientation  
for i=1:4  
    V(i,:) = (R*point_orient(i,:))';  
end  
plot3(fig,V(1:2,1),V(1:2,2),V(1:2,3),'k-','LineWidth',2);  
plot3(fig,V(2:3,1),V(2:3,2),V(2:3,3),'k-','LineWidth',1.5);  
plot3(fig,V(2:2:4,1),V(2:2:4,2),V(2:2:4,3),'k-','LineWidth',1.5);  
% Gripper  
for i=1:8  
    V(i,:) = (eye(3)*point_grp1.p(i,:))';  
end  
patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [1.0 0 0], 'EdgeColor', [0 0 0]);  
%patch(fig,'Faces', F, 'Vertices', W(:,1:3), 'FaceColor', [1.0 0.8 0.8], 'EdgeColor', [0 0 0]);  
for i=1:8  
    V(i,:) = (eye(3)*point_grp2.p(i,:))';  
end  
patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [1.0 0 0], 'EdgeColor', [0 0 0]);  
%patch(fig,'Faces', F, 'Vertices', W(:,1:3), 'FaceColor', [1.0 0.8 0.8], 'EdgeColor', [0 0 0]);  
for i=1:8  
    V(i,:) = (eye(3)*point_grp3.p(i,:))';  
end  
patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [1.0 0 0], 'EdgeColor', [0 0 0]);  
%patch(fig,'Faces', F, 'Vertices', W(:,1:3), 'FaceColor', [1.0 0.8 0.8], 'EdgeColor', [0 0 0]);
for i=1:8
    V(i,:) = (R*point_base_la.p(i,:))';
end

patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [0.25 0.25 0.25], 'EdgeColor', [0 0 0]);

% LA arm
for i=1:8
    V(i,:) = (R*point_arm_la.p(i,:))';
end

patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [0.5 0.5 0.5], 'EdgeColor', [0 0 0]);

% shelf1
for i=1:8
    V(i,:) = (eye(3,3)*point_shelf1.p(i,:))';
end

patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [0.4 0.4 0.4], 'EdgeColor', [0 0 0]);

% shelf2
for i=1:8
    V(i,:) = (eye(3,3)*point_shelf2.p(i,:))';
end

patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [0.4 0.4 0.4], 'EdgeColor', [0 0 0]);

% cup1
for i=1:8
    V(i,:) = (eye(3,3)*point_cup1.p(i,:))';
end

patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [0.8 0.2 0.2], 'EdgeColor', [0 0 0]);

% cup2
for i=1:8
    V(i,:) = (eye(3,3)*point_cup2.p(i,:))';
end
patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [0.8 0.8 0.8], 'EdgeColor', [0 0 0]);
%
cup3
for i=1:8
    V(i,:) = (eye(3,3)*point_cup3.p(i,:))';
end
patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [0.8 0.2 0.2], 'EdgeColor', [0 0 0]);

% Plot the continuum arm
for k=1:num_spheres-1
    if (mod(k,4)==1)
        rad = big_rad;
    else
        rad = lil_rad;
    end
    plot3(fig,point_a(k,1),point_a(k,2),point_a(k,3)+offset,'.k','MarkerSize',rad);
end

% C-Space
% fig = handles.axesCspace;
% hold(fig,'on')
% grid(fig,'on')
% axis(fig,[-2*pi 2*pi -2*pi 2*pi -2*pi 2*pi])
% xlabel(fig,'Omega')
% ylabel(fig,'U')

260
%%% check for collisions

```matlab
% c_obs = {point_shelf1, point_cup1, point_cup2, point_cup3};
collided = check_collision(point_a, offset, {point_grp1, point_grp2, point_grp3}, c_obs);
```

%%% plotting c-space
% if (~collided)
```matlab
%     plot3(fig, w, u, v, 'ko', ...
%          'MarkerSize', 3, 'MarkerFaceColor', [0.0, 0.0, 0.0]);
% end
```

% plotting c-space
% if (~collided)
```matlab
%     plot3(fig, w, u, v, 'rx', ...
%          'LineWidth', 2);
% end
```

```matlab
[r, ~] = size(handles.cpoints);
for i = 2:r
```
```matlab
%     if (handles.cpoints(i,4))
%         plot3(fig, handles.cpoints(i,1), handles.cpoints(i,2), handles.cpoints(i,3), ...
%              'x', 'LineWidth', 2, 'LineColor', [0.2, 0.2, 0.8]);
%     else
%         plot3(fig, handles.cpoints(i,1), handles.cpoints(i,2), handles.cpoints(i,3), ...
%              'o', 'MarkerSize', 3, 'MarkerFaceColor', [0.2, 0.2, 0.8]);
%     end
```
% end

Listing 38: Main continuum simulation file of the Kinematic GUI

```matlab
function plot_torus(u,v,w, fig)
R = 3*sqrt(2)*pi;
```
\[ r = \sqrt{2} \cdot \pi; \]
\[ \theta = (0:0.01:2 \cdot \pi)'; \]
\[ \phi = 0:0.01:2 \cdot \pi; \]
\[ \text{len} = \text{length}(\phi); \]
\[ X = (R + r \cdot \cos(\theta)) \cdot \cos(\phi); \]
\[ Y = (R + r \cdot \cos(\theta)) \cdot \sin(\phi); \]
\[ Z = r \cdot \sin(\theta) \cdot \text{ones}(1, \text{len}); \]
\[ s = \text{surfacing}(\text{fig},X,Y,Z); \]
\[ \text{set}(s, 'FaceColor', [0, 0, 1]); \]
\[ \text{set}(s, 'FaceAlpha', 0.2); \]
\[ \text{set}(s, 'FaceLighting', 'gouraud'); \]
\[ \text{set}(s, 'EdgeColor', 'none'); \]
\[ \text{set}(s, 'EdgeAlpha', 0.1); \]
\[ \% X = (R + r \cdot \cos(\theta)) \cdot \cos(\phi); \]
\[ \% Y = (R + r \cdot \cos(\theta)) \cdot \sin(\phi); \]
\[ \% Z = r \cdot \sin(\theta) \cdot \text{ones}(1, \text{len}); \]
\[ \% \text{end plane} \]
\[ \text{endx} = (R + r \cdot \cos(\theta')) \cdot \cos(-\pi); \]
\[ \text{endy} = (R + r \cdot \cos(\theta')) \cdot \sin(-\pi); \]
\[ \text{psi} = w; \]
\[ x = (R + r \cdot \cos(\theta')) \cdot \cos(\psi); \]
\[ y = (R + r \cdot \cos(\theta')) \cdot \sin(\psi); \]
\[ z = r \cdot \sin(\theta'); \]
\[ \text{patch}(\text{fig}, 'XData', \text{endx}, 'YData', \text{endy}, 'ZData', z, 'FaceColor', [0 0 0], 'EdgeColor', [0 0 0], 'FaceAlpha', 1.0); \]
\[ \text{patch}(\text{fig}, 'XData', x, 'YData', y, 'ZData', z, 'FaceColor', [1 0 0], '
\texttt{EdgeColor}', [0 0 0], 'FaceAlpha', 0.7);

\% patch ('XData', R*cos(\phi), 'YData', R*sin(\phi), 'ZData', zeros(1, len), '
        FaceColor', [0 0 1], 'EdgeColor', [0 0 0], 'FaceAlpha', 0);

\texttt{hold on}

m = sqrt(u^2 + v^2);

\texttt{th = atan2(v, u);
}

cx = R*cos(psi); cy = R*sin(psi); cz = 0;

\texttt{xx = (R + m*cos(th))*cos(psi); yy = (R + m*cos(th))*sin(psi); zz = m*sin(th);
}

% edge of the slice

cxx = (R + r*cos(pi))*cos(psi);

cyy = (R + r*cos(pi))*sin(psi);

czz = r*sin(pi);

% center of slice

\texttt{plot3(fig, cx, cy, cz, 'k.', 'MarkerSize', 12);

\% configuration c = [u v w]

\texttt{plot3(fig, xx, yy, zz, 'k.', 'MarkerSize', 16);

\% \omega = 0 axis for reference

\texttt{plot3(fig, [0,R-r],[0,0],[0,0], 'k--', 'LineWidth', 1.3);

\% \omega line

\texttt{plot3(fig, [0,cxx],[0,cyy],[0,czz], 'k--', 'LineWidth', 1.3);

\% u

\texttt{plot3(fig, [cx,xx],[cy,yy],[0,0], 'k--');

\% v

\texttt{plot3(fig, [cx,xx],[cy,yy],[0,0], 'k--');

263
Listing 39: Function to plot the c-space of CuRLE

```matlab
function [ht, point_base, point_orient, ...
    point_grp1, point_grp2, point_grp3, ...
    point_base_la, point_arm_la, ...
    point_shelf, point_cup1, point_cup2, point_cup3] = points(ext, arm_end, Ha)

T = [
    -0.5, -0.5, -0.5;
    -0.5, 0.5, -0.5;
    0.5, -0.5, -0.5;
    0.5, 0.5, -0.5;
    -0.5, -0.5, 0.5;
    -0.5, 0.5, 0.5;
    0.5, -0.5, 0.5;
    0.5, 0.5, 0.5;
];

ht = h_ + o + la + ext;

w = 0.495; l = 0.495; h = h_;
c = [0, 0, o+h_/2];
point_base.p = T*diag([w l h]) + ones(8,3)*diag(c);
point_base.w = w;
point_base.l = l;
point_base.h = h;
point_base.c = c;
```
point_orient = [
    0, 0, o+h_;
    0, -1/2+0.1, o+h_;
    -w/8, -1/8, o+h_;
    w/8, -1/8, o+h_;
];

% gripper1
w = 0.088;  l = 0.02;  h = 0.01;
c = [ arm_end(1), arm_end(2), arm_end(3)+h/2+ht ];
c = [ 0, 0, 0 ];
c_rot = ( Ha*[c';0] );
p = T*diag([w l h]);
for i = 1:8
    temp = ( Ha*[p(i,: )';0] )';
    point_grp1.p(i,:) = temp(1,1:3) + arm_end + c_rot(1:3,1)' + [0, 0, ht];
end
point_grp1.w = w;
point_grp1.l = l;
point_grp1.h = h;
point_grp1.c = c;

% gripper2
w = 0.02;  l = 0.01;  h = 0.10;
c = [ arm_end(1)+0.033+w/2, arm_end(2), arm_end(3)−0.01+h/2+ht ];
c = [0.044+w/2, 0, h/2];
c_rot = ( Ha*[c';0] );
%p = T*diag([w l h]) + ones(8,3)*diag(c_rot(1:3,:));
p = T*diag([w l h]);
for i = 1:8
    temp = ( Ha*[p(i,: )';0] )';
    point_grp2.p(i,:) = temp(1,1:3) + arm_end + c_rot(1:3,1)' + [0, 0, ht]
end
point_grp2.w = w;
point_grp2.l = l;
point_grp2.h = h;
point_grp2.c = c;

w = 0.02; l = 0.01; h = 0.10;
%c = [arm_end(1)−0.033−w/2, arm_end(2), arm_end(3)−0.01+h/2+ht];
c = [−0.044−w/2, 0, h/2];
c_rot = (Ha*[c';0]);
%p = T*diag([w l h]) + ones(8,3)*diag(c_rot(1:3,:));
p = T*diag([w l h]);
for i = 1:8
    temp = (Ha*[p(i,:)',0])';
    point_grp3.p(i,:) = temp(1,1:3)+arm_end + c_rot(1:3,1') + [0, 0, ht];
end
point_grp3.w = w;
point_grp3.l = l;
point_grp3.h = h;
point_grp3.c = c;

w = 0.035; l = 0.035; h = la;
c = [0,0,o+h,1a/2];
point_base_la.p = T*diag([w l h]) + ones(8,3)*diag(c);
point_base_la.w = w;
point_base_la.l = l;
point_base_la.h = h;
point_base_la.c = c;

%% la_arm
w = 0.0175; l = 0.0175; h = ext;
c = [0, 0, o+h+la+ext/2];
point_arm_la.p = T*diag([w l h]) + ones(8,3)*diag(c);
point_arm_la.w = w;
point_arm_la.l = l;
point_arm_la.h = h;
point_arm_la.c = c;

shelf
w = 0.26; l = 0.66; h = 0.045;
c = [0.75, 0, 1.14];
point_shelf.p = T*diag([w l h]) + ones(8,3)*diag(c);
point_shelf.w = w;
point_shelf.l = l;
point_shelf.h = h;
point_shelf.c = c;

cup1
w = 0.07; l = 0.07; h = 0.20;
c = [0.75, -0.17, 1.265];
point_cup1.p = T*diag([w l h]) + ones(8,3)*diag(c);
point_cup1.w = w;
point_cup1.l = l;
point_cup1.h = h;
point_cup1.c = c;

cup2
w = 0.07; l = 0.07; h = 0.11;
c = [0.75, 0, 1.22];
point_cup2.p = T*diag([w l h]) + ones(8,3)*diag(c);
point_cup2.w = w;
point_cup2.l = l;
Listing 40: Function that returns all the vertex points of the simulation environment

```matlab
function collided = rigidLamp_Sim_GUI(s, u, v, w, handles)
    num_spheres = 29;
    big_rad = 30;
    lil_rad = 15;
    point_a = zeros(num_spheres, 3);
    % rigidDiameter = 8; % big Balls
    rigidDiameter = 5; % small Balls
    % Rotation matrix (by theta)
    T = [ cos(w), -sin(w), 0, 0; sin(w), cos(w), 0, 0; 0, 0, 1, 0; 0, 0, 0, 1 ];
```

```matlab
point_cup2.h = h;
point_cup2.c = c;

%% cup3
w = 0.06; l = 0.06; h = 0.17;
c = [0.75, 0.26, 1.25];
point_cup3.p = T * diag([w l h]) + ones(8,3) * diag(c);
point_cup3.w = w;
point_cup3.l = l;
point_cup3.h = h;
point_cup3.c = c;
end
```
% Mod by 2*pi to "achieve full c-space"

\[
\theta = \sqrt{\left(\frac{\text{mod}(u, 2\pi)}{2}\right)^2 + \left(\frac{\text{mod}(v, 2\pi)}{2}\right)^2};
\]

\[
\phi = \text{atan2}(\text{mod}(v, 2\pi), -\text{mod}(u, 2\pi));
\]

% Actual equation for chords

\[
\theta = \sqrt{\left(\frac{u}{2}\right)^2 + \left(\frac{v}{2}\right)^2};
\]

\[
\phi = \text{atan2}(-v, -u);
\]

% equation of chord length

\[
L = 2 \times \left(\frac{s}{\theta \times 2}\right) \times \sin\left(\frac{\theta \times 2}{2}\right);
\]

\[
L = s;
\]

\[
\Phi = [\cos(\phi), -\sin(\phi), 0, 0];
\]

\[
\sin(\phi), \cos(\phi), 0, 0;
\]

\[
0, 0, 1, 0;
\]

\[
0, 0, 0, 1;\]

\[
\Theta = [1, 0, 0, 0];
\]

\[
0, \cos(\theta), \sin(\theta), 0;
\]

\[
0, -\sin(\theta), \cos(\theta), 0;
\]

\[
0, 0, 0, 1;\]

\[
S = [1, 0, 0, 0];
\]

\[
0, 1, 0, 0;
\]

\[
0, 0, 1, L;
\]

\[
0, 0, 0, 1;\]

\[
\text{for } i = 1: \text{num\_spheres}
\]

\[
\text{scale} = i / \text{num\_spheres};
\]
H_a = T*tranMatrixA(u*scale, v*scale, s*scale);

% point_a(1,1,i) = H_a(1,4);
% point_a(1,2,i) = H_a(2,4);
% point_a(1,3,i) = H_a(3,4);
point_a(i,1) = H_a(1,4);
point_a(i,2) = H_a(2,4);
point_a(i,3) = H_a(3,4);

end

%H_a for rigidLink chord
H_a = T*PHI*THETA*S*THETA*PHI;
endPoint = transpose(squeeze(H_a(1:3,4)));

fig = handles.axesLamp;

% update position of end effector
handles.configTable.Data(handles.X) = {H_a(1,4)};
handles.configTable.Data(handles.Y) = {H_a(2,4)};
handles.configTable.Data(handles.Z) = {H_a(3,4)};

hold(fig, 'on')
grid(fig, 'on')
axis(fig, [-1 1 -1 1 0 2])
xlabel(fig, 'x')
ylabel(fig, 'y')

R = [ cos(w), -sin(w), 0;
     sin(w), cos(w), 0;]
    0, 0, 1;
    
    % plot base
    ext = 0.12; %read_la
    % "read in" points for objects
    %points;
    %h ; o ; la ; point_base ; point_base_la ; point_arm_la ; point_orient;
    % [offset, point_base, point_orient, ...
    %   point_grp1, point_grp2, point_grp3, ...
    %   point_base_la, point_arm_la, ...
    %   point_shelf, point_cup1, point_cup2, point_cup3] = points(ext,
    point_a(num_spheres,:), H_a);

    [offset, point_base, point_orient,...
     point_grp1, point_grp2, point_grp3, ...
     point_base_la, point_arm_la, ...
     point_shelf, point_cup1, point_cup2, point_cup3] = points(ext,
     endPoint, H_a);

    % Lamp box/base
    V = zeros(8,3);
    for i=1:8
        V(i,:) = (R*point_base.p(i,:))';
    end
    F = [1,2,4,3; 5,6,8,7; 1,2,6,5; 2,4,8,6; 3,4,8,7; 1,3,7,5];
    patch(fig,'Faces', F, 'Vertices', V, 'FaceColor', [1.0 0 0], 'EdgeColor'
    , [0 0 0]);

    % Lamp orientation
    for i=1:4
        V(i,:) = (R*point_orient(i,:))';
    end
plot3(fig,V(1:2,1),V(1:2,2),V(1:2,3),'k−','LineWidth',2);
plot3(fig,V(2:3,1),V(2:3,2),V(2:3,3),'k−','LineWidth',1.5);
plot3(fig,V(2:2:4,1),V(2:2:4,2),V(2:2:4,3),'k−','LineWidth',1.5);

% Gripper
for i=1:8
    V(i,:) = (eye(3)*point_grp1.p(i,:))';
end
patch(fig,'Faces',F,'Vertices',V,'FaceColor',[1.0 0 0], 'EdgeColor',[0 0 0]);
%patch(fig,'Faces',F,'Vertices',W(:,1:3),'FaceColor',[1.0 0.8 0.8],
    'EdgeColor',[0 0 0]);
for i=1:8
    V(i,:) = (eye(3)*point_grp2.p(i,:))';
end
patch(fig,'Faces',F,'Vertices',V,'FaceColor',[1.0 0 0], 'EdgeColor',[0 0 0]);
%patch(fig,'Faces',F,'Vertices',W(:,1:3),'FaceColor',[1.0 0.8 0.8],
    'EdgeColor',[0 0 0]);
for i=1:8
    V(i,:) = (eye(3)*point_grp3.p(i,:))';
end
patch(fig,'Faces',F,'Vertices',V,'FaceColor',[1.0 0 0], 'EdgeColor',[0 0 0]);
%patch(fig,'Faces',F,'Vertices',W(:,1:3),'FaceColor',[1.0 0.8 0.8],
    'EdgeColor',[0 0 0]);
% LA base
for i=1:8
    V(i,:) = (R*point_base_la.p(i,:))';
end
patch(fig,'Faces',F,'Vertices',V,'FaceColor',[0.25 0.25 0.25], 'EdgeColor',[0 0 0]);
% LA arm
for i = 1:8
    V(i,:) = (R * point_arm_la.p(i,:)')';
end

patch(fig, 'Faces', F, 'Vertices', V, 'FaceColor', [0.5 0.5 0.5], 'EdgeColor', [0 0 0]);

% shelf
for i = 1:8
    V(i,:) = (eye(3,3)*point_shelf.p(i,:)')';
end

patch(fig, 'Faces', F, 'Vertices', V, 'FaceColor', [0.4 0.4 0.4], 'EdgeColor', [0 0 0]);

for i = 1:8
    V(i,:) = (eye(3,3)*point_cup1.p(i,:)')';
end

patch(fig, 'Faces', F, 'Vertices', V, 'FaceColor', [0.8 0.2 0.2], 'EdgeColor', [0 0 0]);

for i = 1:8
    V(i,:) = (eye(3,3)*point_cup2.p(i,:)')';
end

patch(fig, 'Faces', F, 'Vertices', V, 'FaceColor', [0.8 0.8 0.2], 'EdgeColor', [0 0 0]);

for i = 1:8
    V(i,:) = (eye(3,3)*point_cup3.p(i,:)')';
end

patch(fig, 'Faces', F, 'Vertices', V, 'FaceColor', [0.8 0.2 0.2], 'EdgeColor', [0 0 0]);

% Plot the continuum arm
% for k = 1:num_spheres - 1
%     if (mod(k, 4) == 1)
% rad = big_rad;
% else
% rad = lil_rad;
% end
% plot3(fig, point_a(k,1), point_a(k,2), point_a(k,3)+offset, '.k',
        'MarkerSize', rad);
% end

% Plot the rigid link arm
plot3(fig, [0 endPoint(1)], [0 endPoint(2)], [offset endPoint(3)+offset], '-k', 'LineWidth', rigidDiameter)

% C-Space
fig = handles.axesCspace;
hold(fig, 'on')
grid(fig, 'on')
axis(fig, [-pi pi -pi pi -pi pi])
xlabel(fig, 'Omega')
ylabel(fig, 'U')

% check for collisions
% c_obs = {point_shelf, point_cup1, point_cup2, point_cup3};
% collided = check_collision(point_a, offset, {point_grp1, point_grp2, point_grp3}, c_obs);
collided = false;

% if (~collided)
% plot3(fig, th, u, v, 'ko', ...
%        'MarkerSize', 3, 'MarkerFaceColor', [0.0, 0.0, 0.0]);
% else
% plot3(fig, th, u, v, 'rx', ...
% 'LineWidth', 2);
% end
% [r, ~] = size(handles.cpoints);
% for i = 2:r
% if (handles.cpoints(i,4))
%  plot3(fig, handles.cpoints(i,1), handles.cpoints(i,2), handles.cpoints(i,3), ...
%   'x', 'LineWidth', 2, 'LineColor', [0.2, 0.2, 0.8]);
% else
%  plot3(fig, handles.cpoints(i,1), handles.cpoints(i,2), handles.cpoints(i,3), ...
%   'o', 'MarkerSize', 3, 'MarkerFaceColor', [0.2, 0.2, 0.8]);
% end
% end
end

Listing 41: Main rigid-link simulation file of the Kinematic GUI

function R = rot_mat(th, u, v)
c1 = cos(u); s1 = sin(u);
c2 = cos(v); s2 = sin(v);
c3 = cos(th); s3 = sin(th);
Rz = [c3, -s3, 0; s3, c3, 0; 0, 0, 1];
Ry = [c2, 0, s2; 0, 1, 0; -s2, 0, c2];
Rx = [1, 0, 0; 0, c1, -s1; 0, s1, c1];
R = Rx*Ry*Rz;
end

Listing 42: Function to generate a rotation matrix

function v_prime = rotate(v, R)
v_prime = zeros(size(v));
[r, ~] = size(v);
for i = 1:r
    v_prime(i,:) = (R * v(i,:)' )' ;
end
end

Listing 43: Function to rotate a point by matrix

function Ha = tranMatrixA(u,v,s)

theta = sqrt(u^2 + v^2);
gam = (cos(theta) − 1)/(theta^2);
zet = sin(theta)/theta;

Ha = [gam*(v^2) + 1, −gam*v*u, zet*v, −gam*s*v;
      −gam*u*v, gam*(u^2) + 1, −zet*u, gam*s*u;
      −zet*v, zet*u, cos(theta), zet*s ;
      0, 0, 0, 1 ];
end

Listing 44: Function that returns the transformation matrix of the continuum element

function plot_graph(fig, data, goal, task)
figure(fig);
limits = [−pi pi −pi pi ];
%goal = [1.57, 1.76, 0];
goalIsConnected = false;
goalId = −1;
id = data(:,1);
state.th = data(:,2);
state.u = data(:,3);

276
state.v = data(:,4);

epsilon = 0.0001;

if (task == 1)
    load('C:\Users\zhawks\Documents\roboticsLab\matlab\gui_sim\cspace_shelf_grp.mat', 'points');
elseif (task == 2)
    load('C:\Users\zhawks\Documents\roboticsLab\matlab\gui_sim_thesis\cspace_shelf_grp2.mat', 'points');
end

[r,~] = size(points);
cspace = [points(:,1:2), zeros(r,1)];

% determine edges
[r,c] = size(data);

% index of the path is the last row
idx_p = r;

% preset all edges to -1
r = r-1;
edges = ones(r,c-4)*(-1);

% determine which id is the goal id
for i = 1:r
    delta = [abs(state.th(i) - goal(1)); abs(state.u(i) - goal(2)); abs(state.v(i) - goal(3))];
    if (delta(1) <= epsilon &&...
```matlab
    delta(2) <= epsilon &&
    delta(3) <= epsilon)
    goalIsConnected = true;
    goalId = id(i) + 1;
    end
end

opt_path = data(idx_p,:);

for i=1:r
    j = 5;
    while (j < c && data(i,j) ~= 0)
        edges(i,j-4) = data(i,j);
        edges(i,j+1-4) = data(i,j+1);
        j = j + 2;
    end
end

hold on

MarkerSize = 4;

% plot the start state
plot(state.th(1), state.u(1), 'go', ...
     'MarkerSize',MarkerSize, ...
     'MarkerEdgeColor','k', ...
     'MarkerFaceColor',[0,1,0]);

% plot all the other states
plot(state.th(2:r), state.u(2:r), 'bo', ...
     'MarkerSize',MarkerSize, ...
     'MarkerEdgeColor','k', ...
     'MarkerFaceColor',[0.5,0.5,0.5]);
plot (goal(1), goal(2), 'ro', ...
    'MarkerSize',MarkerSize, ...
    'MarkerEdgeColor','k', ...
    'MarkerFaceColor',[1.0,0.0,0]);

% plot first part of path & c-obs (for legend)
plot (state.th(opt_path(1)+1), state.u(opt_path(1)+1), 'go', ...
    'MarkerSize',MarkerSize, ...
    'MarkerEdgeColor','k', ...
    'MarkerFaceColor',[1.0,1.0,0]);
plot (cspace(1,1), cspace(1,2), 'k.');

% plot edges
[er, ec] = size(edges);
for i = 1:er
    pt1 = [state.th(i), state.u(i)];
    for j = 2:2:ec
        val = edges(i,j);
        if val ~= -1 && val < r
            pt2 = [state.th(val+1), state.u(val+1)];
            plot ([pt1(1), pt2(1)], [pt1(2), pt2(2)], 'k-');
        end
    end
end

plot (state.th(2:r), state.u(2:r), 'bo', ...
    'MarkerSize',MarkerSize, ...
    'MarkerEdgeColor','k', ...
    'MarkerFaceColor',[0.5,0.5,0.5]);

% plot path
for i = 1:length(opt_path)
    plot(state.th(opt_path(i)+1), state.u(opt_path(i)+1), 'go', ...
    'MarkerSize',MarkerSize, ...
    'MarkerEdgeColor','k', ...
    'MarkerFaceColor',[1.0,1.0,0]);
end

% re-plot the start
plot(state.th(1), state.u(1), 'go', ...
    'MarkerSize',MarkerSize, ...
    'MarkerEdgeColor','k', ...
    'MarkerFaceColor',[0,1.0,0]);

% re-plot the goal
plot(goal(1), goal(2), 'ro', ...
    'MarkerSize',MarkerSize, ...
    'MarkerEdgeColor','k', ...
    'MarkerFaceColor',[1.0,0,0]);

% plot the cspace
plot(cspace(:,1), cspace(:,2), 'k.');

if (goalIsConnected)
    title(sprintf('RRT: %d nodes — Goal Id %d', r, goalId));
else
    title(sprintf('RRT: %d nodes', r))
end

xlabel('\omega [rad]')
ylabel('u [-]')
legend('Start', 'Nodes', 'Goal', 'Path', 'C_{obs}', 'Location', '...')
Listing 45: Function that plots the continuum RRT

clear; clc;

%% generic (for rapid testing ...)
% Start = [1.57, 1.76, 0];
Goal = [-1.57, -1.76, 0];
data = csvread('..\path\csv.csv', 1, 0);
plot_graph(9, data, Goal, 1);

%% Scenario 2 (which comes first)
% (a)
% Start = [0, 0, 0];
% Goal = [0, -1.47, 0];
data = csvread('..\path\arm2a.csv', 1, 0);
plot_graph(1, data, Goal, 2);
% (b)
% Start = [0, -1.47, 0];
% Goal = [0, -0.79, 0];
data = csvread('..\path\arm2b.csv', 1, 0);
plot_graph(2, data, Goal, 2);
%
%% Scenario 1 (which comes second)
% (a)
% Start = [0, -0.79, 0];
% Goal = [-1.57, -1.76, 0];
data = csvread('..\path\arm1a.csv', 1, 0);
Listing 46: Script that executes the continuum RRT visualization

Appendix F  Mobile Base Simulation Software

```matlab
function M = animateRRTpath(fig, scenario, data, Goal, dosave, fname)
if nargin < 6
    fname = 'rrtPath.mp4';
elseif nargin < 5
    dosave = false;
    fname = 'blank.mp4';
end
[r, c] = size(data);
idx_p = r;
r = r - 1;
figHandle = figure(fig);
frameCount = 1;
vertex = 1;
set(figHandle, 'units', 'inches', 'pos', [0 0 15 11.5]);
```
%% prep graph

```
cnv = 12*2.54;
xmax = 15*cnv; xmin = -1*cnv;
ymax = 11*cnv; ymin = -1*cnv;
id = data(:,1);
state.x = data(:,2);
state.y = data(:,3);
edges = zeros(r,c-4);
for i = 1:r
  for j = 1:c-4
    edges(i,j) = -1;
  end
end
for i=1:r
  j = 5;
  while (j < c && data(i,j) ~= 0)
    edges(i,j-4) = data(i,j);
    edges(i,j+1-4) = data(i,j+1);
    j = j + 2;
  end
end
if scenario == 1
  build_config_obstacles;
elseif scenario == 2
  build_config_obstacles2;
elseif scenario == 0
  build_config_obstaclesNone;
end
opt_path = data(idx_p,:);
```

%% boundary vertices

```
v = [ 0,0; 14,0; 14,10; 0,10 ]*12*2.54 + [1,1;-1,1;-1,-1;1,-1] * 35.56;
```
% loop
\textbf{for} \ i = 1: r
\hspace{1em} \textbf{figure} ( \text{figHandle} );
\hspace{1em} \text{xlim} ([ \text{xmin} \ \text{xmax} ]);  
\hspace{1em} \text{ylim} ([ \text{ymin} \ \text{ymax} ]);  
\hspace{1em} \text{hold on}
\hspace{1em} \textbf{end}
\hspace{1em} \textbf{for} \ u = 1: \text{numObs}
\hspace{2em} \text{fill} ( \text{Cobs}(u).vert(:,1), \text{Cobs}(u).vert(:,2), [1.0, 0.5, 0.5]);
\hspace{1em} \text{end}
\hspace{1em} \textbf{for} \ v = (1+ \text{numObs}):(1+ \text{numObs}*2)
\hspace{2em} \textbf{for} \ j = 1:4
\hspace{3em} x(:, :) = \text{Cobs}(v).line(j,1,:);
\hspace{3em} y(:, :) = \text{Cobs}(v).line(j,2,:);
\hspace{3em} \text{plot}(x, y, 'k--');
\hspace{2em} \text{end}
\hspace{1em} \textbf{end}
\hspace{1em} \textbf{end}
\hspace{1em} \textbf{end}
\hspace{1em} \textbf{for} \ j = 1:4
\hspace{2em} x(:, :) = \text{Cobs}(v+1).line(j,1,:);
\hspace{2em} y(:, :) = \text{Cobs}(v+1).line(j,2,:);
\hspace{2em} \text{plot}(x, y, 'k-', 'LineWidth', 2);
\hspace{1em} \textbf{end}
\hspace{1em} \textbf{end}
\hspace{1em} \textbf{end}
\hspace{1em} \textbf{end}
\hspace{1em} [e_\text{r}, ec] = \text{size}(\text{edges});
\hspace{1em} \textbf{for} \ m = 1: i
\hspace{2em} pt_1 = [ \text{state}.x(m), \text{state}.y(m)];
\hspace{2em} \textbf{for} \ n = 2:2:ec
\hspace{3em} \text{val} = \text{edges}(m, n);
if val == -1 && val <= i
    pt2 = [state.x(val+1), state.y(val+1)];
    plot([pt1(1), pt2(1)], [pt1(2), pt2(2)], 'k-');
end
end
end

% plot the start
plot(state.x(1), state.y(1), 'go', ...
     'MarkerSize', 8, ...
     'MarkerEdgeColor', 'k', ...
     'MarkerFaceColor', [0,1,0]);

% plot the rest up to this point
for j=2:i
    plot(state.x(j), state.y(j), 'bo', ...
         'MarkerSize', 8, ...
         'MarkerEdgeColor', 'k', ...
         'MarkerFaceColor', [0,0,1]);
end

% plot the goal
plot(Goal(1), Goal(2), 'go', ...
     'MarkerSize', 8, ...
     'MarkerEdgeColor', 'k', ...
     'MarkerFaceColor', [1,0,1]);

hold off
M(frameCount) = getframe;
clf(figHandle);
frameCount = frameCount + 1;
end

% plot path

figure(figHandle);

xlim([xmin xmax]);
ylim([ymin ymax]);

hold on

% plot obstacles

for u=1:numObs
    fill(Cobs(u).vert(:,1),Cobs(u).vert(:,2),[1.0,0.5,0.5]);
end

for v=(1+numObs):(1+numObs*2)
    for j=1:4
        x(:, :) = Cobs(v).line(j,1,:);
        y(:, :) = Cobs(v).line(j,2,:);
        plot(x,y,'k-');
    end
end

% plot boundary

for j=1:4
    x(:, :) = Cobs(v+1).line(j,1,:);
    y(:, :) = Cobs(v+1).line(j,2,:);
    plot(x,y,'k-','LineWidth',2);
end

% plot the edges

[er, ec] = size(edges);

for m = 1:r
    pt1 = [state.x(m), state.y(m)];
    for n = 2:2:ec
        val = edges(m,n);
        if val ~= -1 && val <= r
            pt2 = [state.x(val+1), state.y(val+1)];
        end
    end
end
plot([pt1(1), pt2(1)], [pt1(2), pt2(2)], 'k-');

end

end

% plot the rest up to this point
for j=2:i
    plot(state.x(j), state.y(j), 'bo', ...
         'MarkerSize',8, ...
         'MarkerEdgeColor','k', ...
         'MarkerFaceColor',[0,0,1]);
end

% plot the start
plot(state.x(1), state.y(1), 'go', ...
     'MarkerSize',8, ...
     'MarkerEdgeColor','k', ...
     'MarkerFaceColor',[0,0,1]);

% plot the goal
plot(Goal(1), Goal(2), 'go', ...
     'MarkerSize',8, ...
     'MarkerEdgeColor','k', ...
     'MarkerFaceColor',[1,0,1]);

for i=2:length(opt_path)-1
    plot(state.x(opt_path(i)+1), state.y(opt_path(i)+1), 'go', ...
         'MarkerSize',8, ...
         'MarkerEdgeColor','k', ...
         'MarkerFaceColor',[1,1,0]);
end
hold off
Listing 47: Function that animates the growth of the mobile base RRT

```matlab
numObs = 2;
t_cv = 12*2.54;
t_agent = 35.56;
Cobs(1).vert = [ 4,0;
             5,0;
             5,2;
             4,2 ]*t_cv;
v = Cobs(1).vert;
```
Cobs (1). line(1,:,:) = [ v(1,1), v(2,1); v(1,2), v(2,2) ];
Cobs (1). line(2,:,:) = [ v(2,1), v(3,1); v(2,2), v(3,2) ];
Cobs (1). line(3,:,:) = [ v(3,1), v(4,1); v(3,2), v(4,2) ];
Cobs (1). line(4,:,:) = [ v(4,1), v(1,1); v(4,2), v(1,2) ];

Cobs (2). vert = [ 7,6;
                  9,6;
                  9,10;
                  7,10 ]*t_cnv;

v = Cobs (2). vert;
Cobs (2). line(1,:,:) = [ v(1,1), v(2,1); v(1,2), v(2,2) ];
Cobs (2). line(2,:,:) = [ v(2,1), v(3,1); v(2,2), v(3,2) ];
Cobs (2). line(3,:,:) = [ v(3,1), v(4,1); v(3,2), v(4,2) ];
Cobs (2). line(4,:,:) = [ v(4,1), v(1,1); v(4,2), v(1,2) ];

Cobs (3). vert = Cobs (1). vert + [ -t_agent, 0; t_agent, 0; t_agent, t_agent; -t_agent, t_agent ];
v = Cobs (3). vert;
Cobs (3). line(1,:,:) = [ v(1,1), v(2,1); v(1,2), v(2,2) ];
Cobs (3). line(2,:,:) = [ v(2,1), v(3,1); v(2,2), v(3,2) ];
Cobs (3). line(3,:,:) = [ v(3,1), v(4,1); v(3,2), v(4,2) ];
Cobs (3). line(4,:,:) = [ v(4,1), v(1,1); v(4,2), v(1,2) ];

Cobs (4). vert = Cobs (2). vert + [ -t_agent, -t_agent; t_agent, -t_agent; t_agent, 0; -t_agent, 0 ];
v = Cobs (4). vert;
Cobs (4). line(1,:,:) = [ v(1,1), v(2,1); v(1,2), v(2,2) ];
Cobs (4). line(2,:,:) = [ v(2,1), v(3,1); v(2,2), v(3,2) ];
Cobs (4). line(3,:,:) = [ v(3,1), v(4,1); v(3,2), v(4,2) ];
Cobs (4). line(4,:,:) = [ v(4,1), v(1,1); v(4,2), v(1,2) ];
Listing 48: Script that "builds" the configuration obstacle structures

```matlab
function scenario = mobile_config(fname)
data = csvread(fname, 0, 1);
[r, c] = size(data);
scenario.x_lim = data(1,1:2);
scenario.y_lim = data(2,1:2);
scenario.agent_rad = data(3,1);
delta = scenario.agent_rad * [
    -1,-1,1,-1,1,1,-1; 
    -1,-1,1,-1,1,1,-1; 
    -1,-1,1,-1,1,1,-1; 
];
```

Listing 49: Function that loads the mobile base configuration file

```matlab
clc;
scenario = mobile_config('C:\Users\zhawks\Documents\c++\RRTplanner\-0\RRTplanner\_config.csv');
%Actions = csvread('action_list.csv');

data = csvread('..\mobilepath.csv', 1, 0);
scenario.startId = 2;
scenario.goallId = 3;
mobile_plot_graph(5, data, scenario);

data1 = csvread('..\mobilepath1.csv', 1, 0);
scenario.startId = 1;
scenario.goallId = 2;
mobile_plot_graph(1, data1, scenario);

data2 = csvread('..\mobilepath2.csv', 1, 0);
scenario.startId = 2;
scenario.goallId = 3;
mobile_plot_graph(2, data2, scenario);

data3 = csvread('..\mobilepath3.csv', 1, 0);
scenario.startId = 3;
scenario.goallId = 4;
```
Listing 50: Script that executes the mobile base RRT visualization

```matlab
function mobile_plot_graph(fig, data, S)
figure(fig);
goal = S.nodes(S.goalId,:);
goalIsConnected = false;
goalId = -1;
id = data(:,1);
state.x = data(:,2);
state.y = data(:,3);
state.th = data(:,4);
epsilon = 0.0001;

% determine edges
[r, c] = size(data);
idx.p = r;
r = r-1;
edges = zeros(r, c-4);
for i = 1:r
    for j = 1:c-4
        edges(i, j) = -1;
end
```
delta = \[ \text{abs}(\text{state}.x(i) - \text{goal}(1));\text{abs}(\text{state}.y(i) - \text{goal}(2));\text{abs}(\text{state}.\text{th}(i) - \text{goal}(3)) \];

\text{if} \ (\text{delta}(1) \leq \text{epsilon} \ \&\& \ ... \\
\quad \text{delta}(2) \leq \text{epsilon} \ \&\& \ ... \\
\quad \text{delta}(3) \leq \text{epsilon} ) \\
\quad \text{goalIsConnected} = \text{true}; \\
\quad \text{goalId} = \text{id}(i) + 1;
\text{end}
\text{end}

\text{opt}\_\text{path} = \text{data}((\text{id}x_p,:));

\text{for} \ i=1:r \\
\quad j = 5; \\
\quad \text{while} \ (j < c \ \&\& \ \text{data}(i,j) \neq 0) \\
\quad \quad \text{edges}(i,j-4) = \text{data}(i,j); \\
\quad \quad \text{edges}(i,j+1-4) = \text{data}(i,j+1); \\
\quad \quad j = j + 2; \ \\
\text{end}
\text{end}
\text{hold on}
\text{plot}(\text{state}.x(1), \text{state}.y(1), 'go', ...) \\
\quad \text{'MarkerSize'},8, ... \\
\quad \text{'MarkerEdgeColor'},'k', ... \\
\quad \text{'MarkerFaceColor'},[0,1.0,0]); \\
\text{plot}(\text{state}.x(2:r), \text{state}.y(2:r), 'bo', ...) \\
\quad \text{'MarkerSize'},8, ... \\
\quad \text{'MarkerEdgeColor'},'k', ... \\
\quad \text{'MarkerFaceColor'},[0.5,0.5,0.5]); \\
\text{plot}(\text{goal}(1), \text{goal}(2), 'ro', ...
'MarkerSize', 8, ...
'MarkerEdgeColor', 'k', ...
'MarkerFaceColor', [1.0, 0, 0]);
plot(goal(1), goal(2), 'ro', ...
'MarkerSize', 8, ...
'MarkerEdgeColor', 'k', ...
'MarkerFaceColor', [1.0, 1.0, 0]);

% plot Cobs
%patch('Faces', F, 'Vertices', V1, 'FaceColor', [0 0 1], 'EdgeColor', [0, 0, 0], 'LineStyle', '--', 'FaceAlpha', 0.4);
V = [S.obs(1,1), S.obs(1,2); S.obs(1,3), S.obs(1,4); S.obs(1,5), S.obs(1,6); S.obs(1,7), S.obs(1,8)];
patch('Faces',[1,2,3,4], 'Vertices', V, ...
'FaceColor', [0,0,1], 'EdgeColor',[0,0,0], 'LineStyle', '--', 'FaceAlpha', 0.6);
V = [S.cobs(1,1), S.cobs(1,2); S.cobs(1,3), S.cobs(1,4); S.cobs(1,5), S.cobs(1,6); S.cobs(1,7), S.cobs(1,8)];
patch('Faces',[1,2,3,4], 'Vertices', V, ...
'FaceColor', [0,0,1], 'EdgeColor',[0,0,0], 'LineStyle', '--', 'FaceAlpha', 0.2);

% plot boundary
x1 = S.x_lim(1); x2 = S.x_lim(2);
y1 = S.y_lim(1); y2 = S.y_lim(2);
patch('Faces',[1,2,3,4], 'Vertices', [x1,y1;x2,y1;x2,y2;x1,y2], ...
'FaceColor', [1,1,1], 'EdgeColor',[0,0,0], 'LineStyle', '--', 'FaceAlpha', 0.0);
x1 = S.x_lim(1); x2 = S.x_lim(2);
y1 = S.y_lim(1); y2 = S.y_lim(2);
rad = S.agent_rad;

patch('Faces', [1, 2, 3, 4], 'Vertices', [x1+rad,y1+rad;x2−rad,y1+rad;x2−rad,y2−rad;x1+rad,y2−rad], ...
      'FaceColor', [1,1,1], 'EdgeColor',[0,0,0], 'LineStyle', '−−', 'FaceAlpha', 0.0);

% plot edges
[er, ec] = size(edges);
for i = 1:er
    pt1 = [state.x(i),state.y(i)];
    for j = 2:2:ec
        val = edges(i,j);
        if val ~= −1 && val < r
            pt2 = [state.x(val+1),state.y(val+1)];
            plot([pt1(1), pt2(1), [pt1(2), pt2(2)], 'k−');
        end
    end
end

plot(state.x(2:r), state.y(2:r), 'bo', ...
      'MarkerSize',8, ...
      'MarkerEdgeColor','k', ...
      'MarkerFaceColor',[0.5,0.5,0.5]);

% plot path
for i=1:length(opt_path)
    plot(state.x(opt_path(i)+1), state.y(opt_path(i)+1), 'go', ...
         'MarkerSize',8, ...
         'MarkerEdgeColor','k', ...
         'MarkerFaceColor',[1.0,1.0,0.0]);
end
plot(state.x(1), state.y(1), 'go', ...
    'MarkerSize', 8, ...
    'MarkerEdgeColor', 'k', ...
    'MarkerFaceColor', [0,1,0]);

plot(goal(1), goal(2), 'ro', ...
    'MarkerSize', 8, ...
    'MarkerEdgeColor', 'k', ...
    'MarkerFaceColor', [1,0,0]);

% plot Cobs
for i = 1:4
    V = [S.obs(i,1), S.obs(i,2); S.obs(i,3), S.obs(i,4); S.obs(i,5), S.obs(i,6); S.obs(i,7), S.obs(i,8)];
    patch('Faces',[1,2,3,4], 'Vertices', V, ...
          'FaceColor', [0,0,1], 'EdgeColor',[0,0,0], 'LineStyle', '-', '
          FaceAlpha', 0.6);
    V = [S.cobs(i,1), S.cobs(i,2); S.cobs(i,3), S.cobs(i,4); S.cobs(i,5), S.cobs(i,6); S.cobs(i,7), S.cobs(i,8)];
    patch('Faces',[1,2,3,4], 'Vertices', V, ...
          'FaceColor', [0,0,1], 'EdgeColor',[0,0,0], 'LineStyle', '—', '
          FaceAlpha', 0.2);
end

if (goalIsConnected)
    title(sprintf('RRT: %d nodes − Goal Id %d', r, goalId));
else
    title(sprintf('RRT: %d nodes', r))
end

296
Listing 51: Function that plots the mobile base RRT

clear; clc;
cnv = 12*2.54;
goal = [1.5, 1.5]*cnv;
start = [11.0, 8.0]*cnv;
xmax = 15*cnv; xmin = -1*cnv;
ymax = 11*cnv; ymin = -1*cnv;

build_config_obstacles2;

figure(1);
hold on
plot(start(1), start(2), 'go', ...
    'MarkerSize', 8, ...
    'MarkerEdgeColor', 'k', ...
    'MarkerFaceColor', [0,1.0,0]);
plot(goal(1), goal(2), 'ro', ...
    'MarkerSize', 8, ...
    'MarkerEdgeColor', 'k', ...
    'MarkerFaceColor', [1.0,0,1.0]);
if (numObs > 0)
    fill(Cobs(1).vert(:,1),Cobs(1).vert(:,2),[1.0,0.5,0.5]);
end
for j=1:4
    x(:,:) = Cobs(1+numObs).line(j,1,:);
    y(:,:) = Cobs(1+numObs).line(j,2,:);
    plot(x,y,'k--');
end
% plot Cobs
for i=1:numObs
    fill(Cobs(i).vert(:,1),Cobs(i).vert(:,2),[1.0,0.5,0.5]);
end
for i=(1+numObs):(1+numObs*2)
    for j=1:4
        x(:,:) = Cobs(i).line(j,1,:);
        y(:,:) = Cobs(i).line(j,2,:);
        plot(x,y,'k--');
    end
end
% plot boundary
for j=1:4
    x(:,:) = Cobs(i+1).line(j,1,:);
    y(:,:) = Cobs(i+1).line(j,2,:);
    plot(x,y,'k--','LineWidth',2);
end
title(sprintf('Configuration Space'))
xlabel('X [cm]')
ylabel('Y [cm]')
legend('start','goal','Obs_t_a_s_k','Obs_c_a_s_p_a_c_e','Location',...
Listing 52: Script that plots the mobile base configuration space obstacles

clear; clc;
cnv = 12*2.54;
goal = [1.5, 1.5]*cnv;
start = [11.0, 8.0]*cnv;
xmax = 15*cnv; xmin = -1*cnv;
ymax = 11*cnv; ymin = -1*cnv;

build_config_obstacles2;

figure(1);
hold on
plot(start(1), start(2), 'go', ...
    'MarkerSize', 8, ...
    'MarkerEdgeColor', 'k', ...
    'MarkerFaceColor', [0, 1.0, 0]);
plot(goal(1), goal(2), 'ro', ...
    'MarkerSize', 8, ...
    'MarkerEdgeColor', 'k', ...
    'MarkerFaceColor', [1.0, 0, 1.0]);

% plot Cobs
for i=1:numObs
    fill(Cobs(i).vert(:,1), Cobs(i).vert(:,2), [1.0, 0.5, 0.5]);
end

for i=(1+numObs):numObs*2
    % for j=1:4
Listing 53: Script that plots the mobile base task space obstacles
```matlab
set(figHandle, 'units', 'inches', 'pos', [0 0 15 11.5]);

if scenario == 1
    build_config_obstacles;
elseif scenario == 2
    build_config_obstacles2;
elseif scenario == 0
    build_config_obstaclesNone;
end

cnv = 12*2.54;
xmax = 15*cnv; xmin = -1*cnv;
ymax = 11*cnv; ymin = -1*cnv;

figure(figHandle);
xlim([xmin xmax]);
ylim([ymin ymax]);
hold on
plotRobot(Start,[0,0,1]);
hold off
[r,c] = size(Trajectory);

for i=1:r
    figure(figHandle);
xlim([xmin xmax]);
ylim([ymin ymax]);

    hold on
    % plot obstacles
    for u=1:numObs
        fill(Cobs(u).vert(:,1),Cobs(u).vert(:,2),[1.0,0.5,0.5]);
    end
```
for v=(1+numObs):(1+numObs*2)
    for j=1:4
        x(:, :) = Cobs(v).line(j, 1,:);
        y(:, :) = Cobs(v).line(j, 2,:);
        plot(x, y, 'k-');
    end
end

% plot boundary
for j=1:4
    x(:, :) = Cobs(v+1).line(j, 1,:);
    y(:, :) = Cobs(v+1).line(j, 2,:);
    plot(x, y, 'k-', 'LineWidth', 2);
end
state = Trajectory(i, 2:4);
plotRobot(state,[0,0,1]);
plotRobot(Goal,[1,0,1]);

hold off
M(frameCount) = getframe;
clf(figHandle);
frameCount = frameCount + 1;
end

figure(figHandle)
xlim([xmin xmax]);
ylim([ymin ymax]);
xlabel('X [cm]')
ylabel('Y [cm]')
title('Robot Path')
movie(M, 1);
if dosave
    vid = VideoWriter(fname, 'MPEG-4');
    vid.FrameRate = 15;
    open(vid)
    writeVideo(vid,M)
    close(vid)
end
end

Listing 54: Function that simulates the mobile base executing the RRT output

clear; clc;
data = csvread('experiment_data.csv',1,0);
labels = {'seed', 'type', 'success', 'vlimit', 'goal_look', 'min_nodes', '
um_nodes', 'num_misses', 't_RRT', 't_A*', 'pathNodes', 'pathDist', '
thetaDist', 'thetaOpt'};
type{1} = data(1:20,:);
type{2} = data(21:40,:);
type{3} = data(41:60,:);

for i = 1:3
    d = type{i};
    mu(i,:) = mean(d);
    sig(i,:) = std(d);
end

Listing 55: Script that analyzes the statistics from the RRT experiments
Bibliography


