Development of a Novel, Tree-Like Branching Continuum Robot with Variable Topology

Michael Clayton Lastinger
Clemson University, m.lastinger@gmail.com

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

Recommended Citation
https://tigerprints.clemson.edu/all_theses/3113

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.
DEVELOPMENT OF A NOVEL, TREE-LIKE BRANCHING CONTINUUM ROBOT WITH VARIABLE TOPOLOGY

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
Michael Clayton Lastinger
May 2019

Accepted by:
Dr. Ian D. Walker, Committee Chair
Dr. Apoorva Kapadia
Dr. Richard Groff
Abstract

This thesis describes the design and physical realization of a novel branching continuum robot, aimed at inspection and cleaning operations in hard-to-reach environments at depths greater than human arm lengths. The design, based on a hybrid concentric-tube/tendon actuated continuum trunk core, features two pairs of fully retractable continuum branches. The retractable nature of the branches allows the robot to actively change its topology, allowing it to penetrate narrow openings and expand to adaptively engage complex environmental geometries. We detail and discuss the realization of a physical prototype of the design, and its testing in several simulated application environments.
Dedication

To Dad, Mom, and Evan for all the love and support throughout this journey.
Acknowledgments

I would first like to thank Dr. Ian Walker for giving me the opportunity to conduct research in the field of robotics, which I have always been very passionate about. His expertise, advise, patience, and dedication to my success truly made this a memorable experience and I could not have done it without him.

I would like to thank Dr. Apoorva Kapadia for all of the advice and guidance given to me throughout the course of my research and for serving on my committee.

I would also like to thank Dr. Richard Groff for serving as the final member of my thesis committee.

Special thanks go to Chase Frazelle, Michael Wooten, Phanideep Gonthina, Zach Hawks, and Gil Rutland for all of their ideas and assistance in my research in the lab.

I would also like to thank Mr. Tim Pruett for his assistance with rapid prototyping and Mr. John Sherard for his mentorship and guidance throughout my journey as a graduate student.

Finally, I would like to thank my family and friends for supporting me as I pursue my goals.
# Table of Contents

Title Page ................................................................. i  
Abstract ................................................................. ii  
Dedication ................................................................. iii  
Acknowledgments ......................................................... iv  
List of Figures ........................................................... vi  

1 Introduction .......................................................... 1  
  1.1 Continuum Robotics ............................................ 1  
  1.2 Concentric Tube and Branching Continuum Robots ........... 2  
  1.3 Thesis Overview ............................................... 4  

2 Design and Prototype Development ................................. 5  
  2.1 Overview of the TREE robot .................................. 5  
  2.2 Trunk Improvements .......................................... 6  
  2.3 Branch Design ................................................ 11  
  2.4 Branch Development and Integration ......................... 18  

3 Software Development and Control Issues ......................... 46  
  3.1 Real-Time Sensing and Software Issues ....................... 46  
  3.2 Kinematic Model Implementation and Modification for the Trunk 48  
  3.3 Integrating the Tendril Branch Actuation Packages .......... 53  

4 Baseline Performance Experiments ................................ 57  

5 Potential Applications and Demonstrations ....................... 66  
  5.1 Glove Box Cleaning ......................................... 66  
  5.2 Other Potential Applications ................................ 71  

6 Conclusions and Suggestions for Further Research ............... 76  
  6.1 Conclusions ................................................ 76  
  6.2 Suggestions for Further Research ............................ 77  

Appendices .............................................................. 80  
  A Arduino Code .................................................. 81  
  B C++/Arduino Library Files .................................... 117  
  C TREE Keyboard Controls ...................................... 156  

Bibliography ............................................................ 159
# List of Figures

2.1 Design Concept of Trunk Robot [1] ................................................. 6  
2.2 The Original Prototype of the TREE [1] ....................................... 7  
2.3 Rhino Motors with Spools ............................................................. 8  
2.4 Original Middle Collar Prototype [1] ............................................ 9  
2.5 Modified Middle Collar ............................................................... 9  
2.6 Rhino Motor Mount with “T” slot ................................................ 10  
2.7 Side View of the TREE Actuation Package - Middle Collar is Purple, Distal Collar is White ................................................................. 10  
2.8 Ultrasonic Distance Sensor with Wave Reflecting Panel .................... 11  
2.9 Middle Section “Cart” for Extension/Retraction .............................. 12  
2.10 Design Concept of Trunk Robot with Branches ................................ 12  
2.11 Inspiration for “elbows” found in nature – Angel Oak Tree (Photo courtesy of TripAdvisor [2]) .............................................................. 13  
2.12 Single “Elbow” Spacer ................................................................. 14  
2.13 Rear View of Support Branch Revealing Spring Backbone ................ 14  
2.14 “Elbow” Support Branch Relaxed .................................................. 15  
2.15 Closer Look at Elbow Support Branch ........................................... 15  
2.16 Demonstration of Dexterity of Support Branch ............................... 16  
2.17 Constant Curvature Verification of Support Branch .......................... 16  
2.18 Shape Formed by “Elbow” When Spacers Interlock ......................... 17  
2.19 First Prototypes of Tendril Branches ........................................... 18  
2.20 New Distal Section Spacer Design ................................................ 19  
2.21 New Distal Section Spacer Design with Double Cones ..................... 19  
2.22 Old (left) and New (right) Distal Sections ................................... 20  
2.23 Folded Branch Storage Tube During Installation ............................ 21  
2.24 Folded Branch Storage Tube During Installation (full view) ............. 22  
2.25 Distal Section with One Branch Storage Tube ............................... 22  
2.26 Expanding Previously Folded Storage Tube ................................... 23  
2.27 Shaping Ramp for Tendril Branch Deployment ............................... 24  
2.28 Side View of Ramp Revealing its Curvature ................................. 24  
2.29 Hole in Tube and Ramp for Tendril Deployment ............................ 25  
2.30 Hole in Tube and Ramp for Tendril Deployment (front view) .......... 25  
2.31 New Distal Section with Extended Tendril Branches (top view) ........ 26  
2.32 New Distal Section with Extended Tendril Branches ..................... 27  
2.33 New Distal Section with Extended Tendril Branches (side view) ...... 27  
2.34 New Distal Section with Fully Retracted Tendril Branches ............. 28  
2.35 Beginning to Align the Tendril Spacers ..................................... 29  
2.36 Aligning and Measuring Tendril Spacers ................................... 29  
2.37 Aligning Tendril Spacers (full view) .......................................... 30  
2.38 Tendril Branch with Spacers Before Tendons ................................ 30  
2.39 Tendril Branch with Tendons Routed ........................................ 31
2.40 Final Prototypes of Tendril Branches ........................................... 31
2.41 Tendril Branch on Top of Ramp ................................................... 32
2.42 Tendril Branch Deploying Using Ramp ....................................... 32
2.43 Distal Section with Tendril Branches Partially Deployed (Using Ramps) 33
2.44 Tendril Branch Actuation Assembly .......................................... 34
2.45 Linear Servo Assembly for Tendril Branch Extension/Retraction ....... 35
2.46 Modification of Linear Servo Carriage (Progressing from Right to Left) 35
2.47 Left Tendril Branch Actuation Package Assembled and Wired .......... 36
2.48 Tendril Branch Tendons Attached to Spools ................................ 36
2.49 Push/Pull Dowel for Tendril Branch - Branch Mounting End .......... 37
2.50 Push/Pull Dowel for Tendril Branch - Servo Carriage Mounting End . 38
2.51 Tendril Branch Alignment with Storage Tube via Linear Servo Assemblies 38
2.52 Top View of Tendon Spreader Aligning Tendons with Spools .......... 39
2.53 Spool Orientation and Tendon Routing ...................................... 39
2.54 Spreading Tendons and Spooling - Closer View of Alignment .......... 40
2.55 Rebuilding Distal Section to Accommodate for Branch Actuation Package 41
2.56 Shorter Middle Collar ............................................................... 41
2.57 Shorter Distal Collar ................................................................. 42
2.58 Tendril Branch Actuation Packages Fully Installed ....................... 42
2.59 Left Tendril Branch Bending Outward via 3rd and 4th (shorter) Tendons 43
2.60 Left Tendril Branch Bending Inward via 1st and 4th (shorter) Tendons 43
2.61 TREE with All Branches Extended ............................................. 45

3.1 Experimental Data Relating Encoder Counts to Tendon Lengths .......... 50
3.2 Planar Concept (RPR) [3] ............................................................... 52
3.3 Spatial Concept (RRPR) [3] ............................................................ 53
3.4 Original Trunk Prototype on 10 cm x 10 cm Grid ........................... 54
3.5 Original Trunk Prototype on 10 cm x 10 cm Grid (top view) .......... 54
3.6 Middle and Distal Sections in “S” Configuration ............................ 55

4.1 Original TREE Trunk Prototype Cleaning Tank ............................... 58
4.2 TREE Cleaning Tank Wall ............................................................ 58
4.3 Maximum Force Applied by End Effector While Cleaning Tank - Scale Reads 1417.6 g 59
4.4 TREE Scrubbing a Corner of the Tank ......................................... 59
4.5 TREE Equipped for Small-Scale Cleaning and Inspection with Tendril Branch 60
4.6 Simulated Glove Box Environment for Small-Scale Cleaning ............ 61
4.7 Example of a Glove Box [4] ............................................................ 61
4.8 Post Cleaning/Inspection Experiment ......................................... 62
4.9 Inspection Camera Deployed in Tendril Branch .............................. 62
4.10 Using a Tendril Branch for Inspection While Cleaning ................... 63
4.11 Compressed “Elbows” Providing Support for the TREE .................. 64
4.12 Compressed “Elbows” Providing Support for the TREE (front view) .... 64
4.13 Compressed “Elbows” Providing Support for the TREE (rear view) ....... 65

5.1 Dirty Simulated Glove Box Environment - Experimental Setup .......... 67
5.2 Distal Section and Tendril Branches Inside Dirty Simulated Glove Box Environment 68
5.3 Air Hose Inserted into the Distal Section ...................................... 68
5.4 Small Brush Attached to Tendril Branch for Intricate Cleaning .......... 69
5.5 Tendril Branch Performing Intricate Cleaning Task to Assist Trunk ....... 69
5.6 Tendril Branch Camera View of the Other Branch Cleaning ............... 70
5.7 NASA’s Mars Exploration Rover, Spirit, Before (left [5]) and After (right [6]) Dust Accumulated on Solar Panels. Images courtesy of NASA/JPL-Caltech/Cornell University. ................................................................. 71
5.8 NASA’s Mars Exploration Rover, Opportunity, Before (left) and After (right) Dust Was Blown off of Solar Panels by Wind [7]. Images courtesy of NASA/JPL-Caltech/Cornell Univ./Arizona State Univ. ................................................................. 72
5.9 Simulated Mars Solar Panel Dust Removal Experiment ........................................ 74
5.10 Tendril Branch Camera View of Simulated Mars Solar Panel Dust Removal Experiment 75
Chapter 1

Introduction

Robotics is currently an active and rapidly evolving discipline. Within robotics, the topic of continuum robots has emerged over the past few years to establish continuum robotics as a new and useful form of robot. This thesis introduces and details research, prototyping, and testing of a new type of continuum robot.

1.1 Continuum Robotics

Continuum robots are biologically inspired robots with smooth bodies [8] that often mimic snakes [9], elephant trunks [10], octopus arms [11], tongues, tails [12], worms, and vines [13]. They are often made of compliant materials to maximize bending capabilities. Their maneuverability differs from “conventional” rigid link robots in that they can bend at any point along their structure. It is important to note that the structure of a continuum robot is frequently referred to as the “backbone” despite their sometimes invertebrate nature [3]. While rigid-link robots, much like human arms, can only achieve finitely indexed configurations due to their finite number and location of joints (such as the human elbow, wrist, and shoulder), continuum robots have (theoretically)-infinite degrees of freedom (DOF). This feature allows them to take on a much wider range of configurations necessary for a given application or task at hand. Summaries of early developments in the field of continuum robotics have appeared in [3], [8], and [14].

Features such as a high level of backbone compliance and infinite backbone DOF allow continuum manipulators to be used for grasping irregularly-shaped objects using their entire backbone,
just as an elephant would with its trunk [8], [10]. However, conventional rigid-link robots can only manipulate objects using their specialized end effectors (grippers). This high level of dexterity allows continuum robots to adapt to the shape of an object being grasped. In addition, this dexterity can also allow them to avoid collisions with obstacles in complex environments. This ability is essential in navigating tight, hard-to-reach spaces that conventional robots could not. Therefore, continuum robots are also well-suited for inspection applications in congested spaces.

Most continuum robots developed so far have been at a relatively small scale (less than 1 m in length). This scale coupled with the high maneuverability make continuum robots an optimal choice for numerous medical applications [15], the area in which continuum robots have made the biggest impact to date.

1.2 Concentric Tube and Branching Continuum Robots

The medical field has been one of the main application arenas thus far for continuum robots [15]. Designed to assist in minimally-invasive surgery and diagnostics, these continuum robots are very small. One of the most common designs used in the medical field is the concentric tube design. These robots consist of a set of precurved concentric tubes that can rotate and extend with respect to each other, similar to the way a telescope extends [16], [17]. This design allows the shape of the robot to be controlled by merely rotating these tubes with respect to one another without relying on actuators within the body or tendons to guide it. The need to extend/retract and bend in a carefully engineered way to navigate the confined, often unreachable, spaces inside the human body is satisfied by this technology. Sears and Dupont present a steerable needle concentric tube design in [16] that addresses the need to steer a needle around bony/hard or sensitive (fluid-filled) structures during percutaneous procedures. They report that in order to minimize the damage to tissue during the procedure, it is advantageous for the robot to generate its own bending forces (achieved through the precurved tubes) instead of relying on the surrounding tissue environment. Such robots are designed for specific applications utilizing their dexterous, yet still limited, range of configurations. For surgical purposes, this is sufficient because the curves of the tubes are engineered for navigation in predetermined environments. Such environments may include arteries during intracardiac reconstructive surgery [16], umbilical cords during blood sampling [18], tissues of the brain and liver [18], [19], as well as cerebral ventricles within the brain [20] - in all of which the environmental topology
and constraints are already known.

Another design seen for continuum robots in the medical field are those with branching elements [21], [22]. These can prove advantageous in minimally invasive, single port access (SPA) surgeries where the need to operate multiple tools in a confined space presents challenges [23]. Since the goal is to minimize the size and number of incisions (entry points) on the body (hence the term minimally invasive), all of the branches containing tools must enter through one access location (single port access). Simaan et. al. present and evaluate an insertable robotic effector platform (IREP) which can be deployed through a 15 mm incision for SPA surgery [24], [25]. The IREP has two surgical arms and a third that manipulates a stereo-vision module that tracks the instrument location. To achieve simplicity of actuation, each backbone extension/retraction is coupled. This gives the ability of one arm to be pulled while the radially-opposing arm can be pushed by the same amount [25]. While this simplifies deployment and operation in the surgical field, it limits the overall dexterity and topology of the system. The IREP can also access up to a 50 mm Cartesian workspace, which is sufficient for delicate surgeries but does not scale well to larger applications.

While a majority of continuum robots developed have been significantly less that one meter in length, some larger scale varieties have been demonstrated. The OctArm is a multi-section, soft continuum robot developed for whole-arm grasping in open-air and underwater [11]. Constructed of McKibben pneumatic artificial muscles (PAM), the OctArm is approximately 83 cm long and can extend to a length of approximately 1.33 m. In [13] and [26] for example, thin continuum “Tendril” robots of length over one meter are detailed, designed to be deployed for in-space exploration of complex areas that are not easily accessible. The only example of a truly large-scale continuum robot to date is the EMMA\textsuperscript{T}\textsuperscript{M} manipulator [27], [28] which was a 52.5 ft (16 m) hyper-redundant manipulator aimed at waste tank remediation. EMMA\textsuperscript{T}\textsuperscript{M} used a tower system which lowered the entire 5-stage arm into the area of access and had a motor drive to enable it to rotate along the tower axis. However, this system was designed for a specific task and therefore is not easily generalized to other applications.

Given that the only domain in which branching (parallel) continuum robots have been developed/used is the medical field, the key motivation for the work described in this thesis was to expand this feature to larger scales, out of the human body (where the body structure provides support) and to a wider range of potential applications.
1.3 Thesis Overview

This thesis presents a novel design for a branching continuum robot and demonstrates the performance capabilities of a hardware prototype [29] of the design. The core body of the robot is a relatively large scale hybrid concentric-tube/tendon actuated continuum structure, referred to as a “trunk.” An original prototype of the trunk was restricted to the structural hardware and basic actuation of connected motors via three-position rocker switches [1]. The motors lacked encoders so no position feedback was used to control them. Provided with a mechanical proof of concept of this robot, the goal of the research reported in this thesis was to fully “robotize,” or automate, the system with closed-loop control and further expand its design and capabilities via adding retractable “branches” to the trunk. This required the development and testing of prototypes of the branches integrated with the trunk. The result is the first branching concentric tube continuum robot at a large scale (greater than 1 m).

The following chapter describes the improvements made to the original robot trunk, as well as the design and development of the prototypes of two sets of branches. Chapter 3 discusses the software and kinematic model implemented to control the system. Chapter 4 details the performance of the branches through basic experiments involving inspection and cleaning tasks. Chapter 5 describes several applications and supporting experiments with the system in which the further potential of such a novel type of robot is demonstrated. We demonstrate the robot performing representative tasks exploiting its branches, taking advantage of its variable topology and the ability to adapt to different environments. Chapter 6 presents conclusions drawn from the experiments performed and opportunities for further research.
Chapter 2

Design and Prototype Development

2.1 Overview of the TREE robot

The TREE (Tree Robot for Extended Environments) is a large-scale continuum robot (in comparison with medical continuum robots of comparable design) designed for applications in hard-to-reach environments. The core structure of the TREE is a “trunk” comprised of three concentric tube sections: the base (referred to as the proximal section), the middle, and the tip (referred to as the distal section). Each of the sections is actuated in part by two cables, or tendons, spaced $180^\circ$ apart. By changing the lengths of these tendons, the sections can bend within a 2-dimensional plane. The middle and distal sections are further remotely actuated to rotate $360^\circ$ about their central axis. This allows them to rotate the plane of bending and access a full 3D workspace. These two sections are also translated via linear actuators to extend and retract with respect to the other sections, allowing the robot to deploy in variable and deep environments (Figure 2.1). The TREE has a higher payload capacity than that of its thin continuum counterparts due to its size and the materials with which it is constructed. With a maximum reach of 1.8m (all sections fully extended), it is the first concentric tube robot, to the best of our knowledge, developed with a backbone length greater than 1 meter.
2.2 Trunk Improvements

As stated in Chapter 1, the original prototype of the TREE’s trunk, shown in Figure 2.2, had a very basic control mechanism [1]. The motors actuating the tendons did not have encoders and therefore no position feedback was available. The only method of control was a set of three-position rocker switches to change the direction of the motors (left or right for forward or backward) or make them stop (middle position). The control box is circled in red at the bottom of Figure 2.2, located under the robot. While this was a simple solution to moving the trunk, this control mechanism was designed merely for a mechanical proof of concept of the TREE.

The next logical step for the project was to upgrade the actuators to encoded motors. This is where the efforts discussed in this thesis began. We selected high torque DC servo motors from Rhino Motion Controls [30] to actuate the tendons on each of the three sections of the trunk, as well as handle the rotation of the middle and distal sections. These motors, shown in Figure 2.3, are 12 V, 10 RPM industrial grade motors that have metal gearboxes with high quality metal gears. Providing a torque of 120 kg-cm (1666.5 oz-in), they are strong enough to handle the significant forces required to actuate the tendons. Their relatively slow speed was also desirable for safety reasons due to the large size of the TREE. Another advantage of using these motors is the built-in motor drivers which reduced the external circuitry required. They also support multiple communication methods such as UART, I²C, PPM (pulse-position modulation), and analog signals which provide absolute (32-bit) motor position control and accurate speed control. In addition to having integrated 0.2° resolution optical encoders, the motors also have built-in closed loop PID control. The proportional (P), integral (I), and derivative (D) parameters came factory tuned, but users have the ability to modify and even reset the values back to factory settings. This feedback was an essential specification of the
new motors. For our application, we chose to use serial communication via UART to communicate with them.

On the original prototype of the TREE (Figure 2.2), the tendons actuating the middle and distal sections were routed through their respective “collars” before reaching the motors. These “collars” are parts made using additive manufacturing (3D printing) that attach the ends of the sections to their corresponding pillow block bearings and sprocket-chain mechanisms for rotation. The motors were mounted to the rear “wings” of the collars and the tendons exited the sides of the collars to spool onto the motors, as shown in Figure 2.4. Due to the exit angle of the tendons, the motors were mounted at an angle to allow the tendons to wind smoothly without catching on the lips of the spools [1]. However, an issue with this design was that the high torque required to actuate the tendons caused the cables to rub against the collars and fray. To solve this issue, the collars for each section were modified to allow the tendons to run completely through the collars and exit the rear of the parts. Since the large actuation torque of the tendons also caused the wings of the collars to flex, the mounting location of the motors was moved to the back side of the wings. Any tendencies of the motors to flex under high torque are hindered by the wings, as the motors were forced against them. New motor mounts were designed in SolidWorks [31] to fit the new Rhino
motors and attach to the new collars via a “T” slot attachment (Figures 2.5 and 2.6). This enables the motors to be easily removed if needed. The new collars and motor mounts were 3D printed and installed on the TREE as shown in Figure 2.7. Custom spools were designed in SolidWorks and 3D printed to provide secure, consistent places to store excess tendon cable while actuating the TREE. The spools are also shown in Figure 2.3.

Another improvement made to the original TREE was the addition of ultrasonic distance sensors to track the position of the linear actuators, which extend and retract the middle and distal sections. Without these sensors (shown at the top of Figure 2.8), there was no way for the TREE’s software to know the current positions of the middle and distal “carts.” The middle section cart is shown in Figure 2.9. Close monitoring by the operator of the TREE was previously essential to ensure these carts did not collide with the front or back of the robot’s frame, or each other. With the distance sensors providing the distance (in cm) of the middle and distal carts from the front and back of the robot respectively, software limits were implemented to prevent physical interference. Since the sensors use sonar to determine proximity/distance, we installed flat panels of aluminum-plastic composite material to give the waves a constant surface from which they could reflect back to the sensors. The panel for the middle section can be seen at the top of Figure 2.8 (and at the top of
Figure 2.4: Original Middle Collar Prototype [1]

Figure 2.5: Modified Middle Collar
Figure 2.6: Rhino Motor Mount with “T” slot

Figure 2.7: Side View of the TREE Actuation Package - Middle Collar is Purple, Distal Collar is White
2.3 Branch Design

The next step in the research was to provide the TREE trunk with branching capabilities. The design concept implemented for the branches features two pairs of branches, two “elbows” and two “feelers,” as shown in Figure 2.10 [29]. Each branch is individually actuated and fully retractable. The main purpose of the “elbows,” described further in the following paragraph, is to provide support for the trunk of the robot. Therefore, they are deployed from the proximal section of the trunk. The purpose of the “feelers” is to provide a means of closer inspection of the end effector (tip of the robot), as well as perform small scale manipulation. Therefore, they are deployed from the distal section. Each pair of branches is fully retractable, which allows the TREE to actively change its topology. The TREE is the first continuum robot to possess this capability to the best of our knowledge. In their fully retracted states, the branches allow the TREE to navigate narrow openings as a single continuum element - the trunk. Once through such openings, they can be extended, causing the robot to expand in and engage complex environments in novel and useful
The inspiration for the “elbows” comes in part from the way low hanging tree branches often grow towards the ground and use it for support as they continue to grow. An example of this behavior is found in the branches of the Angel Oak Tree on Johns Island, South Carolina, U.S.A. (Figure 2.11) [2]. In a horizontal orientation of the trunk, due to the flexibility of the materials with which it is built, its tendency is to sag due to gravity. The elbows were designed to stiffen and support the trunk as it extends. The elbows themselves were constructed of 3D printed spacers (diameter of 3.2 cm) with six teeth on top and bottom as shown in Figure 2.12. The tops of the
Figure 2.11: Inspiration for “elbows” found in nature – Angel Oak Tree (Photo courtesy of TripAdvisor [2])

spacers are male fittings and the bottoms are female to allow them to interlock. For the backbones, we selected 20 in. (0.51 m) long compression springs with outer diameters of 0.375 in. (0.0095 m). They have a deflection of 10.69 in. (0.272 m) under a maximum load of 5.63 lbs (2.55 kg). Each spacer has three tendon holes, 120° apart, and tapping through the center to allow them to be screwed onto the spring, as shown in Figures 2.13, 2.14, and 2.15. With compression springs as backbones, the branches remain dexterous as demonstrated in Figures 2.16 and 2.17. The spacers are designed in the shape of a wedge (one side is taller than the other). This causes the top faces to be inclined, forcing the branches to take the form of a predefined curve when fully compressed, creating an elbow shape as shown in Figure 2.18. To simplify the compression of the spring (stiffening of the elbow), we attached one end of a 1.5 mm diameter carbon fiber tube to the end cap of the branch and ran the tube down the center of the spring (Figure 2.13). This tube could also be replaced with a single cable/tendon to achieve a higher level of compliance before the spring is compressed. The branch is compressed by simply pulling this tube/tendon. As the springs compress, the elbows stiffen, and the spacers lock together to form a rigid structure which can support the weight of the trunk. This feature will be discussed further in Chapters 4 and 5.

13
Figure 2.12: Single "Elbow" Spacer

Figure 2.13: Rear View of Support Branch Revealing Spring Backbone
Figure 2.14: “Elbow” Support Branch Relaxed

Figure 2.15: Closer Look at Elbow Support Branch
Figure 2.16: Demonstration of Dexterity of Support Branch

Figure 2.17: Constant Curvature Verification of Support Branch
The “feelers” are adapted versions of the Tendril [13] developed previously by our group. The Tendril design is a thin concentric tube backbone actuated remotely by tendons. These branches use the same diameter carbon fiber tubes (1.5 mm) as the middle section of the Tendril and are actuated by four tendons. The tendons are routed through 3D printed spacers, each with a diameter of 8 mm, fixed at intervals of approximately 2 in. along the carbon fiber tubes. Three tendons are spaced 120° apart and terminated at end caps (diameter of 9 mm). The fourth is threaded between two of those tendons and terminated approximately 1 ft. (0.3 m) from the tip of each branch. This allows the branches to produce more complex configurations by creating a quasi second section. These vine-like continuum branches, shown in Figure 2.19, provide dexterous inspection capabilities when equipped with a camera, such as viewing the end effector from different angles. They can also perform cleaning tasks on a very small and delicate scale, which will be discussed in Chapters 4 and 5.
2.4 Branch Development and Integration

In order to integrate the Tendril branches into the distal section of the trunk, the goal was to find a suitable material in which to house the branches when retracted. The challenge was choosing a material that was flexible enough to not restrict the bending of the sections but sturdy enough to permit extension and retraction of the branches. We found thin-walled, chemical-resistant tubing with an inner diameter of $\frac{7}{16}$ in. (0.011 m) to be the optimal solution. We modified the design of the spacers for the distal section to allow this tube to run alongside the PEX tubing that forms the distal trunk section as shown in Figures 2.20 and 2.21. Since the 3D printed spacers required modification, we updated the design to include conical shapes (previously achieved by modified plastic funnels [1]) to improve structural integrity and aesthetics. We also replaced the original steel ball bearings, which reduced friction as the distal section extended/retracted inside of the middle section, with Delrin (plastic) ball bearings (Figure 2.20). This modification reduced the weight of the distal section and therefore the gravitational effects on the TREE as a whole. We constructed an entirely new distal section (Figure 2.22) to accommodate the necessary changes to incorporate the branches while keeping the original section intact for future experiments.

The holes incorporated into the new spacers to hold the branch storage tubes were designed...
Figure 2.20: New Distal Section Spacer Design

Figure 2.21: New Distal Section Spacer Design with Double Cones
Figure 2.22: Old (left) and New (right) Distal Sections
to have a very low tolerance. Therefore, it was difficult to install the storage tubes. As shown in Figure 2.23, the tubes had to be folded in half in order to easily slide into place. Keeping the tube folded down the full length of the distal section (Figure 2.24) required constantly working the fold, which encouraged it to maintain its shape. Tape was often used to keep the folds in place. Figure 2.25 shows the distal section with one storage tube installed. The disadvantage of folding the tubes during installation is they have to be expanded back to their original cylindrical shape to prevent any hindrance to the Tendril branches. To regain this form-factor, a \( \frac{3}{8} \) in. diameter wooden dowel rod was inserted into the tubes (Figure 2.26) which removed the folds.

To allow the Tendril branches to emerge from their housing, we cut a slot in the branch housing approximately 11 in. (0.28 m) from the tip of the distal section. We originally cut the slot on three sides, leaving the end closest to the tip of the robot intact. This created a flap that could be folded down into the slot to double as a ramp to help guide the branches in and out of the tubes during extension and retraction. However, this flap was not very stiff and not able to resist the force needed to extend each branch.

As a result, the difficulty of keeping the flap in place (even after gluing) presented the need for a better solution. We decided to create rigid ramps that would mount (glue) on top of the tube
Figure 2.24: Folded Branch Storage Tube During Installation (full view)

Figure 2.25: Distal Section with One Branch Storage Tube
flaps. The material for the ramps needed to be thin enough so that a lip would not be present where the ramp met the bottom of the tube. Such a lip would create a place where the Tendril branches would catch as they extended. The material selected was $\frac{1}{32}$ in. thick acrylic. This sheet of acrylic was thin enough to mold into a desired shape using a heat gun, as well as create a minimum lip inside the tube. To create the ramps, small rectangles approximately $\frac{3}{4}$ in. x $\frac{1}{2}$ in. were cut out of the sheet. A $\frac{3}{8}$ in. diameter wooden dowel rod about 5 in. long was used to create a mold for shaping the ramps. Starting approximately 1.25 in. from one end of the dowel, that same end was rounded gradually to the outside edge using a grinding wheel. Then, using vise grips, the small piece of acrylic was centered and secured over the rounded end of the dowel. Using a heat gun, the plastic was heated enough to mold it around the dowel to form the desired shape of a ramp, as shown in Figures 2.27 and 2.28 (although the earlier versions depicted here are longer than the final ramps that were installed). The edges of the ramps were sanded down to create a tapered effect and reduce the likelihood that the Tendril branches would get caught on the ramp edges. The ramps were installed by gluing them to the tops of the previously cut flaps as shown in Figures 2.29 and 2.30.

With the new distal section built, the branch storage tubes in place, and the ramps installed,
Figure 2.27: Shaping Ramp for Tendril Branch Deployment

Figure 2.28: Side View of Ramp Revealing its Curvature
Figure 2.29: Hole in Tube and Ramp for Tendril Deployment

Figure 2.30: Hole in Tube and Ramp for Tendril Deployment (front view)
the next step was to verify the smooth extension and retraction of the branches. Using the Tendril branches shown in Figure 2.19, we tested the branch actuation by hand. Testing was done with the distal section still removed from the robot, as well as with it partially reinstalled. Figures 2.31, 2.32, and 2.33 show the branches extended, and Figure 2.34 shows the branches fully retracted. After some experimentation, it became evident that the Tendril branches had a tendency to get caught on the lips (tops of the entry/exit slots) of the distal section spacers when retracting. This issue needed to be resolved before introducing any form of motorized actuation for these branches. We decided to design conical spacers for the Tendril branches, similar to the ones on the newly built distal section. Designed for the same purposes of smooth extension and retraction of the distal (and middle) sections of the TREE, we felt confident enough in this potential solution that we moved forward with the construction of two new Tendril branches.

Since most of the branches remain inside of the storage tubes at all times, the conical spacers were only necessary at the distal portion of the branches that would protrude from the trunk when extended. Therefore, the new branches were constructed with 12 conical spacers at the distal end and the rest (approximately 22) are the normal flat, disc-like spacers. As with the Tendril in [13] and the first versions of the branches, the spacers are JB Welded onto the carbon fiber tubes. As
Figure 2.32: New Distal Section with Extended Tendril Branches

Figure 2.33: New Distal Section with Extended Tendril Branches (side view)
previously mentioned in Section 2.3, the spacers are placed approximately 2 in. apart. The JB Weld allows a reasonable amount of work time since it sets in 4-6 hours and cures in 15-24 hours. However, care must be taken to ensure the spacers are evenly spaced and aligned (shown in Figures 2.35, 2.36, and 2.37) throughout the construction process and before the JB Weld sets. Once all of the spacers were placed on the carbon fiber tube, a single tendon threaded down the length of the branch and fixed to a rigid surface and/or stationary object at one end was used to ensure all of the spacers remain aligned (Figure 2.35). Figure 2.38 shows one of the branches before the tendons were strung. After the JB Weld fully cured, the tendons could be strung (Figure 2.39).

With the construction of two new Tendril branches complete (Figure 2.40), we tested the branch extension and retraction by hand as before. Figures 2.41 and 2.42 show the extension of the branches from the distal section of the trunk while it is still removed from the main robot. Figure 2.43 shows the branches partially deployed when the distal section is installed on the TREE. They moved on and off the ramps very smoothly. Also, further testing proved that the conical spacers did help guide the branches back into the storage tubes during retraction.

Each branch is actuated by four tendons, as described in Section 2.3, and a linear servo assembly (Figure 2.44) for extension and retraction. These assemblies (one for each Tendril branch)
Figure 2.35: Beginning to Align the Tendril Spacers

Figure 2.36: Aligning and Measuring Tendril Spacers
Figure 2.37: Aligning Tendril Spacers (full view)

Figure 2.38: Tendril Branch with Spacers Before Tendons
Figure 2.39: Tendril Branch with Tendons Routed

Figure 2.40: Final Prototypes of Tendril Branches
Figure 2.41: Tendril Branch on Top of Ramp

Figure 2.42: Tendril Branch Deploying Using Ramp
are each mounted on either side of the distal section PEX tubing (Figure 2.45). The core of the linear servo assembly is a “Single Perpendicular 785 Gear Rack Kit” from ServoCity [32]. This kit consists of a servo with a metal gear attachment, a plastic gear rack, an aluminum support beam, a servo carriage assembly that helps the servo maintain contact with the gear rack, and the necessary nuts and bolts for assembly. The kit comes with a multi-rotation Hitec HS-785HB servo which allows for up to 9.6 in. of travel. Through preliminary testing, we discovered that the servo could potentially travel farther than the gear rack kit allowed. The nuts and bolts holding it together seemed to prevent it from traveling off either end of the gear rack. However, in order to maximize the reach of the Tendril branches, we wanted to maximize the servos’ distance of travel as much as possible. According to the Hitec servo specs, when provided the maximum PWM signal it can handle, the servo can complete approximately 8 full revolutions (actually 7.85). With a gear circumference of 1.57 in. the maximum distance of travel for the servos is 12.32 in. To achieve this larger travel distance than the stock configuration, we had to make the gear rack assembly longer. To do so, we cut one extra gear rack and an extra aluminum support beam in half and bolted them onto a full length gear rack assembly. This created an assembly that was 1.5 times the length of the stock parts (so approximately 18 in. long). However, because the larger assembly required bolts in the middle
to hold the two pieces together, the servo would not be able to travel past those bolts. Therefore, the acrylic pieces of the servo carriage designed to secure the servo to the gear rack, were modified. The slots into which the gear rack assembly slide were modified as shown in Figure 2.46. The progression of the modifications flows from right to left. After this modification, the servos were able to travel over/past the nuts and bolts to achieve a maximum travel distance.

The motors chosen to actuate the tendons for the Tendril branches are Pololu 12V DC 20 mm diameter metal gearmotors [33]. At 12V, these motors run at 29 RPM and provide a torque of 70 oz-in (5 kg-cm), which is limited to this value by the gearbox. These motors were ordered with extended shafts on which we installed Pololu magnetic encoder kits. The high torque and small form-factor of these motors were desirable features in the tendon actuators. A motor package was designed using SolidWorks to hold four motors, one for each tendon per branch, and mount to the servo carriage of the linear servo assembly as shown in Figure 2.47. Two of these motor packages were 3D-printed (one for each branch). Spools on which to store the tendons were also designed in SolidWorks. The spools were designed in two halves that snap together. The holes for the screws used to attach the spools to the motor hubs run down the center so that they are not visible. The tendons are tied to one of the screws on the inside of the spool before it is snapped together and screwed onto the hub. The tendon sits in a groove notched out of the inner part of the spool as shown in Figure 2.48.

With the linear actuation of the servo providing the extension and retraction for the branches and the motor package providing tendon actuation, the only other design challenge left to solve was how to attach the backbone of the Tendril to the servo carriage (and therefore the tendon
Figure 2.45: Linear Servo Assembly for Tendril Branch Extension/Retraction

Figure 2.46: Modification of Linear Servo Carriage (Progressing from Right to Left)
Figure 2.47: Left Tendril Branch Actuation Package Assembled and Wired

Figure 2.48: Tendril Branch Tendons Attached to Spools
Figure 2.49: Push/Pull Dowel for Tendril Branch - Branch Mounting End

motor package). A simple solution which really worked well was a $\frac{3}{8}$ in. wooden dowel rod cut to approximately 3 in. long. On one end of the dowel, a small hole was drilled into the center just large enough to fit the end of the carbon fiber backbone of the Tendril branch inside as shown in Figure 2.49. On the other end, an off-centered hole was drilled and a bolt with its head sawed off was glued into that hole using JB Weld (Figure 2.50). The hole was off-centered to ensure the Tendril branch would be aligned with the entrance to the branch storage tube (Figure 2.51) as well as to prevent the dowel from catching on one of the bolts on the gear rack assembly. A modified bolt was secured in the other end to allow the dowel to be screwed into the side of the servo carriage, causing the Tendril branch to be connected to the actuation package for extension and retraction. Since the tendons would be routed from their small, 8 mm diameter spacers directly to their respective spools, we determined that there should be a tendon spreading piece which slides over the dowel and not only acts as an intermediate piece between the tendons (Tendril end) and the spools, but also helps line up the tendons with the spools as shown in Figures 2.52, 2.53, and 2.54.

Before the Tendril branch actuation packages could be mounted, the distal section of the trunk had to be modified slightly. As shown in Figure 2.55, several of the distal section spacers needed to be removed before the assemblies could be installed. Each actuation package attaches
Figure 2.50: Push/Pull Dowel for Tendril Branch - Servo Carriage Mounting End

Figure 2.51: Tendril Branch Alignment with Storage Tube via Linear Servo Assemblies
Figure 2.52: Top View of Tendon Spreader Aligning Tendons with Spools

Figure 2.53: Spool Orientation and Tendon Routing
to a modified version of the distal section spacers. The only difference is small notches under each storage tube hole into which the gear rack assemblies rest and securely bolt. Also, because of the large length of the middle collar, the travel distance for the servos would be limited. Therefore, we decided to 3D print new, shorter collars for the middle (Figure 2.56) and distal (Figure 2.57) sections of the TREE. We gained 3.25 in. of workspace between the middle and distal “carts” (for linear actuation) by shortening the middle collar and we gained 3.1 in. of retraction distance for the distal section by shortening its collar.

Through careful testing and experimentation with PWM ranges for the linear servos, these branch actuation assemblies provide a maximum extension of approximately 11.25 in. (0.286 m) for each Tendril branch from their fully retracted positions. Also, the tendon motor packages securely travel with their respective servo carriage. They manage to keep the entire actuation assemblies from interfering with the distal section tendon which runs between them as shown in Figure 2.58. They allow each tendon to be independently actuated to achieve dexterous bending of the Tendril branches as shown in Figures 2.59 and 2.60.

Similar accommodations were made to integrate the support branches (elbows) into the proximal section of the trunk. The same chemical-resistant tubing, with a larger inner diameter of
Figure 2.55: Rebuilding Distal Section to Accommodate for Branch Actuation Package

Figure 2.56: Shorter Middle Collar
Figure 2.57: Shorter Distal Collar

Figure 2.58: Tendril Branch Actuation Packages Fully Installed
Figure 2.59: Left Tendril Branch Bending Outward via 3rd and 4th (shorter) Tendons

Figure 2.60: Left Tendril Branch Bending Inward via 1st and 4th (shorter) Tendons
1.5 in. (0.04 m) was used to store the branches. We attached them to the bottom of the base section spacers, one offset to the left of center and the other to the right as shown at the top of Figure 2.61.
Figure 2.61: TREE with All Branches Extended
Chapter 3

Software Development and Control

Issues

In Chapter 2, the improvements to the original TREE trunk were detailed to involve the motor upgrades to the Rhino DC Servo motors. The introduction of closed-loop position feedback to the system made it possible to begin designing a user interface for the TREE. This chapter details those software efforts.

3.1 Real-Time Sensing and Software Issues

When controlling the trunk, the Rhino motors provide the ability to command it to move to a certain configuration via specific encoder counts (motor positions) sent to the motors. The encoder counts are always zero when the system is first powered on. So if the operator wants to “reset” the TREE to its original position/configuration, all that has to be done is send commands for all of the motors to go to count zero. This behavior works as expected as long as the robot remains powered on until it returns back to its original state. If for any reason, the emergency stop (e-stop) button has to be pressed killing all power to the system, the TREE remains in its current configuration. However, when power is restored to the system, the encoder counts are back to reading zero even though the robot is in a “non-zero” configuration.

To improve this, we needed to be able to save the current state of the TREE at regular
intervals and even on command. The memory needed to be non-volatile and persistent so that information would not be lost when the robot is powered off. After investigating options such as using an SD card shield to save encoder counts to a file on a SD card, we decided to take advantage of the on-board EEPROM (Electrically Erasable Programmable Read-Only Memory) memory on the Arduino Mega, which is used to control the trunk. The Mega has about 4 KB of EEPROM storage on-board which was sufficient for our purposes. Arduino has an open-source EEPROM library that made interfacing with the memory a matter of executing “read,” “write,” or “update” commands.

There are several important notes to make about the Arduino Mega EEPROM. First, each memory address can only store 1 byte. Because there are negative encoder counts that are used frequently, the necessity to store negative numbers shifted the range of storable values from [0 to 255] to [-128 to 127]. To be able to save larger encoder values, they would need to be stored over multiple addresses (like upper byte and lower byte), or be scaled down before saving them to memory. We decided to go with the latter. The encoders of the Rhino motors read 1800 counts per revolution [30]. Due to the construction of the TREE trunk, the assumption that no motor should ever need to travel greater than one revolution was a safe one. Therefore, assuming a maximum possible encoder count of 1800 that may need to be stored, we decided to scale the encoder positions down by a factor of 15 before saving them to EEPROM. For example, if the count to be stored was 1800, dividing this value by 15 first yields 120, which is a value that can be stored within 1 byte of memory. It falls in the range described above. Another important note is that the Arduino EEPROM has a finite life. It is only specified to handle approximately 100,000 write/erase cycles for each memory location. The number of reads from each location are unlimited. To extend the life of the EEPROM, we chose to use the update() function instead of write() when storing values in memory. Doing so only allows the board to write to memory if the value to be stored is different than the one currently stored in that location. It would also be beneficial to shift the chosen memory locations periodically to ensure the 100,000 cycle life expectancy is not reached at an early date during the life of the TREE. In the control code for the TREE, the encoder values are scheduled to be updated in EEPROM approximately every minute. However, a command was also implemented that allows saving to be manually triggered when necessary.
3.2 Kinematic Model Implementation and Modification for the Trunk

The ability to store motor (encoder) positions was crucial before beginning to implement any form of kinematics model for the TREE. Forward kinematics herein is a mathematical model-based approach to take encoder counts and tendon lengths in the actuator space and represent them in terms of the shape of the robot via a homogeneous transformation matrix. The forward kinematics model we implemented on the TREE is the Jones Model, described in [34]. Jones uses a modified version of a method called the Denavit-Hartenberg convention, which is commonly used to implement a forward kinematic model for rigid-link robots, to simulate the kinematics of continuum robots. The shape parameters used in the Jones Model are defined as arc length ($s$), curvature ($k$), and rotation out of the plane ($\phi$).

For the TREE, because it is a multi-section robot with the configuration of each section defined by a separate ($s, k, \phi$), a homogeneous transformation matrix must be formed for each of the sections. The final transformation matrix describing the shape/configuration of the entire trunk is the result of those three section matrices multiplied together. On the TREE, $s$ is straightforward to calculate. By definition, the arc length would be the length of the continuum element. Therefore, we define $s$ for each section of the trunk as the bendable length. The bendable length of the proximal section is simply its physical length. The length of the middle section is measured from the point it emerges from the proximal section tube. It is important to remember that due to the linear actuator attached to the middle section, it has the ability to extend or retract and change its length. This no longer poses a problem to sense due to the addition of the ultrasonic distance sensors which keep track of how much (and in which direction) the middle section moves. $s$ for the middle section just must be updated based on the current distance reading. A similar procedure is used to describe the distal section of the TREE. However, one unique characteristic that we did not initially consider is that as the middle section changes length, the effective length of the distal section changes regardless of whether or not is has been actuated. Therefore, we could not merely determine the length of the tip section based on the reading from the distance sensor.
The curvature, $k$, of each section is defined in [34] as

$$k = \frac{(l_1 - l_2)}{d(l_1 + l_2)} \quad (3.1)$$

where $l_1$ and $l_2$ are the lengths of the tendons in cm, and $d$ is the radius of the section (tube) in cm. Therefore, in order to calculate the curvature, we needed a way to relate changes in encoder counts to changes in lengths. We considered trying to predict this change by using the inner radius (to determine circumference) of the spools to calculate the changes in length based on how much the motor was turned. We ultimately decided against this due to the non-uniform way in which the tendons wind onto the spools. It would be difficult to accurately represent this unpredictable behavior, and performing the calculations under the assumption that the winding is uniform would produce inaccurate results as well. Therefore, we conducted an experiment in which the change in length of a tendon was recorded for every 100 counts of rotation of one of the Rhino motors. The results, shown in Figure 3.1, reveal that this relationship is nonlinear. We generated a best fit quadratic, shown as the dotted line in Figure 3.1, to approximate the function that represents the data.

The equation for the best fit quadratic is

$$y = -6E-07x^2 + 0.0034x + 0.068 \quad (3.2)$$

where $x$ is the encoder counts and $y$ is the change in length, $\Delta l$.

The shape parameter, $\phi$ is also straightforward to determine for each of the sections of the TREE. The angle of rotation out of the x-y (left-right) plane can be determined using the current encoder counts of the rotational motors for the middle and distal sections. The proximal section cannot rotate. To determine the relation between revolutions of the motors and revolutions of the actual trunk sections, the gear reduction ratios had to be calculated. The gears attached to the motor shafts of the middle and distal sections have 11 teeth. The sprocket on the 3D printed middle section collar has 45 teeth. Therefore the gear reduction ratio can be calculated as follows:

Middle section gear reduction ratio =

$$\frac{45}{11} = 4.09 : 1 \quad (3.3)$$
Figure 3.1: Experimental Data Relating Encoder Counts to Tendon Lengths
This means that in order for the middle section tube to complete 1 full revolution, the motor (attached to the sprocket via a chain) must complete 4.09 revolutions. With the motor encoders having 1800 counts per revolution (CPR), the total number of encoder/motor counts required for 1 revolution of the middle section is

$$4.09 \times 1800 \text{ counts} = 7362 \text{ counts} \quad (3.4)$$

Similarly, the distal section’s sprocket has 24 teeth. Therefore, the gear reduction ratio of the distal section can be calculated as follows:

Distal section gear reduction ratio =

$$\frac{24}{11} = 2.18 : 1 \quad (3.5)$$

As with the middle section, this means that in order for the distal section tube to complete 1 full revolution, the rotational motor must complete 2.18 revolutions. Since this motor’s encoder also has 1800 CPR, the total number of counts required for 1 revolution of the distal section is

$$2.18 \times 1800 \text{ counts} = 3924 \text{ counts} \quad (3.6)$$

Finally, $\phi$ must be expressed in radians using the fact that each motor turns 0.2 degrees per count (since there are 1800 counts per revolution). With this parameter, degrees can be converted to radians. It is also important to note that the convention in the field of robotics is a counterclockwise (CCW) rotation is represented using a positive angle. Therefore, $\phi$ is positive for a CCW rotation.

Since the base section cannot rotate, we decided to use the Planar Model [3] shown in Figure 3.2 to best approximate the kinematics for this section. This planar concept approximates the kinematics of a continuum robot based on that of a planar RPR (revolute-prismatic-revolute) rigid-link robot. Using this model and the shape parameters defined above, the homogeneous transformation
matrix of the proximal section of the TREE is

\[
[H^0_{3}]_{\text{proximal}} = \begin{bmatrix}
\cos(sk) & -\sin(sk) & 0 & \frac{1}{k}(\cos(sk) - 1) \\
\sin(sk) & \cos(sk) & 0 & \frac{1}{k} \sin(sk) \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

(3.7)

The middle and distal sections can rotate out of the x-y plane, so we elected to use the Spatial Model [3] shown in Figure 3.3 to best approximate the kinematics for these sections. The spatial model approximates the kinematics of a continuum robot based on that of a spatial RRPR (revolute-revolute-prismatic-revolute) rigid-link robot. Using this model and the \(s, k, \phi\) shape parameters defined above, the homogeneous transformation matrices of the middle and tip sections are

\[
[H^0_{4}]_{\text{middle}} = [H^0_{4}]_{\text{distal}} = \begin{bmatrix}
\cos(\phi) \cos(sk) & -\sin(\phi) & -\cos(\phi) \sin(sk) & \frac{1}{k}(\cos(\phi) - \cos(\phi) \cos(sk)) \\
\sin(\phi) \cos(sk) & \cos(\phi) & -\sin(\phi) \sin(sk) & \frac{-1}{k} \sin(\phi) - \sin(\phi) \cos(sk)) \\
\sin(sk) & 0 & \cos(sk) & \frac{1}{k} \sin(sk) \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

(3.8)

The upper left 3x3 matrix of Equations 3.7 and 3.8 is the Rotation Matrix describing the shape/configuration of that particular section of the trunk. The upper right 3x1 matrix is the
Translation Vector, which describes the \( x, y, z \) coordinates of the end effector (i.e. the end of that section). To calculate the overall homogeneous transformation of the entire TREE, the three matrices given in Equations 3.7 and 3.8 are multiplied together as mentioned previously. The resulting transformation matrix for the TREE is

\[
[H]_{TREE} = [H]_{proximal}^0 \ast [H]_{middle}^0 \ast [H]_{distal}^0
\] (3.9)

As with the section matrices, the upper left 3x3 matrix of Equation 3.9 is the overall Rotation Matrix describing the shape/configuration of the entire TREE trunk. The upper right 3x1 matrix is the overall Translation Vector, which describes the \( x, y, z \) coordinates of the end effector (i.e. the tip of the TREE). Validation experiments of the kinematic model were conducted on top of a 10 cm x 10 cm grid as shown in Figures 3.4, 3.5, and 3.6.

### 3.3 Integrating the Tendril Branch Actuation Packages

The control circuitry of the TREE had to be modified extensively to integrate the Tendril branch actuation packages. Since the encoders for the Pololu branch tendon actuation motors are
Figure 3.4: Original Trunk Prototype on 10 cm x 10 cm Grid

Figure 3.5: Original Trunk Prototype on 10 cm x 10 cm Grid (top view)
traditional quadrature encoders, they require two interrupt pins on the Arduino to detect the edges of the signals on each channel (A and B). The Arduino Mega only has 6 interrupt pins and the branch actuation motors required at least 16 interrupt pins. Due to the need for more interrupt pins and the fact that a majority of the digital pins on the Mega were already being used by the trunk, we decided to control the branches using an Arduino Due. To ensure the functionality of the robot was that of one cohesive system (trunk and branches), we set up the Due to communicate serially with the Mega. The Mega was established as the master device and the Due as the slave. This way, all of the control commands and position/configuration feedback for the trunk and branches could be handled via the Mega’s Serial Monitor alone (Serial Monitor must be set to “Insert Newline Character”). A simple preceding character “>” informs the Mega that the following command is to be sent to the Due.

One important hardware modification that was required stemmed from the fact that the Mega uses 5V logic while the Due uses 3.3V logic. Therefore, the TX (transmit) signal from the Mega had to be stepped down to 3.3V before being connected to the RX (receive) pin of the Due. A simple voltage divider circuit consisting of $R_1 = 1 \, k\Omega$ and $R_2 = 2 \, k\Omega$ was used to step the voltage down to 3.3V.
The results of the above effort enabled closed-loop control of the TREE. In the following Chapters, we detail and discuss experiments conducted with the robot.
Chapter 4

Baseline Performance Experiments

In this Chapter, we describe experiments conducted to evaluate the baseline performance of the TREE. In tests prior to the work reported in this thesis, static load tests of the trunk had been performed, which established the static load capacity of its three sections as 11 lb (5 kg) for the proximal section, 8 lb (3.6 kg) for the middle section, and 2.5 lb (1.13 kg) for the distal section [1]. However, no tests of force capability during operations had been conducted.

We therefore conducted experiments to demonstrate the capability of the system to apply force with the tip during novel cleaning tasks in a simulated environment. An empty aquarium, of dimensions 48 in. x 18 in. x 20 in. (122 cm x 46 cm x 51 cm), raised approximately 18 in. (46 cm) above the floor was used for the cleaning environment. The robot was positioned to enter the aquarium from the top as shown in Figures 4.1 and 4.2. The first task was to maneuver the cleaning tool in a back-and-forth, sweeping motion along the bottom of the tank. Once this was achieved, the goal was to repeat this motion while extending the distal section to apply pressure and simulate a scrubbing action. We measured the maximum force applied by the cleaning tool to be 3.13 lbf (13.9 N), without restricting the maneuverability of the robot (Figure 4.3).

After the successful scrubbing demonstration in an easily accessible location, we proceeded to perform the more difficult task of scrubbing an edge and corner of the tank, as shown in Figure 4.4. The cleaning tool was maneuvered by the robot along the bottom edge of the aquarium toward the corner, pressed into the corner, and used a sweeping motion to remove the collected debris. While the goal of this experiment was to demonstrate the dexterity of the robot, the scrubbing experiments explore its pushing/pressing capabilities while performing tasks such as cleaning.
Figure 4.1: Original TREE Trunk Prototype Cleaning Tank

Figure 4.2: TREE Cleaning Tank Wall
Figure 4.3: Maximum Force Applied by End Effector While Cleaning Tank - Scale Reads 1417.6 g

Figure 4.4: TREE Scrubbing a Corner of the Tank
Additional experiments were conducted to demonstrate the capability of the overall integrated system in novel inspection and cleaning tasks as described in [29]. We attached a small paint brush to the tip of the robot (Figure 4.5) to use for cleaning the walls of the empty aquarium (Figure 4.6), serving as a surrogate for a glove box wall. A glove box, shown in Figure 4.7, is used for handling and experimenting with radioactive materials or objects previously exposed to radiation as described in [4]. While the aquarium was dusty, we used a dry erase marker to make highly visible marks to clean as shown in Figure 4.8. Due to the small size of the brush and the large size of the TREE, it would be difficult to guide the brush to a desired location. Therefore, the goal of the experiment was to use an Enable, Inc. minnieScope –XS [35], 1.4 mm diameter camera deployed at the tip of one of the Tendril branches (shown in Figure 4.9) to help us guide the tip and perform close inspection of the environment (aquarium). Using this camera, we were able to guide the paint brush to the marks and clean them. The left half of Figure 4.10 shows the tip of the robot “holding” the paint brush, as well as the Tendril branches (one of which contains the camera). The view of the inspection camera housed by the branch is shown in the right half of Figure 4.10.

We also tested the supporting ability of the elbows. To verify their effectiveness, we tested their stiffening capabilities under the weight of the TREE itself as reported in [29]. With both
Figure 4.6: Simulated Glove Box Environment for Small-Scale Cleaning

Figure 4.7: Example of a Glove Box [4]
Figure 4.8: Post Cleaning/Inspection Experiment

Figure 4.9: Inspection Camera Deployed in Tendril Branch
the middle and distal sections approximately halfway extended, we took measurements with and without using the support elbows. We measured the distance between the end of the base section and the shelf over which the robot was to reach to be $4 \frac{7}{8}$ in. (12.38 cm). With the elbows extended and stiffened as shown in Figures 4.11, 4.12, and 4.13, the distance between the robot and the shelf was $5 \frac{2}{8}$ in. (13.65 cm). From this, we could conclude that the elbows improved the robot support by approximately 10.3%. The deployed elbows made the system visibly more stable than when they were not deployed.
Figure 4.11: Compressed “Elbows” Providing Support for the TREE

Figure 4.12: Compressed “Elbows” Providing Support for the TREE (front view)
Figure 4.13: Compressed “Elbows” Providing Support for the TREE (rear view)
Chapter 5

Potential Applications and Demonstrations

In this Chapter, we discuss several novel potential applications for the TREE. We empirically simulated several of these potential applications with the TREE in which the unique advantages of such a novel type of robot is demonstrated. The robot was required to perform representative tasks exploiting its branches, taking advantage of its variable topology and the ability to adapt to different environments.

5.1 Glove Box Cleaning

The first experiment was an adaptation to the previous cleaning environment application described in Chapter 4. The experimental setup is shown in Figure 5.1. The goal was to simulate a glove box environment after it had been used for radioactive experiments/analysis (as described in [4]) where there is debris left behind. The simulated environment is shown in Figure 5.2. For this experiment, an air tube was deployed (Figure 5.3) through the distal section’s hollow PEX tubing to allow air from a compressor to be directed and controlled, using the tip of the TREE, to blow away debris. The same 1.4 mm diameter camera used in Chapter 4, the Enable, Inc. minnieScope –XS, was deployed at the tip of the left Tendril branch. A small brush was attached to the tip of the right Tendril branch, as shown in Figure 5.4, to assist the trunk with the clean up of hard to move
debris. While the main trunk performed large-scale blowing/dusting tasks, the Tendril branches worked together to inspect the work and touch up missed particles with more intricate movements. The sweeping accomplished by the Tendril branch is shown from a side view in Figure 5.5. The perspective of the camera deployed through the inspection branch is shown in Figure 5.6. The sweeping motion begins in the top left corner with (a) and progresses to (d). The Tendril branch is circled in each frame and the path it has left behind is traced by the red dotted arrow. The arrow also points in the direction of motion at that particular instant.

This experiment required the TREE to coordinate extension, rotation, and bending of the trunk distal section, along with extension and bending of both Tendril branches. Each of these three structures provide a different functionality (non-contact manipulation, contact manipulation, and local sensing). This demonstrates the versatility of the robot, providing a functionality which is unique to the best of our knowledge.
Figure 5.2: Distal Section and Tendril Branches Inside Dirty Simulated Glove Box Environment

Figure 5.3: Air Hose Inserted into the Distal Section
Figure 5.4: Small Brush Attached to Tendril Branch for Intricate Cleaning

Figure 5.5: Tendril Branch Performing Intricate Cleaning Task to Assist Trunk
Figure 5.6: Tendril Branch Camera View of the Other Branch Cleaning
5.2 Other Potential Applications

There are a wide range of potential application fields for this novel design of a branching, concentric tube continuum robot. For example, consider applications like described above where the trunk is blowing air/dusting and the two Tendril branches are assisting through inspection and delicate manipulation. One field that could find such robots useful is the aerospace industry (notably NASA). NASA relies on solar panels to power satellites, rovers, and other equipment while in space. In locations like Mars, various atmospheric events like wind, storms, etc. can cause dust to build up on the solar panels hindering their ability to collect energy as detailed in [36]. Figure 5.7 shows an example of such dust accumulation on NASA’s Mars Exploration Rover, Spirit. The left image in Figure 5.7 is a panoramic self-portrait of Spirit taken on August 27, 2005 [5]. It shows the solar panels with only a thin layer of dust two years after the rover landed and began exploring Mars. The right image of Spirit was assembled from frames taken between October 26-29, 2007 [6]. It shows the significant amount of dust that had accumulated on the solar panels of the rover, which reduced its ability to collect energy from the Sun.

Sometimes, NASA gets lucky and the same windstorms that deposit dust on the rovers’ solar panels blow the accumulated dust off of them (such occurrences are referred to by NASA as “cleaning events”). This happened during the Mars Exploration Rover Opportunity’s mission [7]. The left image in Figure 5.8 shows a thick layer of accumulated dust on Opportunity in January 2014. The right image shows the solar panels on the rover after a “cleaning event” in late March.
2014, which resulted in a power level increase of 70% when compared to previous power levels that year [7]. Such “cleaning events” actually extended the missions of Spirit and Opportunity years beyond their initial three-month duration expectancies.

Possible solutions to the dust accumulation problem have been explored by several groups ([36], [37]). To avoid modifying the rovers themselves, NASA has explored alternate mission approaches. In [36], Landis et. al. at NASA detail the exploration of shortening missions to 30-sols (Martian days) as opposed to 90-sols, restricting the rover’s daily driving to periods during which more solar power is available, and launching the rover a few months earlier to take advantage of more favorable conditions (“seasons”) on Mars. In [37], Mazumder et. al. describe the development of transparent flexible dust shields to minimize obscuration of solar panels (on Mars and Earth) from the accumulation of dust. The dust shields were designed to repel dust particles via an electrostatic charge, enabling solar panels to perform self-cleaning operations. However, they suggest such self-cleaning solar panels equipped with these dust shields could be manufactured to derive their power (for the electrostatic charge) from the solar panels themselves. Depending on the thickness of the accumulated dust, the available power could prove to be insufficient.

A robot like the TREE could help clean off the solar panels in this scenario. In the movie, *The Martian* [38], Ridley Scott et. al. show Mark Watney (played by Matt Damon) flipping over and blowing off the solar panels that were overturned by the storm that ultimately left him stranded on Mars. If NASA had a robot like the TREE mounted on the Mars rover, astronauts or engineers
could routinely clean the solar panels (potentially remotely). To simulate this type of environment, we modified the glove box environment described in Section 5.1 (Figure 5.2). Using a small solar panel covered in red clay particles, we created an environment as shown in Figure 5.9. As shown in image (a) in the top left corner of Figure 5.9, the experiment begins with the solar panel covered by a significant layer of “Mars dust” (red clay). The TREE then performs actions similar to those in the glove box particle cleaning experiment. The distal section moves up and down slowly while air blows through it to remove the dust from the solar panel. Then it rotates clockwise 90 degrees as shown in Figure 5.9(b) to enable it to move left and right as it blows more dust away. After a few full sweeping motions, the left (now top) Tendril branch, through which the inspection camera is deployed, extends and bends downward towards the solar panel (Figure 5.9(c)). This allows the operator of the TREE to see how much dust the distal section left behind. The viewpoints from the Tendril branch camera during this experiment are shown in Figure 5.10 and correspond in sequence to the letters in Figure 5.9. Finally, the right (now bottom) Tendril branch extends and uses the small brush attached to its tip to sweep away the remaining dust particles from the solar panel, as shown in Figure 5.9(d). This process can be automated or controlled by the TREE’s operator.

As was the case with the glove box experiment in Section 5.1, this solar panel cleaning experiment utilized all degrees of freedom of the distal section of the TREE trunk, as well as the two Tendril branches. This further demonstrates the versatility of the robot while illustrating a different potential application.

Another field in which the features of blowing and inspecting could be useful is underwater archeology. Scientists use remotely operated vehicles (ROVs) to explore the ocean floor at depths that exceed safe levels for human divers. A system like the TREE integrated into an ROV could assist scientists in gently uncovering buried debris (or potentially treasure) from shipwrecks. At the time of writing, we are using the TREE to conduct further experiments to empirically simulate these applications.
Figure 5.9: Simulated Mars Solar Panel Dust Removal Experiment
Figure 5.10: Tendril Branch Camera View of Simulated Mars Solar Panel Dust Removal Experiment
Chapter 6

Conclusions and Suggestions for Further Research

In this thesis, we have introduced a novel design for a branching continuum robot and demonstrated the performance capabilities of a hardware prototype of the design. The core body of the robot is a relatively large scale hybrid concentric-tube/tendon actuated continuum structure, which we refer to as a “trunk.” An original prototype of the trunk had been restricted to the structural hardware and basic actuation of connected motors via three-position rocker switches. The motors lacked encoders so no position feedback was used to control them. Provided with a mechanical proof of concept of this robot, the goal of the research reported in this thesis was to fully “robotize” the system with closed-loop control and further expand its design and capabilities via adding retractable “branches” to the trunk. This required the design, development, and testing of prototypes of the branches integrated with the trunk. The result is the first branching concentric tube continuum robot at a large scale (greater than 1 m).

6.1 Conclusions

This thesis detailed the improvements made to previously developed robot trunk hardware, as well as the design and development of the prototypes of two sets of branches designed to retract into the trunk (Chapter 2). Chapter 3 discussed the software and kinematic model implemented to
control the system. We also detailed the control issues encountered during development. Chapter 4 detailed the performance of the branches through basic experiments involving inspection and cleaning tasks. Chapter 5 described several possible applications and supporting experiments with the system in which the further potential of such a novel type of robot is demonstrated. We demonstrated the robot performing representative tasks exploiting its branches, taking advantage of its variable topology and the ability to adapt to different environments. We detailed two experiments in which a tube was deployed through the distal section’s hollow PEX tubing to allow compressed air to be directed and controlled using the tip of the TREE. While the main trunk performed large-scale blowing/dusting tasks, the Tendril branches worked together to inspect the work and touch up missed particles with more intricate movements.

The work in this thesis demonstrates that a large-scale branching continuum robot can achieve a level of dexterity exceeding that of existing continuum robots, many of which are designed for medical applications. At this larger scale, dexterity is a higher priority than the accuracy required by smaller continuum robots used in minimally invasive surgeries. This novel design also has a longer reach and a payload capacity higher than that of thin continuum counterparts at the multi-meter level.

6.2 Suggestions for Further Research

The results from the experiments and testing performed on this prototype suggest several improvements that could be made on the robot. One significant improvement would be the addition of tension sensors on each tendon, for the trunk as well as the branches. Sensing and maintaining adequate tension is one of the most challenging aspects of tendon actuated continuum robots. Currently, when performing tasks with the TREE, the operator must keep a close eye (and at times feel with their hands also) the level of tension in the tendons currently being actuated. There are two reasons to do so: 1) to ensure the tendons are not put under too much stress which would cause them (or the plastic spacers to which they are attached) to break/fail 2) to avoid slack formation in the tendons which may cause them to unravel from their spools and hinder actuation of the robot. The introduction of tension sensors and a consequent slack avoidance algorithm would cause the TREE operation to be more robust.

Another modification which would allow the Tendril branches to extend and retract more
smoothly would be the addition of the conical spacers at the back of the branches to help guide them into their respective storage tubes. We made this modification to the front sections of the branches to guide their retraction into the tubes. However, it was not until we started conducting experiments that we realized the conical spacers were also needed at the back to guide the branches during extension. When rebuilding the Tendril branches and their actuation packages, we had confidence that the dowel attachment would be sufficient in aligning the branches with their storage tubes. While our hypothesis was generally correct, we did not account for the flexing that would occur when the branch tendons are actuated. This flexing causes the carbon fiber tubes to bow out and increase the probability that the flat, traditional spacers will catch on the lip of the tube entrance while the branches are extending. Also, the addition of position/distance tracking of the current location of the Tendril branch actuation packages (similar to what was done with the ultrasonic distance sensors for the middle and distal section carts) would help prevent collisions. Depending on the positions of the middle section cart and the branch actuation packages, collisions can occur if the TREE’s operator is not closely monitoring their configuration.

The decision to begin saving the trunk’s motor positions (encoder counts) to the EEPROM on the Arduino Mega proved to be advantageous. This greatly improved the controllability of the TREE through the implementation of the forward kinematics. It also serves as a back-up of the last known state in case the robot loses power or the emergency stop (e-stop) has to be pressed for any reason. Another improvement to the TREE would be the implementation of this same concept with the tendon motors of the Tendril branches. However, since these motors are currently controlled with the Arduino Due, which does not have on-board EEPROM (that is easily accessible), the previous states of the motors would have to either be stored on the Mega or as a file on an SD card.

Implementation of the forward and inverse kinematics for the branches, especially the Tendril branches, would also be a significant improvement to the TREE. However, this could prove to be non-trivial due to the high length/diameter ratio of the branches (causing torsion) and the restriction on their movements by the storage tubes. Also on the topic of kinematics, implementing the inverse kinematics on the trunk of the TREE would improve the controllability of the system even further. This would provide the ability to give the TREE a specific configuration or desired end effector position and the model would convert that configuration into encoder counts that inform the motor controllers how to achieve it.

The support branch (“elbow”) actuation could be improved by adding a motor to wind/un-
wind the central tendon of each branch. A strong, yet still easily compressible spring could be installed in each support branch housing tube to assist in the extension and retraction of the branches. As the central tendons wind, these springs would compress and allow the branches to fully retract into their tubes. Then to extend the branches, the central tendons would simply need to unwind and the compressed springs inside the branch housing tubes would push the branches out as they decompress. Finally, a servo motor (or two) could be mounted at the end of the TREE’s proximal (base) section to actuate a rigid flap(s) that would help the branches as they lock (stiffen). While the branches are extending, the rigid flaps (attached to the servos) would be positioned such that they would not hinder the branches. Once extended, the servos would move the rigid flaps into place, which would be between the final (back) spacer of each branch and the proximal section spacer which holds the branch housing tube. With these rigid flaps in place, the “elbows” would have a rigid surface against which they can compress. Then to retract the branches again, they would be relaxed (spring backbones decompressed) and the flaps would return to their original positions out of the way of the branches.

To further demonstrate the versatility of the TREE and illustrate another potential application in which it could be used, some underwater experiments should be conducted. As described in Chapter 5, a system like the TREE could be used by scientists in the field of underwater archeology. It could be integrated into a remotely operated vehicle (ROV) and assist scientists in gently uncovering buried debris (or potentially treasure) from shipwrecks. A simulated underwater environment could be constructed and further experiments conducted to empirically simulate this application.

Finally, designing a better user interface for the TREE would improve its usability. Either a graphical user interface (GUI) or a physical controller such as a joystick or Xbox controller could provide a more intuitive means of operation. The implementations and additions mentioned above would all be productive (and possibly necessary) steps toward the achievement of an intuitive control interface for the TREE.
Appendices
Appendix A  Arduino Code

/*
  * This program runs on the Arduino Mega (Master Device). It controls the Rhino
  * Motors for
  * the TREE trunk and communicates with the Due (Slave Device) to control the TREE
  * branches.
  * Created by Michael Lastinger
  *
  * This sketch sends commands to the motors and gets continuous feedback.
  *
  * Connect the TX pin coming out of the motor(orange wire) to pin _- (acting as RX)
  * on Arduino.
  * Connect the RX pin coming out of the motor(yellow wire) to pin _- (acting as TX)
  * on Arduino.
  *
  * NOTE: Not all pins on the Mega/Mega 2560 support change interrupts,
  * so only the following can be used for RX:
  * 10, 11, 12, 13, 50, 51, 52, 53, 62, 63, 64, 65, 66, 67, 68, 69
  *
  * Remember to give common ground connection to the motors/power supply and the
  * Arduino.
  */
#include <SoftwareSerial.h>
#include <Rhino.h>
#include <EEPROM.h>
#include <MatrixMath.h>

/*
  * Section: p = proximal section; m = middle; d = distal
  *
  * Motor Location: up = upper/top; low = lower/bottom; rot = rotational
  *
  * PWMS/DIR are for the linear actuators
  */
Rhino p_left(13,5);  //RX( -> TX of motor), TX( -> RX of motor)
Rhino p_right(12,8);
Rhino m_left(51,49);
Rhino m_right(52,46);
Rhino m_rot(53,47);
Rhino d_left(50,44);
Rhino d_right(11,7);
rhino d_rot(10,6);

#define d_pwm 30
#define d_dir 31
#define m_pwm 32
#define m_dir 33
#define pwm_val 150

#define d_echoPin 2 //Distal Dist. Sensor Echo Pin
#define d_trigPin 4 //Distal Trigger Pin
#define m_echoPin 22 //Middle Dist. Sensor Echo Pin
#define m_trigPin 24 //Middle Trigger Pin

long d_duration, d_distance, m_duration, m_distance; //Duration used to calculate distance

const int d_minDist = 16; //min distance of distal "cart" from sensor (max depends on mid.)
const int m_minDist = 3; //min distance of middle "cart" from sensor
const int m_maxDist = 34; //max distance of middle "cart"
int distanceToTravel, d_startPos, m_startPos;

String rawMsg = "";
String cmdLocal = "";
String cmdSent = ""; // Serial Communication with Due
String cmdRecv = ""; // Serial Communication with Due
char motor_select = ' ';
int reverseDir = 0; //cases 5 & 8

int saveTimer = 0;
int pos_1, pos_2, pos_3, pos_4, pos_5, pos_6, pos_7, pos_8 = 0;
int savedPos_1, savedPos_2, savedPos_3, savedPos_4, savedPos_5, savedPos_6,
    savedPos_7, savedPos_8 = 0;
int8_t value_1, value_2, value_3, value_4, value_5, value_6, value_7, value_8;
bool motorInit = true;

int proxLeftCnt, proxRightCnt = 0;
int midLeftCnt, midRightCnt, midRotCnt = 0;
int distalLeftCnt, distalRightCnt, distalRotCnt = 0;
float proxTendonLen1 = 55.25; //72.07cm total length to motor
float proxTendonLen2 = proxTendonLen1;
float midTendonLen1 = 12.4; // 140.97 cm total length to motor
float distalTendonLen1 = 39.4; // 246.38 cm total length to motor
float midTendonLen2 = midTendonLen1;
float distalTendonLen2 = distalTendonLen1;
deltaProxLen1, deltaProxLen2 = 0;
deltaMidLen1, deltaMidLen2 = 0;
deltaDistalLen1, deltaDistalLen2 = 0;
deltaPhi_m, deltaPhi_d = 0;
phi_p, phi_m, phi_d = 0;
s_p = 55.25; // constant value (cm)
s_m, s_d = 0; // cm
s_mInit = 12.4; // cm
s_dInit = 39.4; // cm

define N 4
float P[N][N];
float M[N][N];
float D[N][N];
float A[N][N];
float H[N][N];

void setup()
{
  Serial2.begin(9600); // Mega to Due Communication
delay(300);
p_left.printOutput(1); // enables Serial printing of function outputs
p_left.init(); // initializes the motor and sets current position as origin
delay(300); // allows each motor time to initialize
p_right.printOutput(1);
p_right.init();
delay(300);
m_left.printOutput(1);
m_left.init();
delay(300);
m_right.printOutput(1);
m_right.init();
delay(300);
m_rot.printOutput(1);
m_rot.init();
delay(300);
d_left.printOutput(1);
d_left.init();
delay(300);
d_right.printOutput(1);
d_right.init();
delay(300);
d_rot.printOutput(1);
d_rot.init();
delay(300);
pinMode(d_pwm,OUTPUT);
pinMode(d_dir,OUTPUT);
pinMode(m_pwm,OUTPUT);
pinMode(m_dir,OUTPUT);
pinMode(d_trigPin, OUTPUT);
pinMode(d_echoPin, INPUT);
pinMode(m_trigPin, OUTPUT);
pinMode(m_echoPin, INPUT);

Serial.println("=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=Commands");
Serial.println("Format: motorNumber command+EncoderCount");
Serial.println("ex) 1 G500");

Serial.println("  Motor Selection  Direction —> Sign of Encoder Val");
Serial.println(" 1 = Base Section, Left Motor   Wind: + ; Unwind: -");
Serial.println(" 2 = Base Section, Right Motor  Wind: - ; Unwind: +");
Serial.println(" 3 = Mid Section, Left Motor   Wind: + ; Unwind: -");
Serial.println(" 4 = Mid Section, Right Motor  Wind: - ; Unwind: +");
Serial.println(" 5 = Mid Section, Rotate   CW: - ; CCW: +");
Serial.println(" 6 = Tip Section, Left Motor  Wind: + ; Unwind: -");
Serial.println(" 7 = Tip Section, Right Motor  Wind: - ; Unwind: +");
Serial.println(" 8 = Tip Section, Rotate   CW: - ; CCW: +");
Serial.println("9 = Mid Section, Linear Actuator Forward: +; Backward: -");
Serial.println("0 = Tip Section, Linear Actuator Forward: +; Backward: -");
Serial.println("G = Go to specified encoder position");
Serial.println("R = Move Relative to current position, specify counts");
Serial.println("P = Get current encoder Position");
Serial.println("Y = Reset the motor");

//Reset the motor positions to previous values/states (stored on Arduino)
readFromEEPROM();

// Direction Methods for Linear Actuators
void fwd(int pwmPin, int pwmVal, int dirPin)
{
analogWrite(pwmPin, pwmVal);
digitalWrite(dirPin, HIGH);
//Serial.println("fwd");
}

void bwd(int pwmPin, int pwmVal, int dirPin)
{
analogWrite(pwmPin, pwmVal);
digitalWrite(dirPin, LOW);
//Serial.println("bwd");
}

void stp(int pwmPin, int dirPin)
{
analogWrite(pwmPin, LOW);
digitalWrite(dirPin, HIGH);
}

void get_dDist()
{
//Calculate the distance (in cm) of distal section travel
digitalWrite(d_trigPin, LOW);
delayMicroseconds(2);
digitalWrite(d_trigPin, HIGH);
delayMicroseconds(10);
digitalWrite(d_trigPin, LOW);
d_duration = pulseIn(d_echoPin, HIGH);
}
void get_mDist() {
  // Calculate the distance (in cm) of middle section travel
  digitalWrite(m_trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(m_trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(m_trigPin, LOW);
  m_duration = pulseIn(m_echoPin, HIGH);
  m_distance = m_duration / 58.2; // Conversion to cm
  delay(50);
}

void saveToEEPROM() {
  /* Save current encoder values to EEPROM.
   * 1 Byte per address in EEPROM – Need to save negative numbers
   * so range is actually −128 to 127.
   * Assume max = 1800 —> 1800/15 = 120 —> so divide by 15*/
  pos1 = p_left.getPos() / 15; delay(100);
  pos2 = p_right.getPos() / 15; delay(100);
  pos3 = m_left.getPos() / 15; delay(100);
  pos4 = m_right.getPos() / 15; delay(100);
  pos5 = m_rot.getPos() / 15; delay(100);
  pos6 = d_left.getPos() / 15; delay(100);
  pos7 = d_right.getPos() / 15; delay(100);
  pos8 = d_rot.getPos() / 15; delay(100);
  Serial.println("Saving: ");
  Serial.println(pos1);
  Serial.println(pos2);
  Serial.println(pos3);
  Serial.println(pos4);
  Serial.println(pos5);
  Serial.println(pos6);
  Serial.println(pos7);
Serial.println(pos_8);

/* "Update" = Value only gets written to memory if it is
 * different than the currently stored value*/
delay(200);
EEPROM.update(0, pos_1);
delay(200);
EEPROM.update(1, pos_2);
delay(200);
EEPROM.update(2, pos_3);
delay(200);
EEPROM.update(3, pos_4);
delay(200);
EEPROM.update(4, pos_5);
delay(200);
EEPROM.update(5, pos_6);
delay(200);
EEPROM.update(6, pos_7);
delay(200);
EEPROM.update(7, pos_8);
delay(200);
}

void readFromEEPROM(){
    // read a byte from each address of EEPROM
    delay(100);
    value_1 = EEPROM.read(0);
delay(100);
    value_2 = EEPROM.read(1);
delay(100);
    value_3 = EEPROM.read(2);
delay(100);
    value_4 = EEPROM.read(3);
delay(100);
    value_5 = EEPROM.read(4);
delay(100);
    value_6 = EEPROM.read(5);
delay(100);
    value_7 = EEPROM.read(6);
delay(100);
value_8 = EEPROM.read(7);
delay(100);

// Convert stored values back to encoder counts
savedPos_1 = (int)value_1*15;
savedPos_2 = (int)value_2*15;
savedPos_3 = (int)value_3*15;
savedPos_4 = (int)value_4*15;
savedPos_5 = (int)value_5*15;
savedPos_6 = (int)value_6*15;
savedPos_7 = (int)value_7*15;
savedPos_8 = (int)value_8*15;

Serial.println("Converted Values: ");
Serial.println(savedPos_1);
Serial.println(savedPos_2);
Serial.println(savedPos_3);
Serial.println(savedPos_4);
Serial.println(savedPos_5);
Serial.println(savedPos_6);
Serial.println(savedPos_7);
Serial.println(savedPos_8);

if (motorInit) {
  // Only executed during setup()
  // Set each motor position to previously stored value/state
  p_left.setPos(savedPos_1);
p_right.setPos(savedPos_2);
m_left.setPos(savedPos_3);
m_right.setPos(savedPos_4);
m_rot.setPos(savedPos_5);
d_left.setPos(savedPos_6);
d_right.setPos(savedPos_7);
d_rot.setPos(savedPos_8);
  motorInit = false;
}
}

void fwdKin() { // lengths in cm
Serial.println("FWD KINEMATICS");

readFromEEPROM(); // Get the saved encoder counts from Arduino

proxLeftCnt = p_left.getPos() - savedPos_1; // current pos - prev. pos
proxRightCnt = p_right.getPos() - savedPos_2;
midLeftCnt = m_left.getPos() - savedPos_3;
midRightCnt = m_right.getPos() - savedPos_4;
midRotCnt = (m_rot.getPos() - savedPos_5) * (-1); // negative to make sure CCW ---> +
distalLeftCnt = d_left.getPos() - savedPos_6;
distalRightCnt = d_right.getPos() - savedPos_7;
distalRotCnt = (d_rot.getPos() - savedPos_8) * (-1); // negative to make sure CCW ---> +

if (abs(proxLeftCnt) <= 14) { // To account for truncation during conversions
    proxLeftCnt = 0;
}

if (abs(proxRightCnt) <= 14) {
    proxRightCnt = 0;
}

if (abs(midLeftCnt) <= 14) {
    midLeftCnt = 0;
}

if (abs(midRightCnt) <= 14) {
    midRightCnt = 0;
}

if (abs(midRotCnt) <= 14) {
    midRotCnt = 0;
}

if (abs(distalLeftCnt) <= 14) {
    distalLeftCnt = 0;
}

if (abs(distalRightCnt) <= 14) {
    distalRightCnt = 0;
}

if (abs(distalRotCnt) <= 14) {
    distalRotCnt = 0;
}

// Calculate Curvature of TREE sections:
Serial.println("Prox Left Cnt:");
Serial.println(proxLeftCnt);
Serial.println("Prox Right Cnt:");
Serial.println(proxRightCnt);

if (proxLeftCnt == 0){ //no change in motor position
    deltaProxLen1 = 0;
}
else if (proxLeftCnt > 0){
    //relationship between change in tendon length and encoder counts
    deltaProxLen1 = -0.0000006*(pow(proxLeftCnt,2)) + 0.0034*proxLeftCnt + 0.068;
}
else if (proxLeftCnt < 0){
    deltaProxLen1 = 0.0000006*(pow(proxLeftCnt,2)) + 0.0034*proxLeftCnt - 0.068;
}
proxTendonLen1 = proxTendonLen1 - deltaProxLen1;

if (proxRightCnt == 0){ //negative encoder counts --> winding
    deltaProxLen2 = 0;
}
else if (proxRightCnt > 0){
    deltaProxLen2 = 0.0000006*(pow(proxRightCnt,2)) - 0.0034*proxRightCnt - 0.068;
}
else if (proxRightCnt < 0){
    deltaProxLen2 = -0.0000006*(pow(proxRightCnt,2)) - 0.0034*proxRightCnt + 0.068;
}
proxTendonLen2 = proxTendonLen2 - deltaProxLen2;
Serial.println("Delta Prox Len 1:");
Serial.println(deltaProxLen1);
Serial.println("Delta Prox Len 2:");
Serial.println(deltaProxLen2);

Serial.println("Prox Tendon Len 1:");
Serial.println(proxTendonLen1);
Serial.println("Prox Tendon Len 2:");
Serial.println(proxTendonLen2);

//Use l1 and l2 in curvature equation for proximal section:
k_p = (proxTendonLen1 - proxTendonLen2)/(d_prox*(proxTendonLen1 + proxTendonLen2));
if (proxTendonLen1 == proxTendonLen2){
    Serial.println("PROX EQUAL!!");
k_p = 0.0001;
}

//Serial.println("Mid Left Cnt:");
Serial.println(midLeftCnt);
Serial.println("Mid Right Cnt:");
Serial.println(midRightCnt);
if (midLeftCnt == 0) { //no change in motor position
deltaMidLen1 = 0;
}
else if (midLeftCnt > 0){
    //relationship between change in tendon length and encoder counts
deltaMidLen1 = -0.0000006*(pow(midLeftCnt,2)) + 0.0034*midLeftCnt + 0.068;
}
else if (midLeftCnt < 0){
deltaMidLen1 = 0.0000006*(pow(midLeftCnt,2)) + 0.0034*midLeftCnt - 0.068;
}
midTendonLen1 = midTendonLen1 - deltaMidLen1;

if (midRightCnt == 0){ //negative encoder counts --> winding
deltaMidLen2 = 0;
}
else if (midRightCnt > 0){
deltaMidLen2 = 0.0000006*(pow(midRightCnt,2)) - 0.0034*midRightCnt - 0.068;
}
else if (midRightCnt < 0){
deltaMidLen2 = -0.0000006*(pow(midRightCnt,2)) - 0.0034*midRightCnt + 0.068;
}
midTendonLen2 = midTendonLen2 - deltaMidLen2;
//Serial.println("Delta Mid Len 1:");
Serial.println(deltaMidLen1);
Serial.println("Delta Mid Len 2:");
Serial.println(deltaMidLen2);

Serial.println("Mid Tendon Len 1:");
Serial.println(midTendonLen1);
Serial.println("Mid Tendon Len 2:");
Serial.println(midTendonLen2);*/
// Use $l_1$ and $l_2$ in curvature equation for middle section:
k_m = (midTendonLen1 - midTendonLen2)/(d_mid*(midTendonLen1 + midTendonLen2));
if (midTendonLen1 == midTendonLen2)
  Serial.println("MID EQUAL!!");
k_m = 0.0001;
}

/* Serial.println("Distal Left Cnt:");
Serial.println(distalLeftCnt);
Serial.println("Distal Right Cnt:");
Serial.println(distalRightCnt); */
if (distalLeftCnt == 0){ //no change in motor position
deltaDistalLen1 = 0;
}
else if (distalLeftCnt > 0){
  // relationship between change in tendon length and encoder counts
  deltaDistalLen1 = -0.0000006*(pow(distalLeftCnt,2)) + 0.0034*distalLeftCnt + 0.068;
}
else if (distalLeftCnt < 0){
  deltaDistalLen1 = 0.0000006*(pow(distalLeftCnt,2)) + 0.0034*distalLeftCnt - 0.068;
}
distalTendonLen1 = distalTendonLen1 - deltaDistalLen1;

if (distalRightCnt == 0){ // negative encoder counts --> winding
  deltaDistalLen2 = 0;
}
else if (distalRightCnt > 0){
  deltaDistalLen2 = 0.0000006*(pow(distalRightCnt,2)) - 0.0034*distalRightCnt - 0.068;
}
else if (distalRightCnt < 0){
  deltaDistalLen2 = -0.0000006*(pow(distalRightCnt,2)) - 0.0034*distalRightCnt + 0.068;
}
distalTendonLen2 = distalTendonLen2 - deltaDistalLen2;
/* Serial.println("Delta Distal Len 1:"); */
Serial.println(deltaDistLen1);
Serial.println("Delta Distal Len 1:");
Serial.println(deltaDistLen2);
Serial.println("Delta Distal Len 2:");
Serial.println(distalTendonLen1);
Serial.println("Distal Tendon Len 1:");
Serial.println(distalTendonLen2);
Serial.println("Distal Tendon Len 2:");
// Use l1 and l2 in curvature equation for distal section:
k_d = (distalTendonLen1 - distalTendonLen2) / (d_dist * (distalTendonLen1 + distalTendonLen2));
if (distalTendonLen1 == distalTendonLen2){
    Serial.println("DISTAL EQUAL!");
    k_d = 0.0001;
}

// Calculate Rotation (phi) of TREE sections:
deltaPhi_m = (TWO_PI/7362)*midRotCnt; // in radians
phi_m = phi_m + deltaPhi_m;
deltaPhi_d = (TWO_PI/3924)*distalRotCnt; // in radians
phi_d = phi_d + deltaPhi_d;

// Calculate Arc Length (s) of TREE sections:
//NOTE: s_p never changes
get_mDist();
s_m = s_mInit + (m_maxDist - m_distance); // m_distance = sensor reading
get_dDist();
s_d = s_dInit + (d_distance - d_minDist); // d_distance = sensor reading
if ((s_m - s_mInit) > 0){ // middle section has extended
    s_d = s_d - (m_maxDist - m_distance); // distal section's effective length is shorter
}
Serial.println("s_p");
Serial.println(s_p, 4);}
Serial.println("s\_m");
Serial.println(s\_m, 4);
Serial.println("s\_d");
Serial.println(s\_d, 4);
Serial.println("k\_p");
Serial.println(k\_p, 4);
Serial.println("k\_m");
Serial.println(k\_m, 4);
Serial.println("k\_d");
Serial.println(k\_d, 4);
Serial.println("phi\_p");
Serial.println(phi\_p, 4);
Serial.println("phi\_m");
Serial.println(phi\_m, 4);
Serial.println("phi\_d");
Serial.println(phi\_d, 4);

// Calculate the Homogeneous Transformation Matrix for the robot
// Spatial Model

P[0][0] = cos(phi\_p)*cos(s\_p*k\_p); P[0][1] = -sin(phi\_p); P[0][2] = -cos(phi\_p)*sin(s\_p*k\_p); P[0][3] = (1/k\_p)*(cos(phi\_p)-cos(phi\_p)*cos(s\_p*k\_p));
P[1][0] = sin(phi\_p)*cos(s\_p*k\_p); P[1][1] = cos(phi\_p); P[1][2] = -sin(phi\_p)*sin(s\_p*k\_p); P[1][3] = -(1/k\_p)*(sin(phi\_p)-sin(phi\_p)*cos(s\_p*k\_p));
P[2][0] = sin(s\_p*k\_p); P[2][1] = 0; P[2][2] = cos(s\_p*k\_p); P[2][3] = (1/k\_p)*sin(s\_p*k\_p);
P[3][0] = 0; P[3][1] = 0; P[3][2] = 0; P[3][3] = 1;

M[0][0] = cos(phi\_m)*cos(s\_m*k\_m); M[0][1] = -sin(phi\_m); M[0][2] = -cos(phi\_m)*sin(s\_m*k\_m); M[0][3] = (1/k\_m)*(cos(phi\_m)-cos(phi\_m)*cos(s\_m*k\_m));
M[1][0] = sin(phi\_m)*cos(s\_m*k\_m); M[1][1] = cos(phi\_m); M[1][2] = -sin(phi\_m)*sin(s\_m*k\_m); M[1][3] = -(1/k\_m)*(sin(phi\_m)-sin(phi\_m)*cos(s\_m*k\_m));
M[2][0] = sin(s\_m*k\_m); M[2][1] = 0; M[2][2] = cos(s\_m*k\_m); M[2][3] = (1/k\_m)*sin(s\_m*k\_m);
M[3][0] = 0; M[3][1] = 0; M[3][2] = 0; M[3][3] = 1;

D[0][0] = cos(phi\_d)*cos(s\_d*k\_d); D[0][1] = -sin(phi\_d); D[0][2] = -cos(phi\_d)*sin(s\_d*k\_d); D[0][3] = (1/k\_d)*(cos(phi\_d)-cos(phi\_d)*cos(s\_d*k\_d));
D[1][0] = sin(phi\_d)*cos(s\_d*k\_d); D[1][1] = cos(phi\_d); D[1][2] = -sin(phi\_d)*sin(s\_d*k\_d); D[1][3] = -(1/k\_d)*(sin(phi\_d)-sin(phi\_d)*cos(s\_d*k\_d));
\[ D[2][0] = \sin(s_d*k_d) ; \ D[2][1] = 0 ; \ D[2][2] = \cos(s_d*k_d) ; \ D[2][3] = (1/k_d)\sin(s_d*k_d) ; \]
\[ \]
\[ D[3][0] = 0 ; \ D[3][1] = 0 ; \ D[3][2] = 0 ; \ D[3][3] = 1 ; \]

// Thresholds for the matrix values
// fabs() is floating point abs val
for (int i = 0; i <= 3; i++){
    for (int j = 0; j <= 3; j++){
        if ((fabs(P[i][j]) < 0.35) && (fabs(P[i][j]) != 0)){ // 0.006; 0.17
            P[i][j] = 0.00;
        }
        if ((fabs(M[i][j]) < 0.35) && (fabs(M[i][j]) != 0)){
            M[i][j] = 0.00;
        }
        if ((fabs(D[i][j]) < 0.35) && (fabs(D[i][j]) != 0)){
            D[i][j] = 0.00;
        }
    }
} Matrix.Print((float*)P, N, N, "P");
Matrix.Print((float*)M, N, N, "M");
Matrix.Print((float*)D, N, N, "D");
Matrix.Multiply((float*)P, (float*)M, N, N, (float*)A); // P*M = A
Matrix.Multiply((float*)A, (float*)D, N, N, (float*)H); // A*D = H = P*M*D
Matrix.Print((float*)H, N, N, "H"); // final matrix for system
}

void loop() {
    if (saveTimer == 100){ // Save (roughly) every minute
        saveToEEPROM();
        saveTimer = 0;
    }
    get_dDist();
    get_mDist();
```java
motor_select = cmdLocal.charAt(0);  // first char of input = motor selection
cmdLocal = cmdLocal.substring(2);  // command to send motor starts after delimiter (space)
int i = 10;

switch (motor_select){
    //——SEND CUSTOM COMMANDS TO MOTORS——
    case '1':
        p_left.sendCmd(cmdLocal);  // returns current encoder count/position
        if (cmdLocal.equals("P")){
            while (i > 0){
                p_left.getPos();
                delay(200);
                i = i - 1;
            }
        }
        break;
    case '2':
        p_right.sendCmd(cmdLocal);
        if (cmdLocal.equals("P")){
            while (i > 0){
                p_right.getPos();
                delay(200);
                i = i - 1;
            }
        }
        break;
    case '3':
        m_left.sendCmd(cmdLocal);
        if (cmdLocal.equals("P")){
            while (i > 0){
                m_left.getPos();
                delay(200);
                i = i - 1;
            }
        }
        break;
    case '4':
        m_right.sendCmd(cmdLocal);
```

96
if (cmdLocal.equals("P")) {
    while (i > 0) {
        m_right.getPos();
        delay(200);
        i = i - 1;
    }
}
break;
}
case '5':
    if (cmdLocal.equals("P")) {
        while (i > 0) {
            m_rot.getPos();
            delay(200);
            i = i - 1;
        }
    }
    else {
        reverseDir = cmdLocal.substring(1).toString() * (-1); // to make sure Ccw --> +
        cmdLocal = cmdLocal.charAt(0) + (String) reverseDir; // rebuild cmd string
        m_rot.sendCmd(cmdLocal);
    }
    break;
case '6':
    d_left.sendCmd(cmdLocal);
    if (cmdLocal.equals("P")) {
        while (i > 0) {
            d_left.getPos();
            delay(200);
            i = i - 1;
        }
    }
    break;
case '7':
    d_right.sendCmd(cmdLocal);
    if (cmdLocal.equals("P")) {
        while (i > 0) {
            d_right.getPos();
            delay(200);
            i = i - 1;
        }
    }
    break;
break;

case '8':
    if (cmdLocal.equals("P")){
        while (i > 0){
            d_rot.getPos();
            delay(200);
            i = i - 1;
        }
    }
    else {
        reverseDir = cmdLocal.substring(1).toInt() * (-1); // to make sure CCW -> +
        cmdLocal = cmdLocal.charAt(0) + (String)reverseDir; // rebuild cmd string
        d_rot.sendCmd(cmdLocal);
    }
    break;

case '9':
    distanceToTravel = cmdLocal.toInt();
    m_startPos = m_distance;
    if (distanceToTravel > 0){
        while ((m_distance != (m_startPos - distanceToTravel)) && m_distance !=
            m_minDist){ // prevents going too far forward
            get_mDist();
            fwd(m_pwm, pwm_val, m_dir); // go forward
        }
        stp(m_pwm, m_dir);
    } else if (distanceToTravel < 0){
        while ((m_distance != (m_startPos - distanceToTravel)) && m_distance !=
            m_maxDist){ // prevents going too far back
            get_mDist();
            bwd(m_pwm, pwm_val, m_dir); // go backward
        }
        stp(m_pwm, m_dir);
    }
    break;

case '0':
    distanceToTravel = cmdLocal.toInt();
ds t a r t P o s = d _ d i s t a n c e ;
//**LATER: make sure can’t go too far forward
if ( distanceToTravel > 0){
    while ( d _ d i s t a n c e != ( d _ s t a r t P o s + distanceToTravel)){
        get_dDist();
        /*Due to orientation of linear actuator, dir methods
        * are reversed to match actual direction of motion
        */
        bwd(d_pwm, pwm_val, d_dir); //go forward
    }
    stp(d_pwm, d_dir);
}
else if ( distanceToTravel < 0){
    while ( ((d_distance != (d_startPos + distanceToTravel)) && d_distance !=
        d _ m i n D i s t ){ //prevents going too far back
        get_dDist();
        fwd(d_pwm, pwm_val, d_dir); //go backward
    }
    stp(d_pwm, d_dir);
}
break;
//= = = = M O V E M O T O R S I N I N C R E M E N T S = = = =
case 'q':
    p_left . sendCmd("R250"); //wind
    break;
    case 'w':
    p_left . sendCmd("R−250"); //unwind
    break;
    case 'e':
    p_right . sendCmd("R−250"); //wind
    break;
    case 'r':
    p_right . sendCmd("R250"); //unwind
    break;
    case 'a':
    m_left . sendCmd("R250"); //wind
    break;
    case 's':
    m_left . sendCmd("R−250"); //unwind
break;
    case 'd':
        m_rot.sendCmd("R-250"); //CW
        break;
    case 'f':
        m_rot.sendCmd("R250"); //CW
        break;
    case 'g':
        m_right.sendCmd("R-250"); //wind
        break;
    case 'h':
        m_right.sendCmd("R250"); //unwind
        break;
    case 'z':
        d_left.sendCmd("R250"); //wind
        break;
    case 'x':
        d_left.sendCmd("R-250"); //unwind
        break;
    case 'c':
        d_rot.sendCmd("R-250"); //CW
        break;
    case 'v':
        d_rot.sendCmd("R250"); //CW
        break;
    case 'b':
        d_right.sendCmd("R-250"); //wind
        break;
    case 'n':
        d_right.sendCmd("R250"); //unwind
        break;
    case 'u': // mid extend 10 cm
        m_startPos = m_distance;
        while ((m_distance != (m_startPos - 10)) && m_distance != m_minDist) { // prevents going too far forward
            get_mDist();
            fwd(m_pwm, pwm_val, m_dir); // go forward
        }
        stp(m_pwm, m_dir);
break;
case 'j': // mid retract 10 cm
    while ((m_distance != (m_startPos + 10)) && m_distance != m_maxDist) { // prevents going too far back
        get_mDist();
        bwd(m_pwm, pwm_val, m_dir); // go backward
    }
    stp(m_pwm, m_dir);
    break;
case 'i': // distal extend 10 cm
    d_startPos = d_distance;
    while (d_distance != (d_startPos + 10)) {
        get_dDist();
        bwd(d_pwm, pwm_val, d_dir); // go forward
    }
    stp(d_pwm, d_dir);
    break;
case 'k': // distal retract 10 cm
    d_startPos = d_distance;
    while ((d_distance != (d_startPos - 10)) && d_distance != d_minDist) { // prevents going too far back
        get_dDist();
        fwd(d_pwm, pwm_val, d_dir); // go backward
    }
    stp(d_pwm, d_dir);
    break;
case 't':
    // Save to EEPROM
    saveToEEPROM();
    break;
case 'y':
    // Read from EEPROM
    readFromEEPROM();
    break;
case 'm':
    // Forward Kinematic Calculations
    fwdKin();
    break;
// =*=*=*=*=*=*=*= AUTOMATED MOTION SCRIPTS *=*=*=*=*=*=*=*=

101
case 'o':
  // Example/Demo Run
  // *** Extend 15 cm first !!! ***
  d_rot.sendCmd("R−981"); // CCW
  d_right.sendCmd("R−750");
  d_left.sendCmd("R−250");
  delay(2000);
  d_rot.sendCmd("R−981"); // CCW
  d_right.sendCmd("R250");
  d_left.sendCmd("R125");
  d_rot.sendCmd("G0");
  d_right.sendCmd("G0");
  d_left.sendCmd("G0");
  delay(5000);
  d_rot.sendCmd("R981"); // CW
  d_left.sendCmd("R750");
  d_right.sendCmd("R250");
  delay(2000);
  d_rot.sendCmd("R981"); // CW
  d_left.sendCmd("R−250");
  d_right.sendCmd("R−125");
  d_rot.sendCmd("G0");
  d_right.sendCmd("G0");
  d_left.sendCmd("G0");
  break;
  //= = = CLEANING/BLOWING/INSPECTION EXPER. SCRIPT = = =
  case 'p':
    //--- Bend tip down ---
    d_left.sendCmd("G−100");
    d_right.sendCmd("G−750");
    delay(5000); // wait 5 sec
    //--- Bend tip up ---
    d_left.sendCmd("G0");
    d_right.sendCmd("G0");
    delay(5000); // wait 5 sec
    //--- Blow air (manually w/ air compressor) ---
// delay (2000); // wait 2 sec

//—rotate tip CW 90ish deg (90 deg = 981 cnts)—
 d_rot.sendCmd("G1200");
delay(7000); // wait 7 sec

//—bend left/right while blowing—
 d_left.sendCmd("G900"); //bend right
 //d_right.sendCmd("G250");
delay(5000); // wait 5 sec
 d_right.sendCmd("G-750"); //bend left
 d_left.sendCmd("G750");
delay(5000); // wait 5 sec
 d_left.sendCmd("G900"); //bend right
 d_right.sendCmd("G0");
delay(2000); // wait 2 sec
 d_left.sendCmd("G0"); // back to normal
 delay(3000); // wait 3 sec

//—inspect w/ Tendril branch (left)—
 Serial2.println("t"); // extend left branch all the way
delay(5000); // wait 5 sec
 Serial2.println("1 850"); // was 750
delay(500); // allow time for command to be sent

//—sweep rest w/ Tendril branch (right)—
 Serial2.println("y"); // extend right branch all the way
delay(5000); // wait 5 sec
 Serial2.println("6 -750");
delay(2000); // allow time for command to be sent
 Serial2.println("5 -750"); // bend left
delay(2000);
 Serial2.println("6 -600");
delay(1000);
 Serial2.println("5 0"); // bend right
delay(500);
 Serial2.println("8 -300");
delay(500);
Serial2.println("7 -300");
delay(2000);
Serial2.println("8 0"); //bend left
delay(500);
Serial2.println("7 0");
delay(500);
Serial2.println("5 -750");
delay(2000);
Serial2.println("5 0"); //bend right
delay(500);
Serial2.println("8 -300");
delay(500);
Serial2.println("7 -300");
delay(500);
break;
//======RESET/ZERO AFTER ABOVE EXPER=====
case 'l':
    d_rot.sendCmd("G0");
delay(1000); //wait 1 sec
d_left.sendCmd("G0");
delay(1000); //wait 1 sec
d_right.sendCmd("G0");
delay(1000); //wait 1 sec
    Serial2.println("g"); //left branch middle/
    break;
default:
    break;
} //end of switch

cmdLocal = ""; //reset command string
saveTimer++;
delay(500);
//Serial.println("Message Received: " + cmdRecv); //***DEBUGGING...REMOVE THIS LINE!
} //end of loop
// ---------- SENDING STRINGS OVER SERIAL ---------------

//*** Make sure Serial Monitor is set to add new line character
Since end of message char is '\n', can use Serial2.println() to send messages
Will not work with just Serial2.print() - message will not be fully read (no '\n')
*

void serialEvent(){ // read from Mega Serial Monitor, send to Due (Branches)
  byte buf[256];
  char c;
  bool msgForDue = false;
  while(Serial.available()){
    c = Serial.read(); // reads input 1 char at a time
    delay(5); // allow time to read
    if (c == '>')// Command is to be sent to the Due
      msgForDue = true;
  }
  if (msgForDue && c != '>'){ // does not add '>' to String that is sent
    cmdSent += c; // adds char to String
    delay(5); // allow time to save char
    if (c == '\n')// marks end of message/command
      cmdSent.getBytes(buf, 256);
      Serial2.write(buf, cmdSent.length());
      cmdSent = "";
      msgForDue = false;
  }
  else{ // Command is to stay local (on the Mega)
    if (c != '>'){ // just to be sure '>' is ignored
      cmdLocal += c;
      delay(5); // allow time to save char
    }
  }
  // end of while
  cmdLocal.trim();
} // end of serialEvent()

void serialEvent2(){ // read from Due (data received), send/print to Mega Serial Monitor
  char c;
  while(Serial2.available()){

c = Serial2.read();
rawMsg += c;
delay(5); //allow time to save char
if (c == '\n'){
  //marks end of message/command
  rawMsg.trim(); //removes new line char
  cmdRecv = rawMsg;
  Serial.println("Received -> " + cmdRecv);
  rawMsg = "";
}

Listing 1: Arduino Mega Code (Master Device) - tree_kinematic_control_master.ino

/*
* This program runs on the Arduino Due (Slave Device). It communicates with the
* Mega (Master Device) to share the branch motor positions.
* This program controls the extension, retraction, and tendon actuation of the
* Tendril branches on the TREE.
* Created by Michael Lastinger
*
* Pololu Magnetic Encoder Kit:
* Encoder: 20 Counts per revolution (cpr)
* To compute the cpr of the gearbox output shaft, multiply the gear ratio by 20.
* Gear ratio (20 mm diam., 12V motor) = 488.28125:1
*/
#include <BranchMotor.h>
#include <Servo.h>

#define enc1PinA 38 //motor 1
#define enc1PinB 39
#define enc2PinA 42 //motor 2
#define enc2PinB 43
#define enc3PinA 46 //motor 3
#define enc3PinB 47
#define enc4PinA 50 //motor 4
#define enc4PinB 51
#define enc5PinA 34 //motor 5
#define enc5PinB 35
# define enc6PinA 30 // motor 6
# define enc6PinB 31
# define enc7PinA 26 // motor 7
# define enc7PinB 27
# define enc8PinA 22 // motor 8
# define enc8PinB 23

volatile int32_t mtrPositions[8] = {0,0,0,0,0,0,0,0};

// Left Branch Actuators:
BranchMotor mtr1(40, 9, 150, &mtrPositions[0]); // dirPin, pwmPin, pwmVal, mtr count/pos
BranchMotor mtr2(44, 8, 150, &mtrPositions[1]);
BranchMotor mtr3(48, 7, 150, &mtrPositions[2]);
BranchMotor mtr4(52, 6, 150, &mtrPositions[3]);
Servo leftBranch; // left servo

// Right Branch Actuators:
BranchMotor mtr5(36, 5, 150, &mtrPositions[4]); // dirPin, pwmPin, pwmVal, mtr count/pos
BranchMotor mtr6(32, 4, 150, &mtrPositions[5]);
BranchMotor mtr7(28, 3, 150, &mtrPositions[6]);
BranchMotor mtr8(24, 2, 150, &mtrPositions[7]);
Servo rightBranch; // right servo

String rawMsg = "";
String cmdRecv = "";
String cmdSent = "";
char ctrl = ' ';
bool newMsg = false;
const int8_t lookup_table[] = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};

void setup() {
    leftBranch.attach(12);
    rightBranch.attach(13);

    pinMode(enc1PinA, INPUT_PULLUP); // sets pin A as input
digitalWrite(enc1PinA, HIGH); // turn on pullup resistors
pinMode(enc1PinB, INPUT_PULLUP); // sets pin B as input
digitalWrite(enc1PinB, HIGH); // turn on pullup resistors
attachInterrupt(digitalPinToInterrupt(enc1PinA), mtr1Interrupt, CHANGE);
attachInterrupt(digitalPinToInterrupt(enc1PinB), mtr1Interrupt, CHANGE);

pinMode(enc2PinA, INPUT_PULLUP); // sets pin A as input
digitalWrite(enc2PinA, HIGH); // turn on pullup resistors
pinMode(enc2PinB, INPUT_PULLUP); // sets pin B as input
digitalWrite(enc2PinB, HIGH); // turn on pullup resistors
attachInterrupt(digitalPinToInterrupt(enc2PinA), mtr2Interrupt, CHANGE);
attachInterrupt(digitalPinToInterrupt(enc2PinB), mtr2Interrupt, CHANGE);

pinMode(enc3PinA, INPUT_PULLUP); // sets pin A as input
digitalWrite(enc3PinA, HIGH); // turn on pullup resistors
pinMode(enc3PinB, INPUT_PULLUP); // sets pin B as input
digitalWrite(enc3PinB, HIGH); // turn on pullup resistors
attachInterrupt(digitalPinToInterrupt(enc3PinA), mtr3Interrupt, CHANGE);
attachInterrupt(digitalPinToInterrupt(enc3PinB), mtr3Interrupt, CHANGE);

pinMode(enc4PinA, INPUT_PULLUP); // sets pin A as input
digitalWrite(enc4PinA, HIGH); // turn on pullup resistors
pinMode(enc4PinB, INPUT_PULLUP); // sets pin B as input
digitalWrite(enc4PinB, HIGH); // turn on pullup resistors
attachInterrupt(digitalPinToInterrupt(enc4PinA), mtr4Interrupt, CHANGE);
attachInterrupt(digitalPinToInterrupt(enc4PinB), mtr4Interrupt, CHANGE);

pinMode(enc5PinA, INPUT_PULLUP); // sets pin A as input
digitalWrite(enc5PinA, HIGH); // turn on pullup resistors
pinMode(enc5PinB, INPUT_PULLUP); // sets pin B as input
digitalWrite(enc5PinB, HIGH); // turn on pullup resistors
attachInterrupt(digitalPinToInterrupt(enc5PinA), mtr5Interrupt, CHANGE);
attachInterrupt(digitalPinToInterrupt(enc5PinB), mtr5Interrupt, CHANGE);

pinMode(enc6PinA, INPUT_PULLUP); // sets pin A as input
digitalWrite(enc6PinA, HIGH); // turn on pullup resistors
pinMode(enc6PinB, INPUT_PULLUP); // sets pin B as input
digitalWrite(enc6PinB, HIGH); // turn on pullup resistors
attachInterrupt(digitalPinToInterrupt(enc6PinA), mtr6Interrupt, CHANGE);
attachInterrupt(digitalPinToInterrupt(enc6PinB), mtr6Interrupt, CHANGE);
pinMode(enc7PinA, INPUT_PULLUP); // sets pin A as input
digitalWrite(enc7PinA, HIGH); // turn on pullup resistors
pinMode(enc7PinB, INPUT_PULLUP); // sets pin B as input
digitalWrite(enc7PinB, HIGH); // turn on pullup resistors
attachInterrupt(digitalPinToInterrupt(enc7PinA), mtr7Interrupt, CHANGE);
attachInterrupt(digitalPinToInterrupt(enc7PinB), mtr7Interrupt, CHANGE);

pinMode(enc8PinA, INPUT_PULLUP); // sets pin A as input
digitalWrite(enc8PinA, HIGH); // turn on pullup resistors
pinMode(enc8PinB, INPUT_PULLUP); // sets pin B as input
digitalWrite(enc8PinB, HIGH); // turn on pullup resistors
attachInterrupt(digitalPinToInterrupt(enc8PinA), mtr8Interrupt, CHANGE);
attachInterrupt(digitalPinToInterrupt(enc8PinB), mtr8Interrupt, CHANGE);

Serial.begin(9600);
Serial2.begin(9600);
}

void loop() {
  if (newMsg) {
    ctrl = cmdRecv.charAt(0);
    cmdRecv = cmdRecv.substring(2); // command to send motor starts after delimiter (space)
    switch (ctrl) {
      // -------------------BRANCH EXTENSION/RETRACTION-------------------
      case '[': // left branch — go to specified position (extend/retract)
        leftBranch.write(cmdRecv.toInt());
        break;
      case ']': // right branch — go to specified position (extend/retract)
        rightBranch.write(cmdRecv.toInt());
        break;
      case 't': // extend left branch fully
        leftBranch.write(145); // 165 but interferes with tendons
        break;
      case 'g': // left branch to middle
        leftBranch.write(90);
        break;
      case 'b': // retract left branch fully
        break;
    }
  }
}
leftBranch.write(5);
break;
case 'y': //extend right branch fully
    rightBranch.write(35); //15 but interferes with tendons
    break;
case 'h': //right branch to middle
    rightBranch.write(90);
    break;
case 'n': //retract right branch fully
    rightBranch.write(180);
    break;

//———TENDON WINDING/UNWINDING———
case '1': //motor 1 - get position or go to specified position
    if (cmdRecv.equals("P")) {
        Serial2.println(mtrPositions[0]);
    }
    else {
        mtr1.goToPos(cmdRecv.toInt());
    }
    break;
case '2':
    if (cmdRecv.equals("P")) {
        Serial2.println(mtrPositions[1]);
    }
    else {
        mtr2.goToPos(cmdRecv.toInt());
    }
    break;
case '3':
    if (cmdRecv.equals("P")) {
        Serial2.println(mtrPositions[2]);
    }
    else {
        mtr3.goToPos(cmdRecv.toInt());
    }
    break;
case '4':
    if (cmdRecv.equals("P")) {
        Serial2.println(mtrPositions[3]);
    }
} else {
    mtr4.goToPos(cmdRecv.toInt());
}
break;
\texttt{case '5':}
    if (cmdRecv.equals("P")) {
        \texttt{Serial2.println(mtrPositions[4]);}
    } else {
        mtr5.goToPos(cmdRecv.toInt());
    }
break;
\texttt{case '6':}
    if (cmdRecv.equals("P")) {
        \texttt{Serial2.println(mtrPositions[5]);}
    } else {
        mtr6.goToPos(cmdRecv.toInt());
    }
break;
\texttt{case '7':}
    if (cmdRecv.equals("P")) {
        \texttt{Serial2.println(mtrPositions[6]);}
    } else {
        mtr7.goToPos(cmdRecv.toInt());
    }
break;
\texttt{case '8':}
    if (cmdRecv.equals("P")) {
        \texttt{Serial2.println(mtrPositions[7]);}
    } else {
        mtr8.goToPos(cmdRecv.toInt());
    }
break;
\texttt{// TENDON ACTUATION IN RELATIVE INCREMENTS ---}
\texttt{// LEFT BRANCH ---}
case 'r': // motor 1 wind 150 counts
    mtr1.goToPos(mtrPositions[0] + 150);
    break;

case 'f': // motor 1 unwind 150 counts
    mtr1.goToPos(mtrPositions[0] - 150);
    break;

case 'e': // motor 2 wind 150 counts
    mtr2.goToPos(mtrPositions[1] + 150);
    break;

case 'd': // motor 2 unwind 150 counts
    mtr2.goToPos(mtrPositions[1] - 150);
    break;

case 'w': // motor 3 wind 150 counts
    mtr3.goToPos(mtrPositions[2] + 150);
    break;

case 's': // motor 3 unwind 150 counts
    mtr3.goToPos(mtrPositions[2] - 150);
    break;

case 'q': // motor 4 wind 150 counts
    mtr4.goToPos(mtrPositions[3] + 150);
    break;

case 'a': // motor 4 unwind 150 counts
    mtr4.goToPos(mtrPositions[3] - 150);
    break;

//------ RIGHT BRANCH ------

case 'u': // motor 5 wind 150 counts
    mtr5.goToPos(mtrPositions[4] - 150); // negative value = wind on right branch
    break;

case 'j': // motor 5 unwind 150 counts
    mtr5.goToPos(mtrPositions[4] + 150); // positive value = unwind on right branch
    break;

case 'i': // motor 6 wind 150 counts
    mtr6.goToPos(mtrPositions[5] - 150);
    break;

case 'k': // motor 6 unwind 150 counts
    mtr6.goToPos(mtrPositions[5] + 150);
    break;

case 'o': // motor 7 wind 150 counts

mtr7.goToPos(mtrPositions[6] - 150);
break;
case 'l': //motor 7 unwind 150 counts
    mtr7.goToPos(mtrPositions[6] + 150);
break;
case 'p': //motor 8 wind 150 counts
    mtr8.goToPos(mtrPositions[7] - 150);
break;
case ';': //motor 8 unwind 150 counts
    mtr8.goToPos(mtrPositions[7] + 150);
break;
default:
    break;
}
} // switch end

ctrl = ' ';
}
} // end of if (newMsg)

newMsg = false;
}
} // loop end

void mtr1Interrupt() { // pins 38 & 39, PC6 & PC7 (port C, pins 6 & 7)
    static uint8_t mtr1_enc_val = 0;
    mtr1_enc_val = mtr1_enc_val << 2; //save current state (00AB) [4 bits] as previous
    (A'B'00) [4 bits], where A is encoder channel A and B is channel B

    /* the OR operation in (), the second OR below reading left to right, "combines"
    (00A0) with (000B) to form (00AB)
    * first OR replaces 0's in (A'B'00) with new A, B to form (A'B'AB) */
    mtr1_enc_val = mtr1_enc_val | (((PIOC->PIO_PDSR & 0x40) >> 5) | ((PIOC->PIO_PDSR & 0
    x80) >> 7));
    mtrPositions[0] = mtrPositions[0] - lookup_table[mtr1_enc_val & 0b1111]; // "0b"
    denotes binary; this gets rid of all other bits except 0-3 (the first 4 bits)
}

void mtr2Interrupt() { // pins 42 & 43, PA19 & PA20
    static uint8_t mtr2_enc_val = 0;
    mtr2_enc_val = mtr2_enc_val << 2;
    mtr2_enc_val = mtr2_enc_val | (((PIOA->PIO_PDSR & 0x80000) >> 18) | ((PIOA->PIO_PDSR
    & 0x100000) >> 20));
void mtr3Interrupt() { // pins 46 & 47, PC17 & PC16
    static uint8_t mtr3_enc_val = 0;
    mtr3_enc_val = mtr3_enc_val << 2;
    mtr3_enc_val = mtr3_enc_val | (((PIOC->PIO_PDSR & 0x20000) >> 16) | ((PIOC->PIO_PDSR & 0x10000) >> 16));
}

void mtr4Interrupt() { // pins 50 & 51, PC13 & PC12
    static uint8_t mtr4_enc_val = 0;
    mtr4_enc_val = mtr4_enc_val << 2;
    mtr4_enc_val = mtr4_enc_val | (((PIOC->PIO_PDSR & 0x2000) >> 12) | ((PIOC->PIO_PDSR & 0x8000) >> 12));
}

void mtr5Interrupt() { // pins 34 & 35, PC2 & PC3
    static uint8_t mtr5_enc_val = 0;
    mtr5_enc_val = mtr5_enc_val << 2;
    mtr5_enc_val = mtr5_enc_val | (((PIOC->PIO_PDSR & 0x0008) >> 3));
}

void mtr6Interrupt() { // pins 30 & 31, PD9 & PA7
    static uint8_t mtr6_enc_val = 0;
    mtr6_enc_val = mtr6_enc_val << 2;
    mtr6_enc_val = mtr6_enc_val | (((PIOD->PIO_PDSR & 0x00020) >> 7));
}

void mtr7Interrupt() { // pins 26 & 27, PD1 & PD2
    static uint8_t mtr7_enc_val = 0;
    mtr7_enc_val = mtr7_enc_val << 2;
    mtr7_enc_val = mtr7_enc_val | (((PIOD->PIO_PDSR & 0x0200) >> 8));
}
void mtr8Interrupt() { // pins 22 & 23, PB26 & PA14
    static uint8_t mtr8_enc_val = 0;
    mtr8_enc_val = mtr8_enc_val << 2;
    mtr8_enc_val = mtr8_enc_val | (((PIOB->PIO_PDSR & 0x4000000) >> 25)|((PIOA->PIO_PDSR & 0x4000) >> 14));
}

// = = = = = = SENDING STRINGS OVER SERIAL = = = = = = = = = = = = =
/**
***Make sure Serial Monitor is set to add new line character
Since end of message char is '\n', can use Serial2.println() to send messages
Will not work with just Serial2.print() — message will not be fully read (no '\n')
*/
void serialEvent() { // read from Due Serial Monitor, send to Mega
    byte buf[256];
    char c;
    while (Serial.available()) {
        c = Serial.read(); // reads input 1 char at a time
        cmdSent += c; // adds char to String
        delay(5); // allow time to save char
        if (c == '\n') { // marks end of message/command
            cmdSent.getBytes(buf, 256);
            Serial2.write(buf, cmdSent.length());
            cmdSent = ""
        }
    }
} //end of serialEvent()

void serialEvent2() { //read from Mega (data received), send/print to Due Serial Monitor
    char c;
    newMsg = true;
    while (Serial2.available()) {
        c = Serial2.read();
        rawMsg += c;
    }
delay(5); //allow time to save char
if (c == '\n') { //marks end of message/command
  rawMsg.trim(); //removes new line char
  cmdRecv = rawMsg;
  Serial.println("Received -> " + cmdRecv);
  rawMsg = "";
}
} //end of serialEvent2()

//Not necessary
void writeToMaster(String msg) {
  byte buf[256];
  msg += '\n'; //add new line char so Mega will print received msg to Serial Mon.
  msg.getBytes(buf, 256); //convert to bytes, store in buf for sending (Serial)
  Serial2.write(buf, msg.length());
}

Listing 2: Arduino Due Code (Slave Device) - tendrilBranchActuationSlave_2.ino
Appendix B  C++/Arduino Library Files

/*
   Rhino.h – Library for controlling Rhino Servo DC Motor with UART (Example: http://
   robokits.co.in/motors/high-torque-encoder-dc-servo-motor-10rpm-with-uart-i2c-ppm-
   drive?cPath=2_71&).

   Released into the public domain.
   Contact: samvrit.goku@gmail.com
   Copyright (C) 2016 Samvrit Srinivas

   Modified by Michael Lastinger

   This program is free software: you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program. If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef Rhino_h
#define Rhino_h

#include "Arduino.h"
#include "SoftwareSerial.h"

class SoftwareSerial;

class Rhino
{
public:

117
Rhino(uint8_t rx, uint8_t tx): _serial(rx,tx), _constraint(0), _print(0) {};

void init()
{
    Serial.begin(9600);
    _serial.begin(9600);
    _serial.flush();
    _serial.write("P0\r");
    if(_print)
        Serial.println("Motor Initiated!");
}

void reset()
{
    _serial.flush();
    _serial.write("Y\r");
    if(_print)
        Serial.println("Motor Reset!");
}

void autoCalibrate()
{
    _serial.flush();
    _serial.write("X\r");
    if(_print)
        Serial.println("Motor auto-calibrating...");
}

void sendCmd(String cmd)
{
    char s[cmd.length()+2];
    cmd.toCharArray(s,cmd.length()+1);
    _serial.flush();
    _serial.write(s);
    _serial.write("\r");
    Serial.write("Command sent: ");
    if(_print)
    {
        Serial.write(s);
        Serial.println();
    }
}
void sendCmd(char cmd[])
{
    _serial.flush();
    _serial.write(cmd);
    _serial.write("\r");
    Serial.write("Command sent: ");
    if(_print)
    {
        Serial.write(cmd);
        Serial.println();
    }
}

void rotate(int speed1)
{
    speed1 = constrain(speed1,−255,255);
    String speed = String(speed1);
    speed = "S"+speed;
    char s[speed.length()+2];
    speed.toCharArray(s,speed.length()+1);
    _serial.flush();
    _serial.write(s);
    _serial.write("\r");
    if(_print)
    {
        Serial.write("Rotating motor at speed: ");
        Serial.println(speed1);
    }
}

void stopMotor()
{
    _serial.flush();
    _serial.write("S0\r");
    if(_print)
    Serial.println("Motor stopped");
}

void gotoPos(long pos1)
{
```java
pos1 = ceil(pos1);
int pos2 = (int)pos1;
if(_constraint)
    pos2 = constrain(pos2,_min,_max);
String pos = String(pos2);
pos = "G"+pos;
char s[pos.length()+2];
pos.toCharArray(s,pos.length()+1);
_serial.flush();
_serial.write(s);
_serial.write("\r");
if(_print)
{
    Serial.write("Go to position: ");
    Serial.println(pos2);
}

} gotoAngleDeg(float angle1)
{
    angle1 *= 5;
    angle1 = ceil(angle1);
    int angle2 = (int)angle1;
    if(_constraint)
        angle2 = constrain(angle2,_min,_max);
    String angle = "G"+String(angle2);
    char s[angle.length()+2];
    angle.toCharArray(s,angle.length()+1);
    _serial.flush();
    _serial.write(s);
    _serial.write("\r");
    if(_print)
    {
        Serial.write("Go to angle: ");
        Serial.println(angle1/5);
    }
}

void gotoAngleRad(float angle1)
```
{  
    angle1 *= 286.6242;  
    angle1 = ceil(angle1);  
    int angle2 = (int)angle1;  
    if (_constraint)  
        angle2 = constrain(angle2, _min, _max);  
    String angle = "G"+String(angle2);  
    char s[angle.length()+2];  
    angle.toCharArray(s, angle.length()+1);  
    _serial.flush();  
    _serial.write(s);  
    _serial.write("\r");  
    if (_print)  
    {  
        Serial.write("Go to angle: ");  
        Serial.println(angle2*286.6242);  
    }  
}  

void gotoAngleDegC(float angle1)  
{  
    angle1 = constrain(angle1, -180, 180);  
    angle1 *= 5;  
    angle1 = ceil(angle1);  
    int angle2 = (int)angle1;  
    if (_constraint)  
        angle2 = constrain(angle2, _min, _max);  
    String angle = "G"+String(angle2);  
    char s[angle.length()+2];  
    angle.toCharArray(s, angle.length()+1);  
    _serial.flush();  
    _serial.write(s);  
    _serial.write("\r");  
    if (_print)  
    {  
        Serial.write("Go to angle: ");  
        Serial.println(angle2);  
    }  
}
void gotoAngleRadC(float angle1)
{
    angle1 = constrain(angle1, -3.1415, 3.1415);
    angle1 *= 286.6242;
    angle1 = ceil(angle1);
    int angle2 = (int)angle1;
    if(_constraint)
        angle2 = constrain(angle2, _min, _max);
    String angle = "G"+String(angle2);
    char s[angle.length()+2];
    angle.toCharArray(s, angle.length()+1);
    _serial.flush();
    _serial.write(s);
    _serial.write("r");
    if(_print)
    {
        Serial.write("Go to angle: ");
        Serial.println(angle2/286.6242);
    }
}

void gotoRel(long pos1)
{
    if(_constraint)
    {
        if(_print)
        {
            printOutput(0);
            int curr_pos = getPos();
            printOutput(1);
            if((pos1 > (_max - curr_pos)) || (pos1 < (_min - curr_pos)))
                pos1 = 0;
        }
        else
        {
            int curr_pos = getPos();
            if((pos1 > (_max - curr_pos)) || (pos1 < (_min - curr_pos)))
                pos1 = 0;
        }
    }
String pos = String(pos1);
pos = "R"+pos;
char s[pos.length()+2];
pos.toCharArray(s, pos.length()+1);
_serial.flush();
_serial.write(s);
_serial.write("\r");
if(_print)
{
    Serial.write("Go to relative position: ");
    Serial.println(pos1);
}

void gotoRelDeg(float pos1)
{
    pos1 = pos1*5;
    int pos2 = ceil(pos1);
    if(_constraint)
    {
        if(_print)
        {
            printOutput(0);
            int curr_pos = getPos();
            printOutput(1);
            if((pos1 > (.max - curr_pos)) || (pos1 < (.min - curr_pos)))
                pos2 = 0;
        }
        else
        {
            int curr_pos = getPos();
            if((pos1 > (.max - curr_pos)) || (pos1 < (.min - curr_pos)))
                pos2 = 0;
        }
    }
    String pos = String(pos2);
pos = "R"+pos;
```java
char s[pos.length()+2];
pos.toCharArray(s, pos.length()+1);
_serial.flush();
_serial.write(s);
_serial.write("\r");
if(_print)
{
    Serial.write("Go to relative position: ");
    Serial.println(pos2/5);
}
}

void gotoRelRad(float pos1)
{
pos1 = pos1 * 286.6242;
int pos2 = ceil(pos1);
if(_constraint)
{
    if(_print)
    {
        printOutput(0);
        int curr_pos = getPos();
        printOutput(1);
        if((pos1 > (_max - curr_pos)) || (pos1 < (_min - curr_pos)))
            pos2 = 0;
    }
    else
    {
        int curr_pos = getPos();
        if((pos1 > (_max - curr_pos)) || (pos1 < (_min - curr_pos)))
            pos2 = 0;
    }
}
String pos = String(pos2);
pos = "R" + pos;
char s[pos.length()+2];
pos.toCharArray(s, pos.length()+1);
pos.toCharArray(s, pos.length()+1);
_serial.flush();
```
_serial.write(s);
_serial.write("\r");

if (_print)
{
    Serial.write("Go to relative position: ");
    Serial.println(pos2/286.6242);
}

void setSFGain(int gain1)
{
    gain1 = constrain(gain1, 0, 32767);
    String gain = String(gain1);
    gain = "A"+gain;
    char s[gain.length()+2];
    gain.toCharArray(s, gain.length()+1);
    _serial.flush();
    _serial.write(s);
    _serial.write("\r");

    if (_print)
    {
        Serial.write("Setting Speed Feedback Gain: ");
        Serial.println(gain1);
    }
}

void setP(int p1)
{
    String p = String(p1);
    p = "B"+p;
    char s[p.length()+2];
    p.toCharArray(s, p.length()+1);
    _serial.flush();
    _serial.write(s);
    _serial.write("\r");
    if (_print)
    {
        Serial.write("Setting P: ");
        Serial.println(p1);
    }
```java
void setI(int i1)
{
    String i = String(i1);
    i = "C"+i;
    char s [i.length()+2];
    i.toCharArray(s, i.length()+1);
    _serial.flush();
    _serial.write(s);
    _serial.write("\r");
    if (_print)
    {
        Serial.write("Setting I: ");
        Serial.println(i1);
    }
}

void setD(int d1)
{
    d1 = constrain(d1,0,255);
    String d = String(d1);
    d = "D"+d;
    char s [d.length()+2];
    d.toCharArray(s, d.length()+1);
    _serial.flush();
    _serial.write(s);
    _serial.write("\r");
    if (_print)
    {
        Serial.write("Setting D: ");
        Serial.println(d1);
    }
}

void setSpeed(int speed1)
{
    speed1 = constrain(speed1,0,255);
}
```
String speed = String(speed1);
speed = "M" + speed;
char s[speed.length() + 2];
speed.toCharArray(s, speed.length() + 1);
_serial.flush();
_serial.write(s);
_serial.write("\r");
if (_print)
{
    Serial.write("Setting maximum speed: ");
    Serial.println(speed1);
}

void setI2CAddr(int addr1)
{
    addr1 = constrain(addr1, 0, 255);
    String addr = String(addr1);
    addr = "D" + addr;
    char s[addr.length() + 2];
    addr.toCharArray(s, addr.length() + 1);
    _serial.flush();
    _serial.write(s);
    _serial.write("\r");
    if (_print)
    {
        Serial.write("Setting I2C address: ");
        Serial.println(addr1);
    }
}

void setPos(int pos1)
{
    if (_constraint)
    {
        if (_print)
        {
            printOutput(0);
            int curr_pos = getPos();
        }
```java
printOutput(1);
if (pos1 > _max)
    pos1 = _max;
else if (pos1 < _min)
    pos1 = _min;
}
else
{
    int curr_pos = getPos();
    if (pos1 > _max)
        pos1 = _max;
    else if (pos1 < _min)
        pos1 = _min;
}
String pos = String(pos1);
pos = "P"+pos;
char s[pos.length()+2];
pos.toCharArray(s, pos.length()+1);
_serial.flush();
_serial.write(s);
_serial.write("\r");
if (_print)
{
    Serial.write("Setting motor position: ");
    Serial.println(pos1);
}
}

void setAngleDeg(float angle1)
{
    angle1 *= 5;
    int angle = (int) ceil(angle1);
    if (_constraint)
    {
        if (_print)
        {
            printOutput(0);
            int curr_pos = getPos();
```
    printOutput(1);
    if (angle > _max)
        angle = _max;
    else if (angle < _min)
        angle = _min;
}
else
{
    int curr_pos = getPos();
    if (angle > _max)
        angle = _max;
    else if (angle < _min)
        angle = _min;
}
}
String angle2 = String(angle);
angle2 = "P" + angle2;
char s[angle2.length() + 2];
angle2.toCharArray(s, angle2.length() + 1);
_serial.flush();
_serial.write(s);
_serial.write("\r");
if (_print)
{
    Serial.write("Setting motor position: ");
    Serial.println(angle);
}

void setAngleRad(float angle1)
{
    angle1 *= 286.6242;
    int angle = (int) ceil(angle1);
    if (_constraint)
    {
        if (_print)
        {
            printOutput(0);
            int curr_pos = getPos();
        }
```java
printOutput(1);
if (angle > _max)
    angle = _max;
else if (angle < _min)
    angle = _min;
else
{
    int curr_pos = getPos();
    if (angle > _max)
        angle = _max;
    else if (angle < _min)
        angle = _min;
}

String angle2 = String(angle);
angle2 = "P"+angle2;
char s [angle2.length() +2];
angle2.toCharArray(s, angle2.length() +1);
_serial.flush();
_serial.write(s);
_serial.write("\r");
if (_print)
{
    Serial.write("Setting motor position: ");
    Serial.println(angle);
}

void setConstraint()
{
    _constraint = 1;
}

void setConstraint(float min, float max)
{
    _constraint = 1;
    _min = (long)min;
    _max = (long)max;
}
```
```cpp
void setConstraintRad(float min, float max)
{
    _constraint = 1;
    _min = (long)(min*286.6242);
    _max = (long)(max*286.6242);
}

void setConstraintDeg(float min, float max)
{
    _constraint = 1;
    _min = (long)(min*5);
    _max = (long)(max*5);
}

void removeConstraint()
{
    _constraint = 0;
}

void printOutput(bool choice)
{
    if (choice)
        _print = 1;
    else if (!choice)
        _print = 0;
}

int getPos()
{
    int position;
    bool neg = 0;
    String inString = "";
    _serial.flush();
    _serial.write("P\r");
    _serial.listen();
    while (_serial.available()<=3); //was 2
    while (!_serial.available())
    {
        char inChar = _serial.read(); //was int
        if (isDigit(inChar))
            inString += inChar;
        if (inChar=='-')
            neg = 1;
```
if (inChar==’\r’)
{
    position = inString.toInt();
    if (neg)
        position *= -1;
    if (_.print)
        Serial.println(position);
    return position;
}

} float getAngleDeg()
{
    float angle;
    bool neg = 0;
    String inString = "";
    _.serial.flush();
    _.serial.write("P\r");
    _.serial.listen();
    while (_.serial.available()<=2);
    while (_.serial.available())
    {
        int inChar = _.serial.read();
        if (isDigit(inChar))
            inString += (char)inChar;
        if (inChar==’-’)
            neg = 1;
        if (inChar==’\r’)
        {
            angle = (inString.toInt())/5;
            if (neg)
                angle *= -1;
            if (_.print)
                Serial.println(angle);
            return angle;
        }
    }
}

float getAngleRad()
{  
    float angle;
    bool neg = 0;
    String inString = "";
    _serial.flush();
    _serial.write("P\r");
    _serial.listen();
    while (_serial.available()<=2);
    while (_serial.available())
    {
        int inChar = _serial.read();
        if(isDigit(inChar))
            inString += (char)inChar;
        if(inChar=='-')
            neg = 1;
        if(inChar=='\r')
        {
            angle = (inString.toInt()) / 286.6242;
            if(neg)
                angle *= -1;
            if (_print)
                Serial.println(angle);
            return angle;
        }
    }
}
int getSpeed()
{
    int speed;
    bool neg = 0;
    String inString = "";
    _serial.flush();
    _serial.write("M\r");
    _serial.listen();
    while (_serial.available()<=2);
    while (_serial.available())
    {
        int inChar = _serial.read();
        if(isDigit(inChar))
        {  
            inString += (char)inChar;
        }  
    }  
}
inString += (char)inChar;
if (inChar=='-')
neg = 1;
if (inChar=='\r')
{
    speed = inString.toInt();
    if (neg)
        speed *= -1;
    if (_.print)
        Serial.println(speed);
    return speed;
}
}
}
int getRotSpeed()
{
    int speed;
    bool neg = 0;
    String inString = "";
    _serial.flush();
    _serial.write("S\r");
    _serial.listen();
    while (_serial.available()<=2);
    while (_serial.available())
    {
        int inChar = _serial.read();
        if (isDigit(inChar))
            inString += (char)inChar;
        if (inChar=='-')
            neg = 1;
        if (inChar=='\r')
        {
            speed = inString.toInt();
            if (neg)
                speed *= -1;
            if (_.print)
                Serial.println(speed);
            return speed;
        }
```cpp
int getI2CAddr()
{
    int addr;
    bool neg = 0;
    String inString = "";
    Serial.flush();
    Serial.write("E\r");
    Serial.listen();
    while(Serial.available()<=2);
    while (Serial.available())
    {
        int inChar = Serial.read();
        if(isDigit(inChar))
            inString += (char)inChar;
        if(inChar=='-')
            neg = 1;
        if(inChar=='\r')
        {
            addr = inString.toInt();
            if(neg)
                addr *= -1;
            if(_print)
                Serial.println(addr);
            return addr;
        }
    }
}

int getSFGain()
{
    int gain;
    bool neg = 0;
    String inString = "";
    Serial.flush();
    Serial.write("A\r");
    Serial.listen();
    while(Serial.available()<=2);
    while (Serial.available())
    {
```
```
int inChar = _serial.read();
if (isDigit(inChar))
    inString += (char)inChar;
if (inChar=='-')
    neg = 1;
if (inChar=='\r')
{
    gain = inString.toInt();
    if (neg)
        gain *= -1;
    if (_print)
        Serial.println(gain);
    return gain;
}

int getP()
{
    int gain;
    bool neg = 0;
    String inString = "";
    _serial.flush();
    _serial.write("Br");
    _serial.listen();
    while (_serial.available()<=2);
    while (_serial.available())
{
    int inChar = _serial.read();
    if (isDigit(inChar))
        inString += (char)inChar;
    if (inChar=='-')
        neg = 1;
    if (inChar=='\r')
{
        gain = inString.toInt();
        if (neg)
            gain *= -1;
        if (_print)
    Serial.println(gain);
    return gain;
}
}
}
}
int getI()
{
    int gain;
    bool neg = 0;
    String inString = "";
    _serial.flush();
    _serial.write("C\r");
    _serial.listen();
    while (_serial.available()<=2);
    while (_serial.available())
    {
        int inChar = _serial.read();
        if(isDigit(inChar))
            inString += (char)inChar;
        if(inChar=='-')
            neg = 1;
        if(inChar=='\r')
        {
            gain = inString.toInt();
            if(neg)
                gain *= -1;
            if(_print)
                Serial.println(gain);
            return gain;
        }
    }
}
int getD()
{
    int gain;
    bool neg = 0;
    String inString = "";
    _serial.flush();
    _serial.write("D\r");
```cpp
_serial.listen();
while (_serial.available()<=2);
while (_serial.available())
{
    int inChar = _serial.read();
    if (isDigit(inChar))
        inString += (char)inChar;
    if (inChar=='-')
        neg = 1;
    if (inChar=='
        
    }
    gain = inString.toInt();
    if (neg)
        gain *= -1;
    if (_print)
        Serial.println(gain);
    return gain;
}
}
private:
SoftwareSerial _serial;
bool _constraint, _print;
long _max, _min;
};
#endif

Listing 3: Rhino Motor Library - Rhino.h

/*
 * MatrixMath.h Library for Matrix Math
 *
 * Created by Charlie Matlack on 12/18/10.
 * Modified from code by RobH45345 on Arduino Forums, algorithm from
 * Modified to work with Arduino 1.0/1.5 by randomvibe & robtiilaart
 * Made into a real library on GitHub by Vasilis Georgitzikis (tzikis)
 * so that it's easy to use and install (March 2015)
 */

138
#ifndef MatrixMath_h
#define MatrixMath_h

#if defined(ARDUINO) && ARDUINO >= 100
    #include "Arduino.h"
#else
    #include "WProgram.h"
#endif

class MatrixMath
{
    public:
        //MatrixMath();
        void Print(float* A, int m, int n, String label);
        void Copy(float* A, int n, int m, float* B);
        void Multiply(float* A, float* B, int m, int p, int n, float* C);
        void Add(float* A, float* B, int m, int n, float* C);
        void Subtract(float* A, float* B, int m, int n, float* C);
        void Transpose(float* A, int m, int n, float* C);
        void Scale(float* A, int m, int n, float k);
        int Invert(float* A, int n);
};

extern MatrixMath Matrix;
#endif

Listing 4: MatrixMath Library - MatrixMath.h

/*
 * MatrixMath.cpp Library for Matrix Math
 * 
 * Created by Charlie Matlack on 12/18/10.
 * 
 * Modified from code by RobH45345 on Arduino Forums, algorithm from
 * 
 */

#include "MatrixMath.h"

#define NR_END 1
MatrixMath Matrix; // Pre-instantiate

// Matrix Printing Routine
// Uses tabs to separate numbers under assumption printed float width won't cause problems

void MatrixMath::Print(float * A, int m, int n, String label)
{
    // A = input matrix (m x n)
    int i, j;
    Serial.println();
    Serial.println(label);
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            Serial.print(A[n * i + j], 4);
            Serial.print("t");
        }
        Serial.println();
    }
}

void MatrixMath::Copy(float * A, int n, int m, float * B)
{
    int i, j, k;
    for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
    {
        B[n * i + j] = A[n * i + j];
    }
}

// Matrix Multiplication Routine
// C = A*B
void MatrixMath::Multiply(float * A, float * B, int m, int p, int n, float * C)
{
    // A = input matrix (m x p)
    // B = input matrix (p x n)
// m = number of rows in A
// p = number of columns in A = number of rows in B
// n = number of columns in B
// C = output matrix = A*B (m x n)
int i, j, k;
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
    {
        C[n * i + j] = 0;
        for (k = 0; k < p; k++)
    }

// Matrix Addition Routine
void MatrixMath::Add(float* A, float* B, int m, int n, float* C)
{
    // A = input matrix (m x n)
    // B = input matrix (m x n)
    // m = number of rows in A = number of rows in B
    // n = number of columns in A = number of columns in B
    // C = output matrix = A+B (m x n)
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
}

// Matrix Subtraction Routine
void MatrixMath::Subtract(float* A, float* B, int m, int n, float* C)
{
    // A = input matrix (m x n)
    // B = input matrix (m x n)
    // m = number of rows in A = number of rows in B
    // n = number of columns in A = number of columns in B
    // C = output matrix = A-B (m x n)
    int i, j;
for (i = 0; i < m; i++)
for (j = 0; j < n; j++)
}

// Matrix Transpose Routine
void MatrixMath::Transpose(float *A, int m, int n, float *C)
{
    // A = input matrix (m x n)
    // m = number of rows in A
    // n = number of columns in A
    // C = output matrix = the transpose of A (n x m)
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            C[m * j + i] = A[n * i + j];
}

void MatrixMath::Scale(float *A, int m, int n, float k)
{
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)

// Matrix Inversion Routine
// * This function inverts a matrix based on the Gauss Jordan method.
// * Specifically, it uses partial pivoting to improve numeric stability.
// * The algorithm is drawn from those presented in
// * The function returns 1 on success, 0 on failure.
// * NOTE: The argument is ALSO the result matrix, meaning the input matrix is REPLACED
int MatrixMath::Invert(float *A, int n)
{
    // A = input matrix AND result matrix
    // n = number of rows = number of columns in A (n x n)
```java
int pivrow; // keeps track of current pivot row
int k, i, j; // k: overall index along diagonal; i: row index; j: col index
int pivrows[n]; // keeps track of rows swaps to undo at end
float tmp; // used for finding max value and making column swaps

for (k = 0; k < n; k++)
{
    // find pivot row, the row with biggest entry in current column
    tmp = 0;
    for (i = k; i < n; i++)
    {
        if (abs(A[i * n + k]) >= tmp) // 'Avoid using other functions inside abs()'?
        {
            tmp = abs(A[i * n + k]);
            pivrow = i;
        }
    }

    // check for singular matrix
    if (A[pivrow * n + k] == 0.0f)
    {
        Serial.println("Inversion failed due to singular matrix");
        return 0;
    }

    // Execute pivot (row swap) if needed
    if (pivrow != k)
    {
        // swap row k with pivrow
        for (j = 0; j < n; j++)
        {
            tmp = A[k * n + j];
            A[k * n + j] = A[pivrow * n + j];
            A[pivrow * n + j] = tmp;
        }
        pivrows[k] = pivrow; // record row swap (even if no swap happened)

        tmp = 1.0f / A[k * n + k]; // invert pivot element
    }
}
```
\[ A[k \times n + k] = 1.0f; \quad // \text{This element of input matrix becomes result matrix} \]

// Perform row reduction (divide every element by pivot)
for (j = 0; j < n; j++)
{
    A[k \times n + j] = A[k \times n + j] \times \text{tmp};
}

// Now eliminate all other entries in this column
for (i = 0; i < n; i++)
{
    if (i != k)
    {
        \text{tmp} = A[i \times n + k];
        A[i \times n + k] = 0.0f; \quad // \text{The other place where in matrix becomes result mat}
        for (j = 0; j < n; j++)
        {
            A[i \times n + j] = A[i \times n + j] - A[k \times n + j] \times \text{tmp};
        }
    }
}

// Done, now need to undo pivot row swaps by doing column swaps in reverse order
for (k = n - 1; k >= 0; k--)
{
    if (\text{pivrows}[k] != k)
    {
        for (i = 0; i < n; i++)
        {
            \text{tmp} = A[i \times n + k];
            A[i \times n + k] = A[i \times n + \text{pivrows}[k]];  
            A[i \times n + \text{pivrows}[k]] = \text{tmp};
        }
    }
}
return 1;
Listing 5: MatrixMath Library - MatrixMath.cpp

```cpp
/*
 * Library for controlling Pololu 20mm 12V DC Motor w/ Encoder:
 * https://www.pololu.com/product/3497/
 * Created by Michael Lastinger
 * Motor: @12V 29 RPM, 350 oz-in (25 kg-cm); gearbox torque limit is 70 oz-in (5 kg-cm)
 * Encoder: 20 counts per revolution (cpr)
 * To compute the cpr of the gearbox output shaft, multiply the gear ratio by 20
 * Gear ratio = 488.28125:1
 */

#ifndef BranchMotor_h
#define BranchMotor_h

#include "Arduino.h"
#include "Encoder.h"
#include "PID_v1.h"

class BranchMotor
{
public:
    BranchMotor() = delete; // remove default constructor
    // Constructor
    // Recall - ":" denotes initialization list
    BranchMotor(int8_t dirPin, int8_t pwmPin, int pwmVal, volatile int32_t *pos):
        dir(dirPin), pwm(pwmPin), pwmVal(pwmVal), encCnt(pos), mtrPID(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT){
        pinMode(dir, OUTPUT);
        pinMode(pwm, OUTPUT);
    }

    void goToPos(int32_t p){
        int32_t initPos = *encCnt; // initial encoder position
        setpoint = p;
        if ((initPos - setpoint) < 0){
```
forward();

} else if ((initPos - setpoint) > 0) {
    backward();
}
else{
    stop();
}

mtrPID.SetMode(AUTOMATIC);
// PID controller:
while (moving) {
    input = (double)*encCnt;
    Serial.println(input);
    error = setpoint - input;
    mtrPID.Compute(); // update PID output
    // Serial.print("PID OUTPUT (PWM) : ");
    // Serial.println(output);
    if (error >= 0) {
        forward();
    }
    if (error < 0 ){
        backward();
    }
    analogWrite(pwm, output); // use output of PID controller to control motor speed (motor slows as it reaches setpoint)
    if (abs(error) <= 25){ // within 25 cnts of setpoint
        lowErrCnt++;
    }
    if (lowErrCnt >= 100){ // error is consistently low \rightarrow setpoint "reached"
        stop(); // stop motor (turn "off")
        lowErrCnt = 0; // reset low error cnt
    }
}

void forward() {
    analogWrite(pwm, pwm_Val);
    digitalWrite(dir, LOW);
    moving = true;
void backward()
{
    analogWrite(pwm, pwm_Val);
    digitalWrite(dir, HIGH);
    moving = true;
}

void stop()
{
    analogWrite(pwm, LOW);
    digitalWrite(dir, HIGH);
    moving = false;
}

int32_t getRev()
{ // returns current # of revolutions of motor shaft (position)
    mtrPos = *encCnt/9766; // gear_ratio*cpr = 488.28125*20 = 9765.625
    return mtrPos;
}

private:
    int8_t dir;
    int8_t pwm;
    int pwm_Val;
    // Encoder enc;
    bool moving = false;
    int32_t mtrPos;
    volatile int32_t *encCnt;
    double setpoint, input, output, error;
    // best (no tendon attached) Kp=0.28 (PWM=2.80; w/in 8 cnts)
    // best on TREE Kp=0.51
    double Kp=0.51, Ki=0, Kd=0;
    PID mtrPID;
    int lowErrCnt = 0;

}; //end of class def

Listing 6: Branch Motor Library - BranchMotor.h
/*********************************************/
* Modified by Michael Lastinger
* for TREE Branch actuation (closed-loop control)
*********************************************/

#ifndef PID_v1_h
#define PID_v1_h
#define LIBRARY_VERSION 1.1.1

class PID
{
  public:

  // Constants used in some of the functions below
  #define AUTOMATIC 1
  #define MANUAL 0
  #define DIRECT 0
  #define REVERSE 1

  // Commonly used functions
  *********************************************/
  PID(double*, double*, double*, // * constructor. links the PID to the
       double, double, double, int); // Setpoint. Initial tuning parameters are also set here

  void SetMode(int Mode); // * sets PID to either Manual (0) or Auto (non-0)

  bool Compute(); // * performs the PID calculation. It should be
                   // called every time loop() cycles. ON/OFF and
                   // calculation frequency can be set using SetMode

  void SetOutputLimits(double, double); // clamps the output to a specific range. 0–255 by default, but
// it's likely the user will want to change this depending on
// the application

// available but not commonly used functions

******************************************************************************
void SetTunings(double, double, // * While most users will set the tunings
once in the
double); // constructor, this function gives the
user the option
      // of changing tunings during runtime for
Adaptive control
void SetControllerDirection(int); // * Sets the Direction, or "Action" of the
controller. DIRECT
      // means the output will increase when error is positive.
REVERSE
      // means the opposite. it's very unlikely that this will be
needed
      // once it is set in the constructor.
void SetSampleTime(int); // * sets the frequency, in Milliseconds,
with which
      // the PID calculation is performed.
default is 100

// Display functions
******************************************************************************
double GetKp(); // These functions query the pid for internal values.
double GetKi(); // they were created mainly for the pid front-end,
double GetKd(); // where it's important to know what is actually
int GetMode(); // inside the PID.
int GetDirection(); //

private:
void Initialize();

double dispKp; // * we'll hold on to the tuning parameters in user-entered
double dispKi; // format for display purposes
double dispKd; //
double kp; // *(P)roportional Tuning Parameter
double ki; // *(I)ntegral Tuning Parameter
double kd; // *(D)erivative Tuning Parameter

int controllerDirection;

double *myInput; // * Pointers to the Input, Output, and Setpoint variables
double *myOutput; // This creates a hard link between the variables and the
double *mySetpoint; // PID, freeing the user from having to constantly tell us
// what these values are. with pointers we’ll just know.

unsigned long lastTime;
double iTerm, lastInput, lastError;

unsigned long SampleTime;
double outMin, outMax;
bool inAuto;
}
#endif

Listing 7: PID Library - PID_v1.h

/* ****************************************************************************
 * Arduino PID Library – Version 1.1.1
 * by Brett Beauregard <br3ttb@gmail.com> brettbeauregard.com
 * Modified by Michael Lastinger for TREE Branch actuation (closed–loop control)
 * This Library is licensed under a GPLv3 License
 *****************************************************************************/
#if ARDUINO >= 100
 #include "Arduino.h"
#else
 #include "WProgram.h"
#endif

150
#include <PID_v1.h>

/^Constructor (...)*/
* The parameters specified here are those for which we can’t set up
* reliable defaults, so we need to have the user set them. /

PID::PID(double* Input, double* Output, double* Setpoint,
         double Kp, double Ki, double Kd, int ControllerDirection)
{

    myOutput = Output;
    myInput = Input;
    mySetpoint = Setpoint;
    inAuto = false;

    PID::SetOutputLimits(0, 150); // default output limit corresponds to
                                   // the arduino pwm limits

    SampleTime = 100;            // default Controller Sample Time is 0.1 seconds

    PID::SetControllerDirection(ControllerDirection);
    PID::SetTunings(Kp, Ki, Kd);

    lastTime = millis() - SampleTime;
}

/^Compute()*/
* This, as they say, is where the magic happens. This function should be called
* every time “void loop()” executes. the function will decide for itself whether
* a new
* pid Output needs to be computed. returns true when the output is computed,
* false when nothing has been done. /

bool PID::Compute()
{
    if (!inAuto) return false;
    unsigned long now = millis();
    unsigned long timeChange = (now - lastTime);

if (timeChange >= SampleTime)
{
    /*Compute all the working error variables*/
    double input = *myInput;
    double error = *mySetpoint - input;
    ITerm += (ki * error);
    if (ITerm > outMax) ITerm = outMax;
    else if (ITerm < outMin) ITerm = outMin;
    // double dInput = (input - lastInput); // should this be dError? see next line 
    ...
    double dError = (error - lastError);

    /*Compute PID Output*/
    double output = kp * error + ITerm - kd * dError; // dError was dInput

    if (output < 0) output = abs(output); // ensure negative values don't get forced to 0
    if (output > outMax) output = outMax;
    else if (output < outMin) output = outMin;
    *myOutput = output;

    /*Remember some variables for next time*/
    // lastInput = input;
    lastError = error;
    lastTime = now;
    return true;
}
else return false;

/* SetTunings(...)*********************************************************************/
/* This function allows the controller's dynamic performance to be adjusted. */
/* It's called automatically from the constructor, but tunings can also */
/* be adjusted on the fly during normal operation */
*******************************************************************************/
void PID::SetTunings(double Kp, double Ki, double Kd)
{
    if (Kp < 0 || Ki < 0 || Kd < 0) return;
double SampleTimeInSec = ((double)SampleTime)/1000;
kp = Kp;
ki = Ki * SampleTimeInSec;
kd = Kd / SampleTimeInSec;

if (controllerDirection == REVERSE)
{
    kp = (0 - kp);
    ki = (0 - ki);
    kd = (0 - kd);
}

/* SetSampleTime(...) *********************************************/
/* sets the period, in Milliseconds, at which the calculation is performed */
void PID::SetSampleTime(int NewSampleTime)
{
    if (NewSampleTime > 0)
    {
        double ratio = (double)NewSampleTime
                        / (double)SampleTime;
        ki *= ratio;
        kd /= ratio;
        SampleTime = (unsigned long)NewSampleTime;
    }

/* SetOutputLimits(...) *********************************************/
/* This function will be used far more often than SetInputLimits. while 
* the input to the controller will generally be in the 0–1023 range (which is 
* the default already,) the output will be a little different. maybe they’ll 
* be doing a time window and will need 0–8000 or something. or maybe they’ll 
* want to clamp it from 0–125. who knows. at any rate, that can all be done 
* here. */
void PID::SetOutputLimits(double Min, double Max)
if (Min >= Max) return;
outMin = Min;
outMax = Max;

if (inAuto)
{
    if (MYOutput > outMax) MYOutput = outMax;
    else if (MYOutput < outMin) MYOutput = outMin;
    if (ITerm > outMax) ITerm = outMax;
    else if (ITerm < outMin) ITerm = outMin;
}

void PID::SetMode(int Mode)
{
    bool newAuto = (Mode == AUTOMATIC);
    if (newAuto == !inAuto)
    { /* we just went from manual to auto */
        PID::Initialize();
    }
    inAuto = newAuto;
}

void PID::Initialize()
{
    ITerm = *MYOutput;
    lastInput = *MYInput;
    if (ITerm > outMax) ITerm = outMax;
else if (ITerm < outMin) ITerm = outMin;

} /* SetControllerDirection(...)
 * The PID will either be connected to a DIRECT acting process (+Output leads
to +Input) or a REVERSE acting process(+Output leads to −Input.) we need to
* know which one, because otherwise we may increase the output when we should
* be decreasing. This is called from the constructor.
*/

void PID::SetControllerDirection(int Direction)
{
    if (inAuto && Direction != controllerDirection)
    {
        kp = (0 − kp);
        ki = (0 − ki);
        kd = (0 − kd);
    }
    controllerDirection = Direction;
}

/* Status Funcions
 * Just because you set the Kp=−1 doesn’t mean it actually happened. these
 * functions query the internal state of the PID. they’re here for display
 * purposes. this are the functions the PID Front−end uses for example
 */

double PID::GetKp(){ return dispKp; }

double PID::GetKi(){ return dispKi;}

double PID::GetKd(){ return dispKd;}

int PID::GetMode(){ return inAuto ? AUTOMATIC : MANUAL;}

int PID::GetDirection(){ return controllerDirection;}

Listing 8: PID Library - PID_v1.cpp
Appendix C  TREE Keyboard Controls

NOTE: Arduino Serial Monitors must be set to “Insert Newline Character” and “>” must precede any message/command intended for the Arduino Due (Tendril branches). Keyboard templates courtesy of Teaching Ideas [39].
Bibliography


