DESIGN, ANALYSIS, AND CONTROL OF A CABLE-SUSPENDED ROBOT FOR LARGE-SCALE ADDITIVE MANUFACTURING

A Dissertation

by

IVAN CORTES

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2022

Major Subject: Mechanical Engineering

ABSTRACT

Additive manufacturing (AM) and cable-suspended robots (CSRs) are two areas of technology that evolved rapidly in the last decades. This dissertation explores combining the two ideas to create a cable-suspended robot for large-scale additive manufacturing. AM technologies allow for the creation of products by combining material, layer by layer, in an automated process. This capability has been introduced in numerous industries with the promise of increasing productivity and lowering operating costs. However, AM has typically been limited to small physical dimensions because AM processes require precise control of materials. CSRs may facilitate AM at larger scales by providing a low-cost, flexible method for position control in some types of AM, such as fused deposition modeling (FDM) or directed energy deposition (DED). However, position control at large scales is challenged by environmental factors, such as external disturbances and structural imperfections in the AM machine. An analysis of the CSR configurations and control strategies is useful in determining whether large-scale AM is a practical application for this kind of robot. This dissertation presents (1) the general design concept and analyses of a CSR for large-scale AM, (2) closed-loop control strategies in presence of external disturbances, and (3) experimental results of a laboratory-scale prototype. The complete work suggests that a CSR with closed-loop control may be used for precise position control even withstanding some structural imperfections. However, the control strategies presented here leave room for improvement of the CSR design as it remains vulnerable to high-frequency external disturbances.

DEDICATION

I completed this work thinking of my family. They always supported and encouraged me, and I am forever grateful for that. I dedicate this dissertation to them first.

I must also mention my great friends who were always by my side. I lived through this experience with them. Thank you fellas.

ACKNOWLEDGMENTS

I would like to give special thanks and acknowledgment to my research advisor for his constant support and guidance in the last several years. His advice will forever guide my work as a researcher and engineer. Thank you, Dr. Kim, for all of your help.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Won-jong Kim, Prabhakar Pagilla, and Bruce Tai of the J. Mike Walker ’66 Department of Mechanical Engineering and Professor Mehrdad Ehsani of the Department of Electrical and Computer Engineering. Professor Swaroop Darbha, also of the J. Mike Walker ’66 Department of Mechanical Engineering, supported this work as an examiner in the student’s final examination.

All work conducted for this dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by fellowship awards from the J. Mike Walker ’66 Department of Mechanical Engineering.

NOMENCLATURE

2D	Two dimensional
3D	Three dimensional
AC	Alternating current
ADC	Analog-to-digital converter
AM	Additive manufacturing
CCM	Cable-connection matrix
CSR	Cable-suspended robot
CV	Computer vision
DAQ	Data acquisition
DC	Direct current
DED	Directed energy deposition
DLP	Digital light processing
DOF	Degree of freedom
EBAM	Electron beam additive manufacturing
FDM	Fused deposition modeling
FPS	Frames per second
IMU	Inertial measurement unit
PC	Personal computer
PID	Proportional integral derivative
PPR	Pulses per revolution
PWM	Pulse-width modulation

SEP	Static equilibrium point
SLA	Stereolithography
SLS	Selective laser sintering
UV	Ultraviolet

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Automation is often touted as the next step of evolution for many industrial processes, and additive manufacturing (AM) is one example of a process that relies on automation to precisely control the flow of materials. This dissertation analyzes and experiments with one solution for AM—the use of a cable-suspended robot (CSR). Specifically, the goal of this research is to develop a suitable configuration and control strategy for a CSR that can be used for large-scale, extrusion- or deposition-based AM. The research activities included in this work are hardware and controller design, mathematical analysis, and experimentation with a prototype of the envisioned CSR.

This research is at the intersection of two technology ideas: AM and CSRs. Both ideas have been studied extensively in various forms. However, attempting to use a CSR for AM presents significant engineering challenges. In particular, a CSR may not be able to provide the structural rigidity required for some AM processes. Furthermore, a large-scale CSR is likely to experience structural imperfections or external disturbances. Such disturbances are rarely considered in laboratory-scale CSRs and are specifically addressed in this work.

The remainder of this chapter gives a brief review of AM, CSRs, and related concepts while Chapter 2 summarizes the contributions of this dissertation. Chapter 3 details the analyses and simulations performed. Chapter 4 describes the experimental prototype and accompanying equipment. Chapter 5 presents the experimental results and their significance. Finally, Chapter 6 offers the conclusions reached through this work.

1.1. Additive Manufacturing

AM is a technical term that encompasses many different technologies. Therefore, it is important to distinguish the technologies and identify those that this research addresses. In addition, the proposed emphasis on large-scale manufacturing requires a consideration of environmental disturbances not usually present in small, controlled settings. The functional and performance requirements of AM at a large scale set the expectations for a CSR used in this application and guide the scope of the research.

1.1.1. Technologies and Applications

Popular science and media publications often use the term three-dimensional (3-D) printing when discussing AM technologies. The idea is that "3D printing" allows for the creation of objects in a 3-D space, as opposed to the usual printing of two-dimensional (2-D) images on a flat surface. AM is a technical term that includes various technologies and processes, each specialized for use with different materials. The American Society for Testing Materials defines seven AM process categories in ISO/ASTM 52900:2015. These are: material extrusion, material jetting, binder jetting, directed energy deposition, powder bed fusion, sheet lamination, and vat photopolymerization. A cable suspended-robot may be impractical for some AM processes due to the handling requirements of bulk materials. A brief overview of some common AM technologies reveals how they differ from traditional manufacturing processes and how CSRs could play an important role.

Many traditional manufacturing processes are subtractive in nature. Some examples are cutting and milling, in which material is removed from a larger piece to reveal a final part. Individual parts can then be combined using fasteners or small amounts of binding material (for example, weld or glue). These steps work well, since bulk material can be transported in large pieces and a variety of cutting tools have been developed to remove excess material precisely. However, the process of removing material might be seen as wasteful, constrained, or expensive in some cases. Waste occurs when much of the material being removed is not used for the final product. The shape of the final part is constrained by the dimensions of the bulk material and, also, by the configuration of the cutting tool. Often, specialized tool setups are created to achieve a desired product shape. The time and energy required to design these tools and perform the material removal means there is an increase in production costs. Experienced designers learn about the common material subtraction processes and use this knowledge to specify parts that are easier to manufacture.

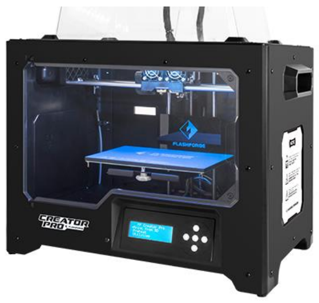
There are other manufacturing processes by which a part is created from a batch of raw material and where only the necessary amount of material is used. Some examples are injection molding, casting, and pressing. In these cases, the raw material is handled in fluid or powder form so that it can take shape of a pre-made container, or mold. Once inside the mold, the material is solidified. The mold is carefully designed so that the lose material will fill every void, and the material flow is closely controlled throughout the process. These methods can be very efficient for mass production, since molds can be duplicated and, often, reused. Still, it takes some time to create the mold.

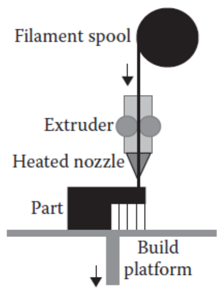
AM is a newer form of production that has developed rapidly in the last few decades. In contrast to subtractive manufacturing and molding, AM technologies create parts by adding small amounts of material at a time, typically in thin layers that resemble

2-D slices of the part [1]. As the new material is added, it binds to the previous material. This fine control of material allows for the creation of complex parts without the need for a mold or significant cutting. Furthermore, the incremental addition of material makes AM processes perfect candidates for automation. In most AM processes, the designer only needs to provide a digital model of the final part, and specialized software determines the necessary machine-level commands to create the part. The end result is that, although material is added slowly, the process is efficient and effective for creating custom parts quickly. Depending on the specific technology, AM can produce quick prototypes or even mass produce full-quality parts. Some of the most common AM technologies, and example applications, are described in the following paragraphs.

Perhaps the most recognizable AM technology is fused deposition modeling (FDM). The working principle of this technology is that a new layer of material is deposited on top of an existing layer, and the two layers fuse together by contact. The material is delivered in a semi-fluid form using an extrusion nozzle, and the material solidifies shortly after being deposited. Therefore, this technology is applicable to materials that can be liquified and extruded in a controlled manner. One of the main functions of an FDM machine is to move the nozzle in precise patterns in order to create the part. Another major function of the machine is to accurately the control material flowrate through the nozzle. The material properties determine how fast new layers can be added while maintaining the structural integrity of the part. In some cases, extra material is deposited to serve as structural support. The extra material is later removed. The overall speed of production, energy requirement, and quality of the part largely depend on the material being used, the nozzle diameter, and the positioning resolution of the machine. Also, the part size is constrained by the size of the FDM machine.

Plastics or plastic composites are popular materials for FDM machines. So-called “desktop 3D printers,” used by many hobbyists today, are a good example (Figure 1). These machines are not much larger than the common paper printer and typically feature three stepper motors to control the *X*-*Y*-*Z* movement of the nozzle, another motor to feed a spool of material into the extruder, and heating elements to melt the material that is being deposited. To use this machine, one would simply create a 3-D model, process it using a special FDM software, and then send the processed data to the machine. The data seen by the machine is usually in the form of G-code, which is a sequential list of standard machine-level commands. Thus, the extrusion process is completely automated. There may be minimal post-processing to remove material fibers or support structures from the part. A variety of hard and flexible materials are available—even some containing wood or metal particles [2]. The various materials are accommodated by adjusting some machine settings, such as the nozzle temperature or extrusion speed.


(a)

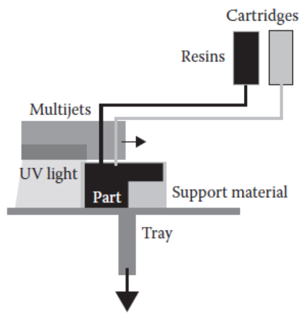


(b)

Figure 1. (a) The Creator Pro machine from flashforge.com. This FDM machine features a build volume of 227×148×150 mm, two extruders, and produces polymer parts with a minimum layer thickness of 0.1 mm. (b) A diagram of the FDM process [2].

If the working principle of FDM were taken to a microscopic level, one might arrive at another AM technology called material jetting. In this method, small amounts of material are deposited using a microscopic jet spray, similar to the way a common printer deposits ink on paper. Depending on the material being used, material jetting could involve the use of an ultraviolet (UV) light to cure the material. Some machines even feature multi-material printing, allowing for the creating of parts with localized mechanical properties. In general, material jetting machines offer much higher resolution than FDM but are also larger and more expensive (Figure 2). Material jetting is excellent for creating production parts with high precision. For example, this technology is often used in dentistry.

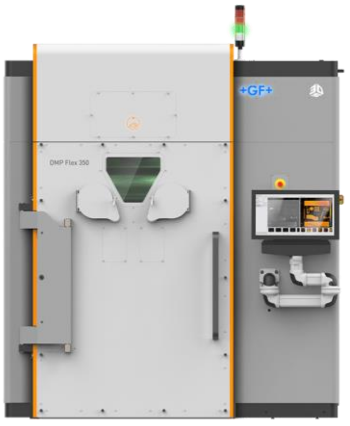
(a)



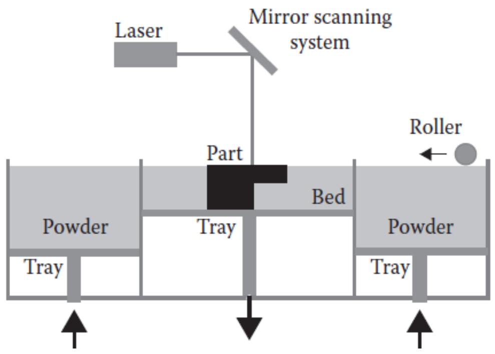
(b)

Figure 2. (a) The Connex3 Objet500 from Stratasys.com. This production-quality material jetting machine offers a build size of 490×390×200 mm, multiple material options, and layers as thin as 16 μm. (b) A diagram of the material jetting process [2].

Another group of AM technologies is powder bed fusion. This includes selective laser sintering (SLS), which can even be used to create metal parts [3]. In this method, a thin layer of material powder is spread evenly over a surface. Then, select areas of the material are sintered using a high-power laser. The sintered areas are solidified, and the non-sintered powder provides structural support. The newly-sintered layer is then lowered slightly, making way for the next layer of material. This process typically produces much higher-quality parts than FDM, but at an increased cost. For successful sintering, and to avoid warping, the temperature and gas constant of the material must be controlled. Even a moderately-sized build volume requires a relatively large machine to create a stable build environment (Figure 3). Nevertheless, SLS machines see extensive use in industry, and are even used for mass production. Multiple parts can be made during a single build process, since the laser can trace complex patterns across the entire build surface in mere seconds. One disadvantage is that the material powder must be available in large batches to be spread over the entire build surface for each new layer, even if the sintered area is small. Therefore, much of the material is unused during each build.



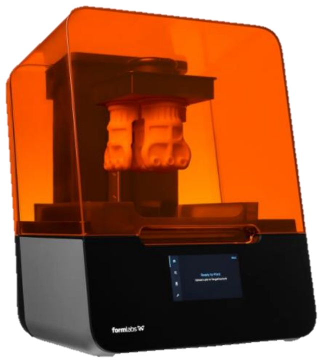
(a)



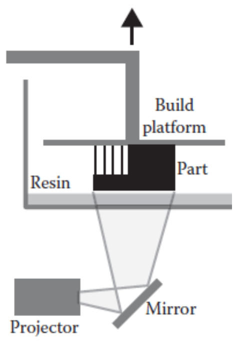
(b)

Figure 3. (a) The DMP Flex 350 machine from 3Dsystems.com. This production-quality SLS machine produces metal parts in a build volume of 275×275×420 mm with a minimum layer thickness of 5 μm. This machine requires compressed air, Argon gas, and water cooling to operate. (b) A diagram of the SLS process [2].

Similar to powder bed fusion, but with liquid material instead of powder, is a group of technologies called vat polymerization. As the name suggests, these technologies feature liquid polymers that are held in vats and then selectively hardened using light energy. This group includes digital light processing (DLP) and stereolithography (SLA) machines (Figure 4). These processes can also require control of environmental variables like temperature or gas content for the material to harden in a predictable manner. The liquid-material requirement limits this process to certain types of polymers in resin form.



(a)

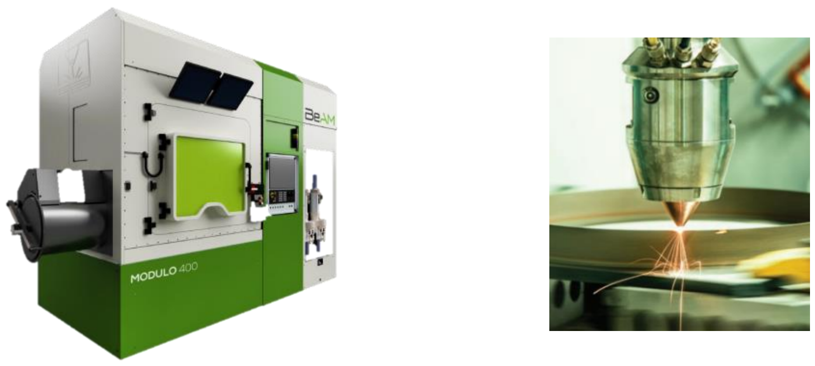


(b)

Figure 4. (a) The Form 3 machine from Formlabs.com. This desktop SLA machine features a 145×145×185 mm build volume and a minimum layer thickness of 25 μm. The laser is housed in the lower compartment and the resin held in a tray in the middle of the machine. (b) A diagram of a DLP machine [2]. In SLA, a laser is used instead of a projector. In some machines, the projector or laser are above the resin.

Another AM technology group is sheet lamination, where material sheets are cut to shape and then fused together, like layers of a cake. The cutting and binding process varies depending on the material type. For example, paper layers can easily be cut to shape using a blade and, then, bound together using glue.

There are many other AM technologies, each suited for different materials and applications. For example, directed energy deposition (DED) is a category of AM technologies that is similar to welding and is commonly used to manufacture metal parts [2, 4]. Typically, metal is fed through a nozzle and melted using a high-power laser near the nozzle outlet. Like FDM, DED depends on accurate positioning of the nozzle and careful control of the material flowrate. The positioning function of the machine can be fulfilled by a gantry mechanism, like the desktop FDM printers, or with robotic manipulators that can approach the working piece at specific angles. Recently, DED has been used to create a variety of functional and artistic metal pieces (Figure 5).



(a) (b)

Figure 5. The Modulo 400 machine from Beam-machines.com (a) and a close view of the deposition nozzle (b). This DED machine is used to produce or repair metal parts. The build volume is 650×400×400 mm and the nozzle diameter is around 1 to 2 mm. The metal is supplied in powder form and is melted at the nozzle using a laser.

In summary, the various AM technologies each provide a way to combine material in a controlled way. Some of the processes introduce the material in small amounts whereas others hold the raw material in large batches. The large batches are typically kept in a controlled environment to preserve the material's properties. On the other hand, those technologies that control the flow of material in small amounts may be easier to implement at a large scale because the material can be protected until the point of delivery (for instance, at a nozzle). Section 1.1.2 gives some examples of how AM technologies were implemented at a large scale, and Section 1.2.3 explains the role that a CSR could play in the build process.

1.1.2. Large-Scale Manufacturing

Large-scale production may not be practical for all of the AM technologies. Some of the technologies require tight environmental controls that are difficult to maintain for large build volumes, and some technologies require the transfer of material in large batches. Consequently, large-scale AM has so far been limited to material deposition methods like FDM or DED. For the purpose of this discussion, large scale manufacturing is loosely defined as any size such that the completed part would require machinery to be transported. For example, a large-scale machine could produce objects the size of a car.

There are several active examples of large-scale FDM. Figure 6 shows an example of

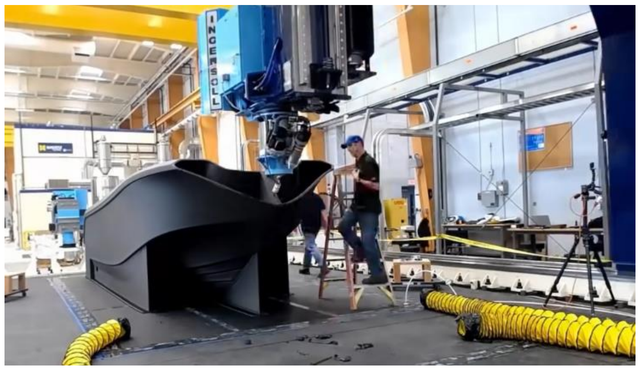


Figure 6. Full-size boat hull created in 72 hours using FDM of polymer material [5].

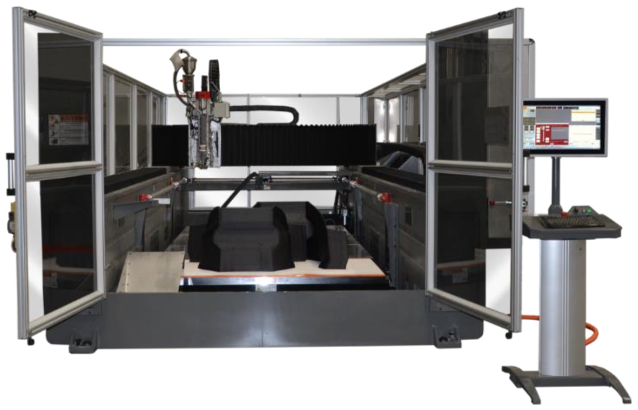


Figure 7. The Big Area Additive Manufacturing machine from E-ci.com. This large-format FDM machine extrudes plastics through a large nozzle (with a 0.2- to 0.4-in diameter) and can create pieces as large as 6.1×2.3×1.8 m.

large-scale plastic FDM by the Advanced Structures and Composites Center at the University of Maine. This image shows a boat hull that was manufactured in 72 hours using the “World’s Largest 3D Printer” [5]. The printer is an FDM system that deposits polymer using a Cartesian positioning system, similar to the positioning system found in desktop printers like the one in Figure 1. A comparable machine, appropriately named the Big Area Additive Manufacturing, was created by the company Cincinnati and uses pelletized thermoplastics to reduce operation costs (Figure 7).

FDM has also been implemented with other materials. Figure 8 shows a robotic arm with a concrete extruder as its end effector. A special concrete mix is pumped to the extruder through a hose. The mix is fluid enough to be pumped and extruded, but it also sets quickly so that it can be layered. According to its creators, this specialized machine can create a small home in 24 hours [6]. Concrete construction is one major intended application for large-scale FDM and aims to increase availability of low-cost housing

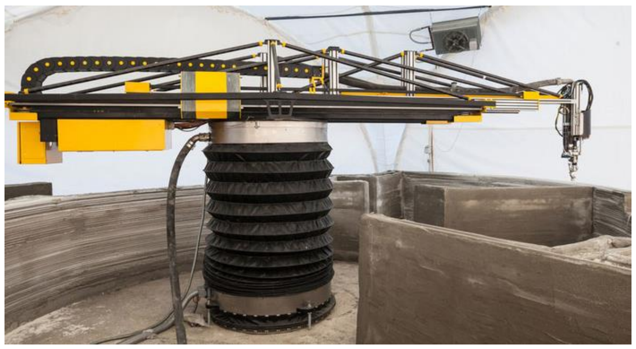


Figure 8. Robotic arm with a concrete extruder developed by Apis Cor [6]

[7−10]. The machines used in concrete extrusion are similar to those used in polymer extrusion because they both operate under the same FDM principle. The main machine functions are to position the extrusion nozzle and control the material flow.

There are many more examples of large-scale FDM applications, often illustrated in popular science publications [11−13]. However, the machines used are usually similar in their construction and operation. One may conclude that implementing FDM at a large scale is a matter of nozzle placement and material transport to the nozzle. However, a large-scale implementation may be more vulnerable to structural imperfections or external disturbances. For example, an outdoor concrete 3-D printer must take into account environmental variables like temperature, humidity, and wind. The FDM machine must be rigid enough to withstand external forces, and the material must be carefully selected so that it solidifies in a predictable manner, even if environmental conditions change.

The DED technologies also have applications at large scales, particularly in metal manufacturing. Figure 9 shows an example of a large metal 3-D printer by Sciaky Incorporated that is capable of printing metal parts over 19 ft in length [14]. The pictured

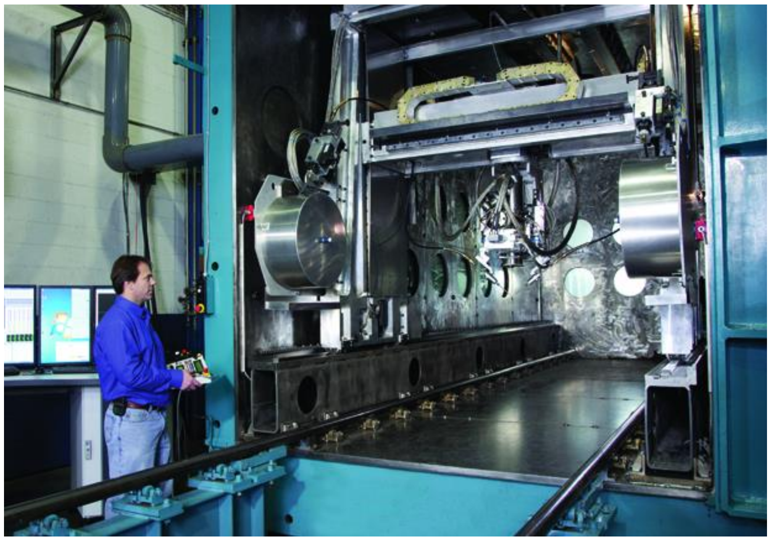


Figure 9. The EBAM 300 machine by Sciaky Incorporated. This machine uses electron beam additive manufacturing (EBAM) technology to create high-strength metal parts. It is claimed to be the largest 3D printer for metal parts that is fit for industrial and commercial use [14].

machine uses a metal wire feed and electron beam additive manufacturing (EBAM) technology to combine the metal. This process allows for the use of high-strength materials, such as titanium and Inconel. Therefore, one common application is the production of aerospace components. According to the Sciaky company, "forgings that used to take 6-12 months to complete can be completed in 2 days with the EBAM 300." Like FDM, DED is scaled to various sizes by implementing appropriate positioning mechanisms for the nozzle.

The use of other AM technologies being used at large scales is limited, though some were attempted [15]. Some barriers for scaling other technologies may be the requirement for environmental controls, such as temperature or gas content, or the large amounts of bulk material used to create each part layer. Nevertheless, the remaining work in this dissertation focuses on extrusion- or deposition-based AM. In these technologies one main function of the AM machine is to move a material nozzle in a controlled manner. This is a task that can be accomplished by many mechanisms, including a CSR. Section 1.2 gives an overview of CSRs and how they were studied in the past, including some existing applications in AM. Section 1.2.3 explains how the AM applications provide constraints that help guide the scope of this dissertation.

1.2. Cable-Suspended Robots

The second technology idea relevant to this dissertation is CSRs. In the large-scale manufacturing examples pictured in Figures 6–9, the position of the extruder or deposition nozzle is controlled by a rigid mechanism, such as a gantry or robotic arm. In contrast, a CSR controls the end effector position using flexible cables. Some limitations arise from the fact that the cables can only pull, and not push, the end effector. However, the flexibility of the cables could mean lower transportation and setup costs compared to a rigid machine. A complete description of CSRs and their application to AM is presented in this section.

1.2.1. Configurations

For this dissertation, a CSR is defined as a robot where the end effector movement is controlled using cables. Furthermore, the only forces acting on the end effector, besides its weight, are the cable tensions. Some rigid structures may be used, but only to anchor the cables. CSRs exist in many configurations, as dictated by the intended use. The distinguishing features of the configurations are the number and placement of cables. These determine the degrees of freedom (DOFs) that the system can effectively control.

Depending on the configuration, the robot may be referred to as a CSR, cable-driven robot, cable-driven mechanism, cable-driven parallel robot, or similar [16−21].

The simplest CSR is a single cable that lowers or lifts the end effector (Figure 10). This simple example illustrates the working principle and some limitations of a CSR. The end effector height is controlled by varying the cable length, typically by a motor and spool that winds the cable. The end effector height can be calculated from the cable length. An important detail is the location of the cable anchor point. The effector cannot be lifted above the anchor point, and the effector will move up and down directly below the anchor point. There is no way to control the left-right movement of the effector once the anchor point is fixed. Thus, this CSR configuration offers 1-DOF control, and any external force perpendicular to the cable tension would cause the end effector to swing uncontrollably. Also, the maximum downward acceleration is that due to gravity since the cable cannot push down on the end effector. This configuration is similar to a construction crane.

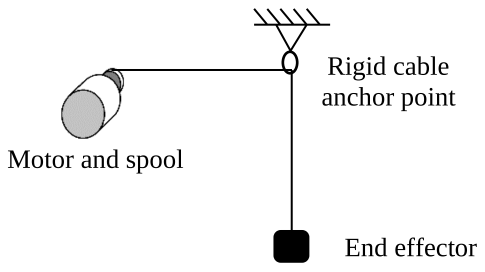


Figure 10. The simplest CSR, with only one cable that raises or lowers an end effector.

A second cable may be added, as in Figure 11, to add another control DOF. The pictured example is from a study where researchers optimized the 2-D force on the point

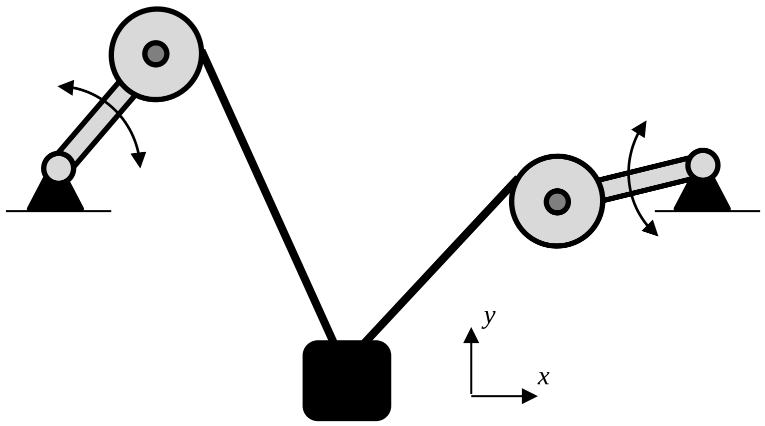


Figure 11. CSR concept that uses two cables and adjustable pulley locations [16].

P by moving the cable anchor points. The anchor points were adjusted by moving the pulleys in the top corners of the machine [16]. Together, the two cables produce a 2-D force on the end effector, and the direction of the force is adjusted by moving the anchor points. Still, the only downward force on the effector is its weight, so the downward acceleration is limited to gravity.

Figure 12 depicts a CSR configuration that allows for 3-DOF position control. The

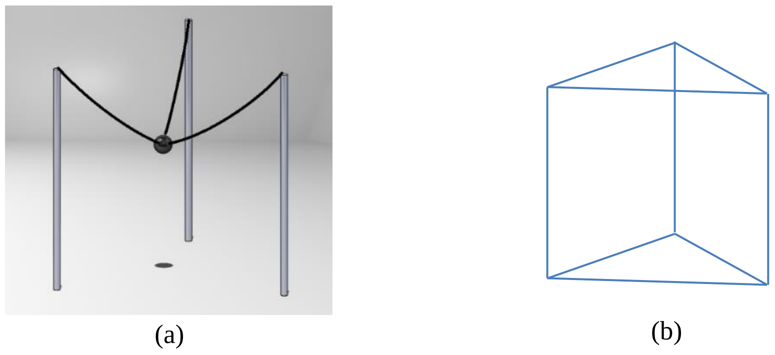


Figure 12. (a) A CSR configuration with three cables and (b) the movement space.

three cables, together, counteract the force of gravity, and the 3-D position of the end effector is uniquely defined by the three cable lengths. If any of the cables is removed, the end effector would swing down to a new position. The downward acceleration is still limited because there is no cable pulling down on the effector. Also, the movement of the effector is limited to the triangular-prism space created by the three anchor point columns.

To improve the control and stiffness of the end effector, some configurations use additional cables. If the additional cables do not increase the control DOFs, they are called redundant. Redundant cables can be used to increase the system stiffness. Figure 13 shows an example of a CSR with six cables [17]. Despite having six cables, the pictured system does not have 6 DOFs. Instead, the cables are arranged to increase the system stiffness and constrain the *θ* orientation of the end effector. Thus, the pictured system has 3-DOF control along coordinates *x*, *y*, and *θ*. There is no control in the *z* direction (the *z* direction is not pictured, but it points out of the page). Cables 2 and 3 pull the end effector down, allowing for downward accelerations larger than gravity. These two cables also increase

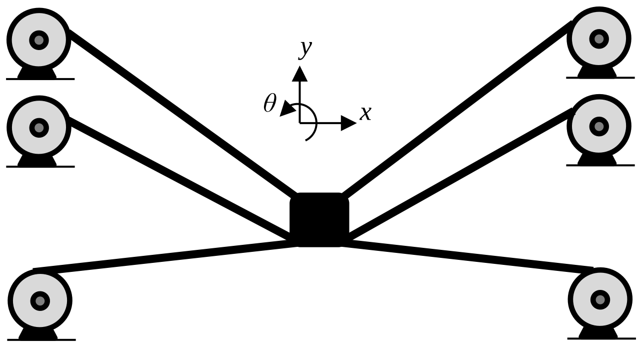


Figure 13. CSR concept with six cables to increase stiffness and improve control [17].

the CSR's stiffness since movements in the positive *y* direction are resisted by the cable tensions. Furthermore, cables 2 and 3 can be considered redundant as they could be removed without losing control DOFs.

In general, adding cables to a CSR configuration increases the control DOFs and stiffness. However, the placement of the cables plays an important role in defining the control directions and allowable movement space. Furthermore, some cables may produce tensions that oppose each other (as in Figure 13), creating moments and shear forces on the end effector. If each cable length is controlled by a separate motor, a larger number of cables also increases the complexity of real-time control, increasing the risk of errors or instabilities. Errors and instabilities can lead to high cable tensions, unwanted shear, or moments. Therefore, one method for selecting a CSR configuration is to use the minimum number of cables such that the desired DOFs and stiffness are achieved. Then, the cables should be anchored such that the desired movement space can be achieved without any cables interfering with each other.

If the end effector is small enough, such that all cables meet relatively close to each other on the effector body, the end effector may be modeled as a point mass. Then, there are 3 possible DOFs, corresponding to 3-D space coordinates of the end-effector body, and at least three cables are needed to provide the position control. If the end effector is large, relative to the dimensions of the CSR, and the cables are attached to the effector in separate locations, then three additional DOFs, corresponding to the orientation of the end-effector, are possible. In this case, at least 6 cables are needed for full control. There are many examples of both point-mass and rigid-body CSRs in the literature [19−24].

1.2.2. Analysis and Control

The sample configurations presented in Section 1.2.1 illustrate some of the control challenges for CSRs. These challenges arise from the flexibility of the cables but are also related to the cable placement. Analyses found in literature are typically accompanied by co-design of the CSR configuration to achieve a specific purpose. As with other robotic systems, one may be interested primarily on the position-control capabilities of a CSR. Some quantities of interest might be the positioning precision, accuracy, power requirement, and robustness to disturbances.

Some analyses focus on the kinematic properties of the CSR configurations [19−20, 25−27]. The kinematics describe how the cable lengths relate to the position and orientation of the end effector. This relationship is often described using trigonometric functions or vector norms. However, these mathematical equations might not hold if there is cable slack or unknown stretch. Whenever a cable is in slack, it no longer plays a role in the movement of the end effector. On the other hand, cable stretch can be difficult to measure, so the effective cable length may not be known. Position control strategies based on kinematics may include efforts to avoid cable slack, estimate the slack, or accurately measure the cable length [17, 21, 28−29]. Cable slack is more difficult to avoid when there are redundant cables, and cables with little stretch can easily transition to slack. All of these factors play a role in the positioning accuracy and resolution of the CSR.

CSR system dynamics, which include the cable tensions and the end effector accelerations, are also an active area of study [21, 24, 30]. For stiff cables, even a small cable stretch can produce a large tensile force. Furthermore, negative cable tensions are not possible, so cable slack is problematic in the sense that it produces no tension. In a real system, there may be residual tensions from cable sagging or vibrations. These phenomena are difficult to predict, so many studies avoid them altogether by carefully designing and constructing the CSR machine components [31]. Some studies try to account for the cable mass and elasticity, but it is also common for the cables to be treated as massless and inextensible [17−18, 21, 30, 32]. There is also extensive research in optimizing the stiffness for CSR configurations with redundant cables [17, 32−34]. A higher stiffness makes the system more robust to external force disturbances.

Various controllers were designed and tested using laboratory-scale experiments [25−30]. One of the simplest methods of control that does not explicitly account for cable tensions is to independently control each cable length to move the end effector to the desired position. This method depends on accurate kinematics, since the required cable lengths are calculated using the position-to-cable-length equations, and a good independent control loop for each cable length. Any system imperfection, such as cable-length errors or cable anchor-point errors, can result in unwanted cable sagging or positioning error. If the CSR is accurately constructed and there is minimal cable sagging this method is simple and effective [35]. One could envision an augmented control system where position errors are measured and used to adjust the cable lengths in real time, as in [30−31, 36−38]. However, the majority of studies in the literature primarily depended on accurate kinematics and avoided using position feedback.

If the individual cable tensions can be controlled, then another form of control can be used, where the cable tensions are varied to accelerate the end effector along a desired trajectory [39−40]. In a CSR configuration with redundant cables, there may not be a unique solution of cable tensions for a desired acceleration, so researchers optimized the cable tensions to achieve various objectives, such as increased stiffness, decreasing the likelihood of cable sag, or reducing vibrations [24−25, 27]. Some researchers tested additional layers of control, such as fuzzy logic or input shaping [30, 38, 41−42]. Some techniques, such as sliding mode control, increase robustness to uncertainties like unknown end-effector mass [22, 43−44]. There are also controller designs based on the analysis of the reachable CSR workspace and avoidance of cable collisions [28, 45−46]. Finally, some control techniques focused on path planning and optimization to achieve specific movements [47−48]. Many of these works include demonstrations of stability, optimality, or algorithm efficiency [49−50]. These kinds of assurances are important, since CSRs are highly nonlinear systems, and real-time calculations and data processing for such systems could be too slow to allow for feedback control.

The successful position control of a CSR depends on the quality of the sensors and actuators used. To control the cable lengths, some researchers relied on the open-loop control of stepper motors [34, 38]. Others used rotary encoders on the motor shaft [31, 37, 41, 50]. The cable tensions were measured using force transducers mounted either on the end effector or near the cable anchor point [17, 27, 41]. The cables used were typically made of thin steel wire, which experience negligible sagging and stretching at small scales [17, 21]. To measure the position of the end effector, some researchers used camera tracking or laser systems [34, 36−37]. Another way is to measure the effector's acceleration using an inertial measurement unit (IMU) [17, 38, 50]. In general, higher-quality components allow for a more precise position control. The sampling period of the sensors also plays a major role in any digital-controller implementation. For example, if the position measurements are too slow, then position errors cannot be detected fast enough to quickly correct. A variety of systems, including personal computers (PCs), data acquisition (DAQ) boards, and commercial microcontrollers, are used to implement the real-time control [17, 31, 37−38].

1.2.3. Applications in Additive Manufacturing

To advance in the design of a CSR, it is important to first identify an intended application. The application guides the selection of a configuration and control strategy, as well as the choice of practical sensors and actuators. One can also begin to consider any likely disturbances. In this dissertation, the intended application is large-scale AM.

Using a CSR for AM is not a new idea. There are several proposals for portable, cable-suspended AM systems [34, 51−52]. However, there are a few realizations of this concept. One realization is found in [53], where researchers created a cable-suspended FDM machine that could create a 2.16-m foam statue in 38 hours (Figure 14). In this system, six cables were used to suspend a foam extruder with 6-DOF control. The cable lengths were independently controlled using separate motors that each tracked a reference cable length using proportional-integral-derivative (PID) control. A feedforward term was added to the controller in order to compensate for the static weight of the extruder. Also, a laser was mounted by the extruder to monitor the height of the foam. This data was used to adjust the height between layers. The true, 3-D position of the extruder was not measured to correct for errors in real time, resulting in some printing errors.

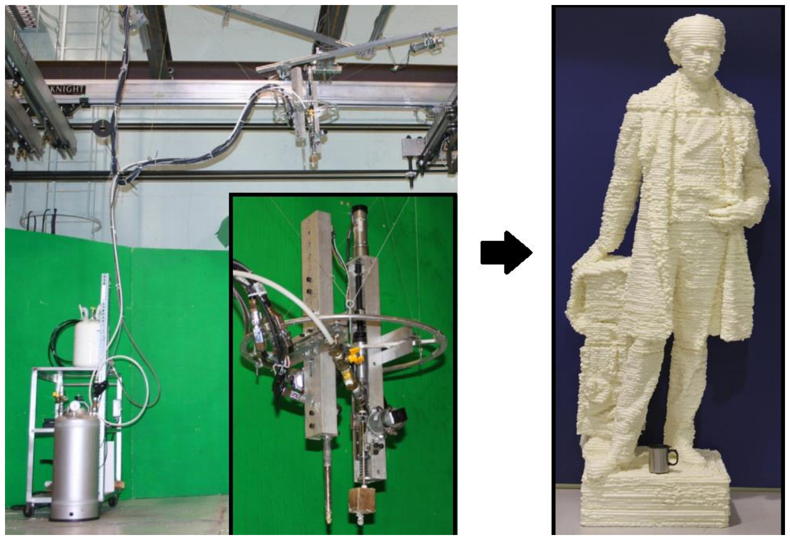


Figure 14. One example of FDM accomplished by using a CSR [53].

Another example of a cable-suspended FDM machine is pictured in Figure 15. Here the extruder platform is suspended by a spring, and three cable pairs at the floor level move the extruder above the build platform [29]. In this case, stepper motors were used to

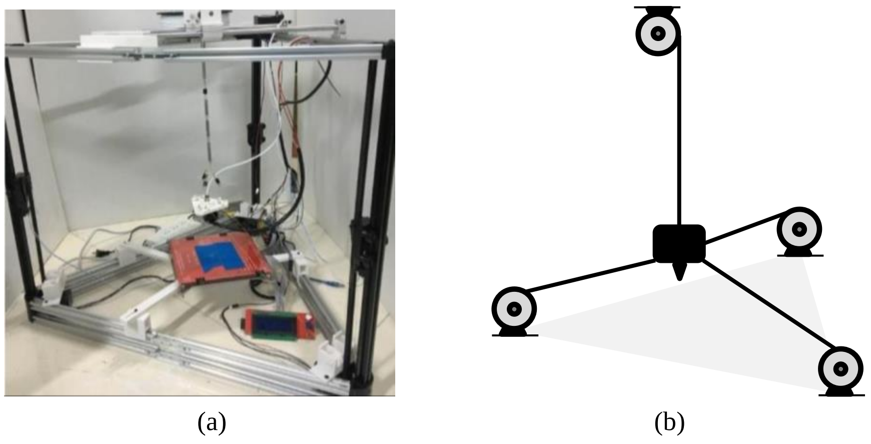


Figure 15. Cable-suspended FDM machine controlled using stepper motors. (a) Actual machine and (b) representative diagram [29].

precisely control the length of each cable, and there was no other position feedback for the nozzle. This machine is much smaller than the one in Figure 14, and is similar in size to the desktop FDM machine in Figure 1. This machine also illustrates the importance of selecting a CSR configuration to match the application. In this case, the single spring pulling up on the extruder is solely responsible for counteracting gravity and also maintaining the tension in the other cables. The floor-level cables, although important for positioning the extruder along the *X*-*Y* plane, could interfere with the workpiece after some number of layers. Thus, the allowable build volume is much smaller than the machine. Nevertheless, the system’s stiffness is enhanced by this configuration, compared to something like Figure 12, since any movement direction of the nozzle is counteracted by at least one cable tension or by the spring tension.

The machine configurations pictured in Figures 14−15 can be scaled and used to produce larger pieces. As discussed in Section 1.1.2, the ability to extrude materials for large-scale AM is an active area of study with many proven examples. However, it has yet to be determined how well a CSR might perform at a large scale since such a system can be vulnerable to environmental disturbances like wind, cable vibrations, or imprecise cable placement. To qualify a CSR for large-scale AM, one might be interested in measuring the system’s positioning resolution, tracking error for straight or circular paths, and the ability to maintain the proper extruder orientation even if there are some disturbances. Position feedback could help enable this functionality. If so, it is useful to investigate exactly how the position feedback might be used in control.

One remarkable example of the intended application is shown in Figure 16, where the CSR concept was scaled to a 13.6×9.4×3.3-m build volume to deposit clay material [54]. According to the creators of this system, who specialize in CSRs for construction, the pictured machine uses thermal sensors to monitor the clay structure. However, there is no real-time feedback of the nozzle position. The eight cables of the CSR are independently controlled using servo motors. This configuration allows for a 6-DOF control of the extrusion nozzle. The clay material, which is stored in a cylinder onboard the extruder platform, can be deposited in widths of 6−30 mm, and the movement speed of the nozzle is up to 1 m/s. Despite being implemented inside a controlled environment with minimal disturbances, the nozzle movement is affected by vibrations. Nevertheless, the creators of this machine produced pieces up to 3.5 m in length and 0.86 m in height. Concrete extrusion using the same machine is expected in future work.

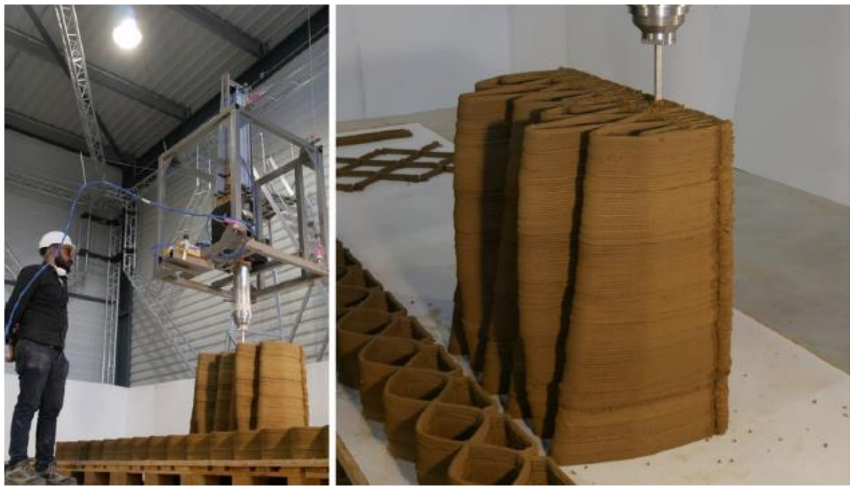


Figure 16. Example of a large CSR that deposits clay material [54].

Another notable work was recently published in [55]. In this work, an eight-cable CSR was designed specifically for concrete FDM (Figure 17). Significant contributions in this research were the full design of a winch, extrusion system, and cable tension sensors. The authors also considered cable sagging and incorporated the relevant calculations in the controller design. The controller consists of three layers. The first layer calculates reference cable tensions based on the desired extruder position and the system kinematics, including the calculation of cable sagging. This layer also minimizes the sum of cable tensions using a common optimization method. The second layer uses feedback from the cable tension sensors to calculate new reference cable lengths. Finally, the third layer of the controller manages the eight cable lengths via rotation of the winches. There is no measurement or feedback of the true nozzle position.

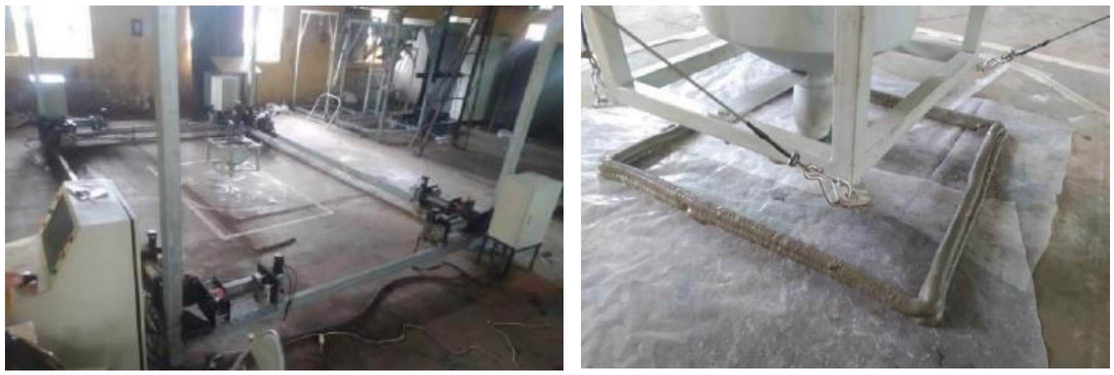


Figure 17. A recent example of a large-scale CSR for concrete FDM. The test path is a square with the side length of 0.8 m [55].

2. CONTRIBUTIONS OF THIS DISSERTATION

The contributions of this dissertation are placed in three groups of work. The first group is the design and analysis of a CSR for large-scale AM. This includes the discussion of a configuration, its structural properties, and the consideration for various sensors and actuators. The second group contains several control strategies for the CSR that rely on the definition of system models. The third group of work is the construction and use of a prototype experiment to test some of the control strategies. This chapter summarizes the main contributions in each of these groups while the details are presented in Chapters 3−5. As a whole, the most novel contribution of this dissertation is the new consideration given to some types of system imperfections and disturbances.

2.1. Cable-Suspended Robot Design

The first contribution of this dissertation is the recommendation of a general design for a portable CSR that can be used for large-scale AM. This is not a detailed design, but an overview of the machine configuration and an analysis of the structural implications. The analysis includes tension and stiffness properties, as well as some algorithms to study the useful operation space of the robot. Some sensor and actuator options are also discussed.

2.1.1. Configuration Analysis

A basic configuration for the proposed CSR is selected. As discussed in Section 1.2.1, the configuration includes the number and location of cables in the system. These details determine the robot operation space and affect the system's portability. The forward and inverse kinematics are studied, and the effects of system imperfections are detailed in mathematical equations. The control DOFs and the allowable movement region are also described. Finally, the configuration is supported by simulations of the system tension and stiffness at different positions throughout the movement region. These analyses include a brief introduction to some cable stiffness models.

2.1.2. Sensor and Actuator Selection

Upon the selection of a robot configuration, the number and types of required sensors and actuators are considered. These elements of the design are discussed in general terms as the proper component selection in a large-scale CSR is more appropriate in future work. Nevertheless, the operating principle of the sensors and actuators is presented.

The main function of the actuators is to control the cable lengths. Thus, the resolution or frequency limitations of the actuators directly affects the positioning precision of the system. These limitations, as well as the power and control systems required to operate the actuators, are discussed.

Some sensors are included with the actuator design, but others can be used to directly measure the real-time position of the CSR to provide feedback. Various types of sensors for this purpose are presented, and their limitations are anticipated.

2.2. Position Control

The second contribution of this dissertation is the synthesis of position control strategies for the CSR while considering system imperfections and disturbances. The control effort can be divided into two tasks, system modeling and controller design. A third, novel task involves estimating cable-placement errors when they are unknown.

2.2.1. System Modeling

The first task in providing position control is creating a mathematical model of the CSR. Several cable-stiffness models are included, and two CSR-system models are detailed. One system model is based on controlling the cable lengths, and the other model assumes direct control of the cable tensions. Both models allow for including system imperfections and external disturbances. The imperfections include cable placement errors, cable-lengths errors, and sensor-measurement errors. Dynamic disturbances include external forces, measurement noises, and sudden cable-length changes. All of the models are simulated in MATLAB to produce time-series plots.

2.2.2. Controller Design

The second task in providing position control is to create effective control laws for the system. Several control laws, using both system models, are presented. The first system model, which is based on cable-length control, assumes that the cable lengths and position of the CSR are measured in real time. The second system model, which is based on cable-tension control, assumes that the cable tensions are measured. The control laws, their stability, and robustness are analyzed in MATLAB by simulating the closed-loop control of the CSR in various conditions.

2.2.3. Cable Placement Estimation

One novel task in this dissertation is estimating-cable placement errors using indirect measurements. This result can be useful in practice since it may be impossible to avoid some cable-placement errors for large-scale CSRs. A novel algorithm for estimating the cable placement is simulated and tested in the forthcoming experiment.

2.3. Prototype Experiment

The third contribution of this dissertation is the validation the CSR configuration and controller design via experimentation. Though it is desirable to test a large-scale machine, the experiment in this dissertation was performed using a representative model in a laboratory setting. The experiment represents the robot configuration, sensors, and actuators selected during the design phase of this work. The prototype is not equipped to perform FDM, or any other AM, since the focus is on the positioning function of the CSR.

2.3.1. Closed-Loop Control

The experiment was an opportunity to test the real-time control of a CSR with position feedback. As mentioned in Sections 1.2.2−1.2.3, there are a few examples of closed-loop position control in the literature. In most cases, the control system relies on tension feedback only. In a few cases, a camera or laser was used to record the CSR movement. This dissertation provides experimental results with and without the use of position feedback to show that close-loop control can be used to improve system performance.

2.3.2. Disturbance Testing

The prototype experiment also demonstrates the effectiveness of closed-loop control despite various disturbances and system imperfections, such as cable-length and camera-tracking errors. To the author’s knowledge, no experiment has been conducted previously to test a CSR under thse conditions, and the results can be useful for successful implementations of a large-scale CSRs.

3. ANALYSIS AND SIMULATION

This chapter contains analysis and simulation to support the design and control of a CSR for AM. First, the CSR configuration is selected. The configuration properties, such as the kinematics and stiffness, are derived. The system's sensors and actuators are also described. Second, this chapter presents the controller design, offering various system models and control methods that address system imperfections and disturbances. This chapter also includes a novel method to estimate cable-placement errors.

3.1. Cable-Suspended Robot Design

The intended application is an important factor in selecting a CSR design. In this dissertation, it is envisioned that a CSR be used for some types of AM, like FDM or DED, where the main function of the AM machine is to position the extrusion or deposition nozzle. Therefore, the CSR should provide for 3-D positioning of the nozzle with some accuracy and without interfering with the workpiece. The AM machine should also exhibit some minimum stiffness so that external forces do not significantly disturb the build process. In this section, a CSR configuration is selected and analyzed for AM use.

3.1.1. Base Configuration

To provide for 3-D positioning of an AM nozzle, a CSR must have at least three cables. As shown in Figure 12, three cables originating from the same height can position a rigid body within a triangular-prism region. The triangle vertices are the three cable anchor points. Ideally, a CSR with this configuration could deposit material in 2-D slices that fit within the trianglular footprint and up to the height of the cable anchor points.

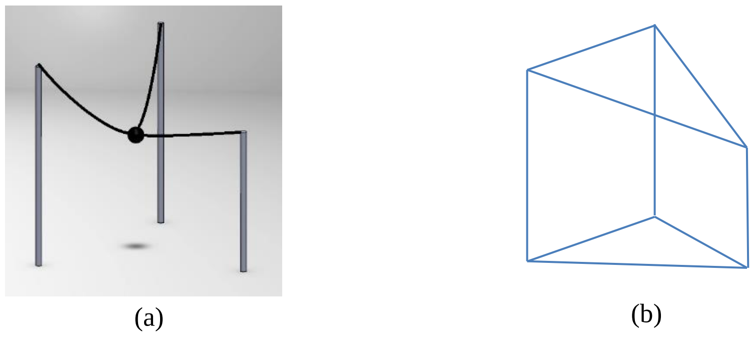


Figure 18. (a) A CSR configuration with three cables, where one cable anchor point is lower than the other two and (b) the movement space.

Figure 18 shows another three-cable configuration where one of the cable anchor points is lower than the other two. Compared to Figure 12, this configuration has the same triangular footprint. However, the allowable 3-D movement region is reduced. This simple example demonstrates that, in order to maximize the use of vertical space, all of the cable anchor points should be at the same height.

A three-cable configuration has the advantage of no redundant cables since any resting position of the suspended body would require a positive tension in all three cables. Furthermore, the three required cable tensions are unique for each equilibrium position. This fact is explored in Section 3.1.4. One major disadvantage of a three-cable system is that the triangular footprint significantly constrains the possible size and shape of the material layers. For example, the maximum diameter of a circular layer would be much smaller than any of the triangle sides. Any non-triangular shape would meet geometric restrictions if it is to fit within the trianglular footprint.

A workspace with a square or rectangular footprint can be used instead of a triangular one. Then, four or more cables are needed. Figure 19 depicts a simple CSR with

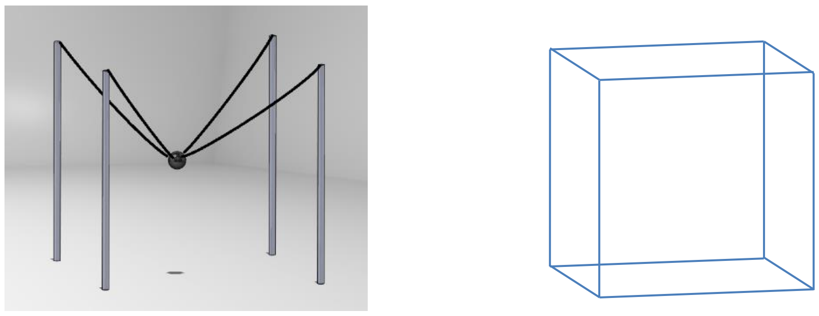

(a) (b)

Figure 19. (a) A CSR configuration with four cables and (b) the movement space.

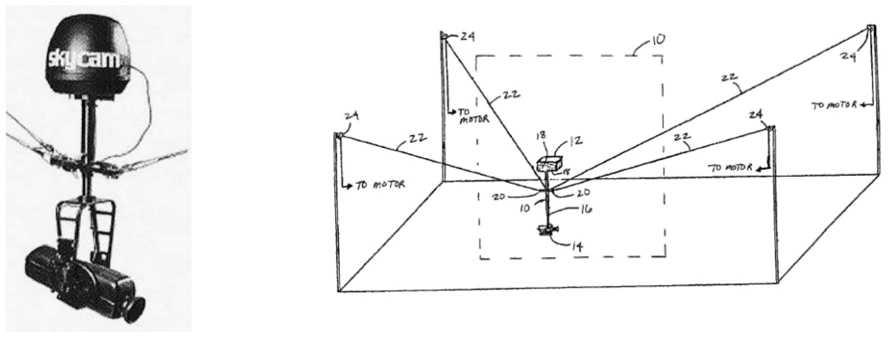
four cables arranged in a square at equal height. With this setup, the suspended body can be positioned within a rectangular-prism region. Compared to the three-cable configuration, the four-cable system provides a more efficient use of the space.

The four-cable system is over-constrained, however, since only three cables are needed to hold the body at rest in any position within the work region. The fourth cable is redundant. Interestingly, the redundant cable can change depending on, among other things, the instantaneous position of the body. For example, if the suspended body in Figure 19 is exactly in the center of the machine, any one of the four cables could be considered redundant, since any of them could be removed without affecting the position of the body. Whenever there are more than three cables there can be ambiguities about the redundant cables.

More cables can be added to the CSR. If done so with equal spacing, the footprint of the resulting CSR is always an equilateral polygon. As the number of cables increases, the workspace footprint approaches the shape of a circle. Since additional cables may over-constrain the system without significantly increasing the workspace region, a four-cable configuration appears to be a favorable selection.

Many of the CSR configurations in the literature use more than four cables, and for good reason. If the orientation of the rigid body is to be controlled, in addition to its position, then six or more cables are required. This is because six independent forces are needed to fully control the 6 DOFs of a rigid body. There are numerous examples in the literature of clever cable arrangements that provide positional and angular control of the CSR end effector. Some of these examples are mentioned in Section 1.2.1 [19−24]. The number of cables alone does not determine the control DOFs. The placement of the cables must also be considered. Some cable arrangements are subject to cable collisions or a limited range of movement even if the arrangement offers 6-DOF control. For this reason, it is beneficial to minimize the number of cables used.

The present focus is on the application of CSRs for large-scale AM. In a large-scale AM machine, the dimensions of the material nozzle are much smaller than the lengths of the cables. Then, for the purpose of controlling the cable lengths, the nozzle assembly can be modeled as a point mass. In other words, the cables can be considered to meet at a single point, as depicted in Figures 18−19. In this case, the CSR's primary function is to provide 3-DOF position control. Four cables are sufficient for this task. To support this claim, one may look to one existing example of a large-scale CSR, pictured in Figure 20. This system, called the SkyCam, is commonly used to move a camera within large sports venues that can be more than 100 m in length. Four cables provide the position



(a) (b)

Figure 20. (a) One embodiment of the SkyCam system, a sport camera that is suspended by four cables [56]. (b) The system drawing as presented in a 2005 U.S. patent [57].

control for the camera platform. Since the camera platform is much smaller than the cable lengths, the cables can be approximated as meeting at a single point. The platform itself contains mechanisms that regulate the camera orientation.

A CSR for large-scale AM could be constructed similar to Figure 20 by replacing the camera platform with a nozzle mechanism. This idea was demonstrated in [52−55], but with the use of more than four cables to control the extruder orientation. As an example, in [52] a four-cable configuration was proposed to control the extruder movement, plus eight additional horizontal cables to maintain the nozzle orientation (Figure 21). In the pictured machine, the pulleys of the horizontal cables are raised as material layers are added. This way, the cables do not interfere with the workpiece. The material itself would be stored in a container besides the machine and pumped to the extruder via a hose. The stated motivation for this concept was that this machine would be more cost effective than a traditional concrete manufacturing process. Furthermore, the

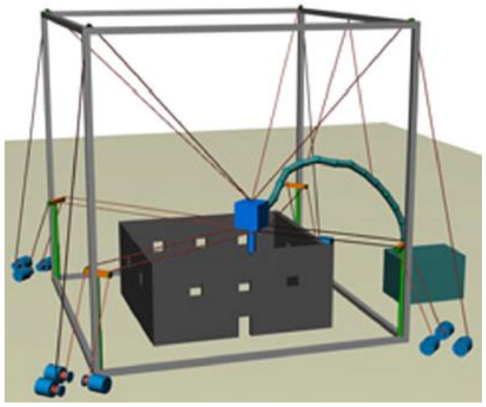
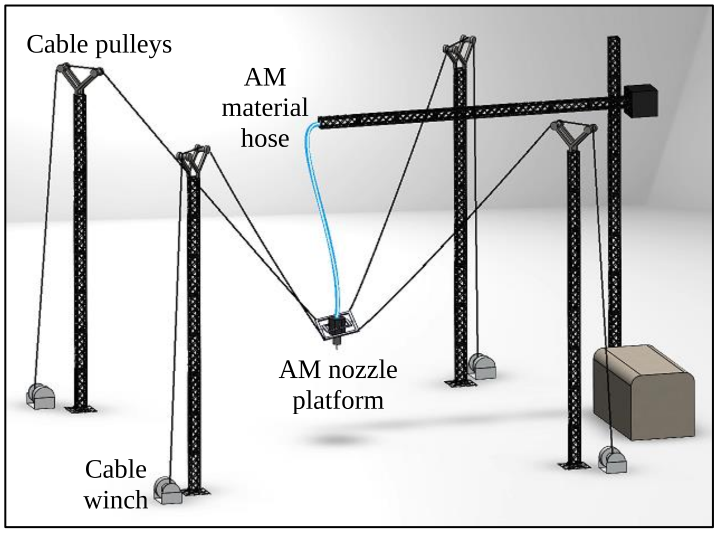


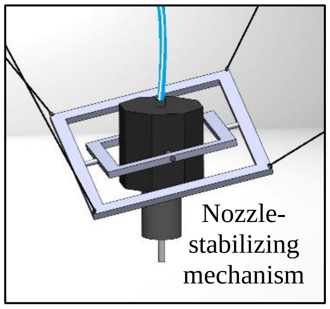
Figure 21. A CSR configuration for large-scale AM where four cables from the top corners position the extruder and additional horizontal cables stabilize the extruder [52].

cable-driven machine would be lighter and easier to transport than a rigid mechanism like the ones pictured in Figures 6−8.

In this dissertation, a four-cable configuration is selected for study. The configuration is depicted in Figure 22. The components in this image are not to drawn to scale and are exaggerated for clarity. Four cables, alone, position the extruder platform within the large rectangular-prism work region. Since the extruder is much smaller than the cable lengths, the extruder is treated as a point mass. The orientation of the extruder is locally, and independently, controlled using a gimbal mechanism. The details of this mechanism are not prescribed in this dissertation, but the mechanism could provide for orientation and fine-position control. The coarse positioning is provided by the CSR. The AM material is delivered from a reservoir to the extruder via a thin hose from above. Some mechanism may be necessary to guide the hose and ensure that it does not weigh down or push excessively on the extruder. The other structures required in this configuration are



(a)



(b)

Figure 22. (a) A large-scale AM CSR with 4 cables and (b) a detail view of the AM nozzle platform. This configuration concept is the object of study in this dissertation.

four towers with pulleys that serve as stationary cable anchor points. Each cable length is controlled by its own winch mechanism at the ground level. Since all of the cables pull at the extruder from above, the cables will not interfere with the material layers that are being deposited. The extruder can also be positioned anywhere in the work region without any cables crossing each other.

The four-cable concept in Figure 22 guides the remaining analysis work in several ways. First, the concept must be supported by answering questions of stability and stiffness. Without any cables at ground level, upward forces cannot be completely counteracted. Furthermore, the effective mass of the extruder may vary during operation due to the attached material hose. The issue of redundant cables must also be studied to ensure that operation of the machine is smooth enough for its intended use. Finally, the towers that position the cables must be precisely located for proper operation, and any placement errors could significantly affect the positioning accuracy. The analysis in the forthcoming sections answers these questions. An important assumption in the analysis methods is that the extruder can be treated as a point mass.

3.1.2. Kinematics

The kinematics of a robot describes how the end-effector position and the robot joint variables are related to each other. In the case of a CSR the joint variables are the cable lengths. The forward kinematics describes how the cable lengths are used to calculate the end-effector position, and the inverse kinematics describes how the end-effector position is used to calculate the cable lengths. These calculations are complicated by the fact that cable slack can occur and may be difficult to accurately model especially if there are cable vibrations. The calculations could also be complicated if there is more than one solution for a certain robot state. This is not a problem for the configuration in Figure 22 since every 3-D position of the end effector corresponds to a unique set of cable lengths, assuming the cables have no slack.

The inverse kinematics can be derived using a vector approach, as is often seen in the literature. Let the position of the end effector be stored in the vector

$$\boldsymbol{p} = [x \quad y \quad z]^T. \tag{1}$$

As previously noted, this is the point-mass approximation of the end effector, and all cables are assumed to meet at this coordinate. The position of the *i*’th cable anchor point is stored in the vector

$$\boldsymbol{c}_i = [x_i \quad y_i \quad z_i]^T. \tag{2}$$

If cable *i* has no slack, the required cable length is the Euclidian distance from the anchor point to the end effector. That is, the cable length is

$$l_i = \|\boldsymbol{c}_i - \boldsymbol{p}\|_2 = \sqrt{(x_i - x)^2 + \left(y_i - y\right)^2 + (z_i - z)^2}. \tag{3}$$

The calculation is the same for every cable in the system.

The actual length of a cable may differ from that predicted in (3). If there is slack in a cable, its length must be slightly longer than the straight-line distance from the anchor point to the effector. The amount of slack depends on factors like the density and length of the cable. A heavier and longer cable will have more slack. The elasticity of the cable will also change its effective length. With stretch, the cable length will be slightly larger than the un-loaded length. Together, cable elasticity and slack make it difficult to predict the true cable lengths. One solution is to experimentally determine the cable lengths at many points in the CSR workspace and use this data to add a correction factor in (3).

If a cable is redundant at a certain time instant, it can be removed without affecting the end-effector position. Therefore, the length of a redundant cable can be longer than that calculated in (3), as demonstrated in Figure 23. However, its length should not be shorter, since a shorter cable would pull the end effector away from the desired position. Neither should the redundant-cable length be much longer than that prescribed in (3) because the cable should be ready to provide tension in subsequent time steps. The best policy for managing cables is to keep them close to the lengths required by the kinematics.

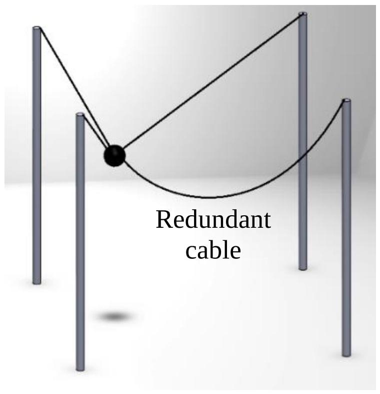


Figure 23. The length of a redundant cable can be bigger than that predicted by (3), but it should not be shorter.

One should also consider whether the desired end-effector position is stable, in the sense that the end effector could remain at rest in that position. Figure 24 shows an example of a position that cannot be supported by a four-cable CSR because it would require at least one of the cables to be in compression. Nevertheless, one could naively

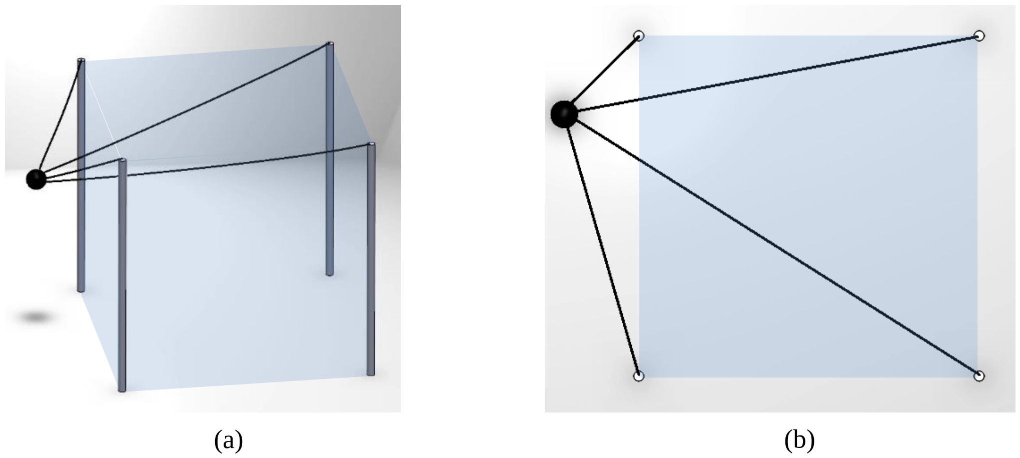


Figure 24. (a) An end-effector position that cannot be supported by the four-cable configuration. The shaded region is the supported workspace. (b) Top view of the unsupported position.

calculate the cable lengths for such a position. This example demonstrates that the kinematic calculations can be made more realistic by imposing some restrictions on the end-effector position.

The supported workspace of a CSR is the 3-D convex hull of the cable anchor points and their footprints on the ground. To demonstrate this principle, consider the six-cable example pictured in Figure 25. This configuration is identical to that in Figure 24 but with two additional cable anchor points. One of the new points is higher than the original four and is to the right of the square area. The second new anchor point is at a lower height and in the interior of the square area. The new point to the right expands the supported workspace, but the new point in the interior does not. In fact, the cable anchor point in the interior could be removed without changing the supported region. Also, there are some positions in the workspace, such as the one pictured, where the end effector is higher than the short anchor point. If the CSR were to be used for AM, all of the cables

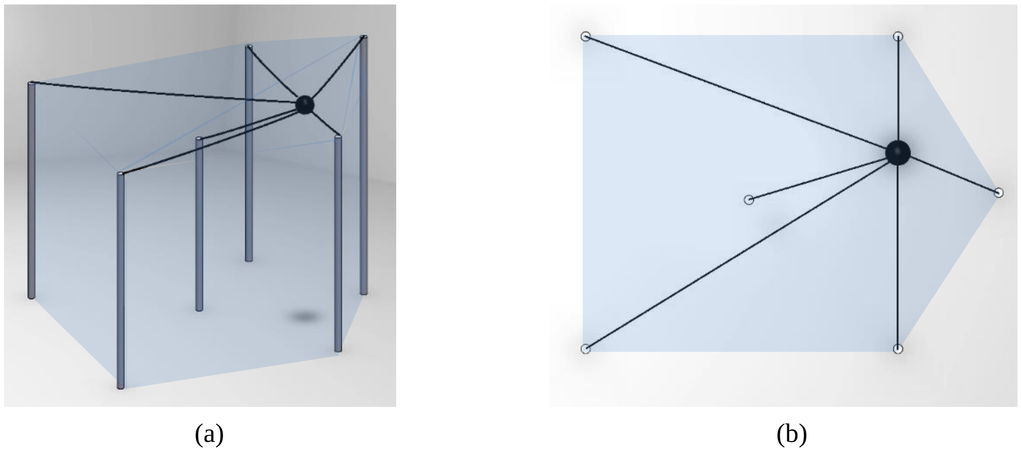


Figure 25. (a) An example six-cable configuration and (b) a top view. The shaded region is the supported workspace, which is the 3-D convex hull of the cable anchor points and their footprints.

should remain higher than the end effector to avoid disturbing the work piece layers.

Based on the above considerations, a three-step process can be used to constrain the kinematic workspace for a general AM CSR. First, the 2-D convex hull for all of the cable anchor points, as seen from a top view, is drawn. This is demonstrated in Figure 25(b). Second, any cable anchor point in the interior of the convex hull is removed. This step is optional if all of the anchor points must be used for another reason. Finally, the 2-D convex-hull shape is placed at the height of the lowest anchor point. Then, the entire 3-D region below the convex hull will be available for AM, and any point within this space can be used to calculate the valid cable lengths using (3).

The forward kinematics describes how the end-effector position can be calculated from the cable lengths. Redundant cables and cable slack also complicate this calculation. If (3) is squared on both sides, the result is

$$l_i^2 = (x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2, \tag{4}$$

which is the equation of a sphere. The radius of the sphere is the cable length and the center is the cable anchor point. A sphere can similarly be drawn for each cable, and the intersection of the spheres is the location of the end effector. However, any cable slack or stretch makes (4) invalid. Furthermore, only three valid spheres are required to find the end-effector position. Knowing this, the forward kinematics calculation can be based on a procedure that first identifies three valid spheres and, then, finds their intersection. In other words, only three cables without slack are needed to define the position of the end effector, and the challenge is correctly finding three such cables.

The simplest forward-kinematics case is when the CSR only has three cables. Then, any valid position of the end effector creates a tension in the three cables and (4) gives three valid equations. These equations can then be solved, numerically or analytically, for the position of the end effector. If there are two solutions to the equations, the solution with the smaller *z* coordinate is selected, since gravity will pull the end effector to the lowest possible height. Figure 26 gives a visual example of a three-cable system with the cables replaced by spheres. The intersection point of the spheres indicates the point-mass position, and the equation for each sphere is given by (4).

Suppose that a fourth cable is added to Figure 26. If the new cable is taut, the imaginary sphere corresponding to the cable will intersect the other spheres at the location of the point mass. If the new cable has slack, the sphere prescribed by (4) will contain the point-mass. Then, that particular sphere is not useful for finding the position of the point

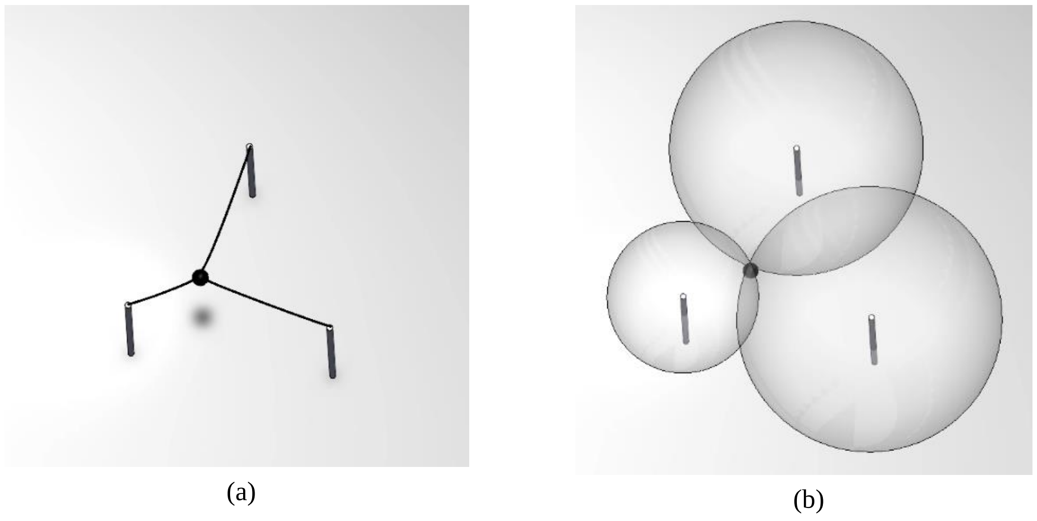


Figure 26. (a) Top view of a three-cable CSR. (b) The cables are replaced by spheres centered at the cable anchor points and with the radius equal to the cable lengths. The intersection of the three spheres is the position of the end effector.

mass. If the slacked cable is gradually shortened, it will eventually start pulling on the end effector. At that point, one of the original three cables would get slack. It is also possible for the fourth cable to be too short to connect to the other cables. A cable that is too short can be visualized as a sphere that does not intersect the other spheres.

The forward kinematics gets more complicated as cables are added. For example, it is possible to have six cables such that two groups of three cables can each support a different position for the suspended mass. It is also possible that various solutions exist depending on which cables are included in the calculation.

One procedure for the forward kinematics calculation of a general CSR system is presented here. First, some conditions for a valid end-effector position are stipulated:

1. For all of the cables to come together at the end-effector position, any set of two cables must be able to reach each other.
2. When the end effector is at a resting position, three cables will be in tension and the remaining cables will either be in tension or have slack.
3. The end-effector position must be within the 3D convex hull of the cable anchor points and their footprint points on the ground.

These conditions can be visualized using imaginary spheres, like those presented in Figure 26. Condition 1 says that if all spheres simultaneously overlap at some point, then any two spheres must overlap with each other. Condition 2 says that the end-effector point is at the intersection of three sphere boundaries, and the rest of the spheres either touch or contain that point. The third condition excludes sphere-intersection points that are not in the supported work region of the CSR. Based on these conditions, the following steps are used to determine the position of the end effector given a set of cable lengths and anchor points:

1. Remove any cables that cannot reach all other cable ends. Alternatively, keep the largest group of cables such that any two cables in the group can connect.
2. Enumerate all possible combinations of three cables. For each combination, find the intersection points of the cable spherical boundaries, using (4). This will yield a set of candidate points for the end-effector position.
3. Remove all candidate intersection points that are not within the 3D convex hull of the anchor points and their footprint points.
4. Select the candidate intersection point that is contained within all of the spherical boundaries. This will be the end-effector position.

For a small number of cables, this procedure can easily be programmed in a computer algorithm. As the number of cables grows, the same algorithm will take exponentially more time to complete. Step 2 includes enumerating all possible combinations of three cables, which is an operation that takes much more time as the number of cables increases. Step 1 can also take time, since it is not trivial how to select the largest group of cables that can connect with each other. One way to accomplish this task is to form an adjacency matrix for the cables and, then, use a maximal clique algorithm [58].

Some examples of the procedure are now presented. The steps outlined above were programmed in an algorithm using MATLAB, and the code for this algorithm is presented in Appendix B. Figure 27 shows an example where four cable anchor points are arranged in a square, at equal height. The cable lengths were purposefully selected so that one cable

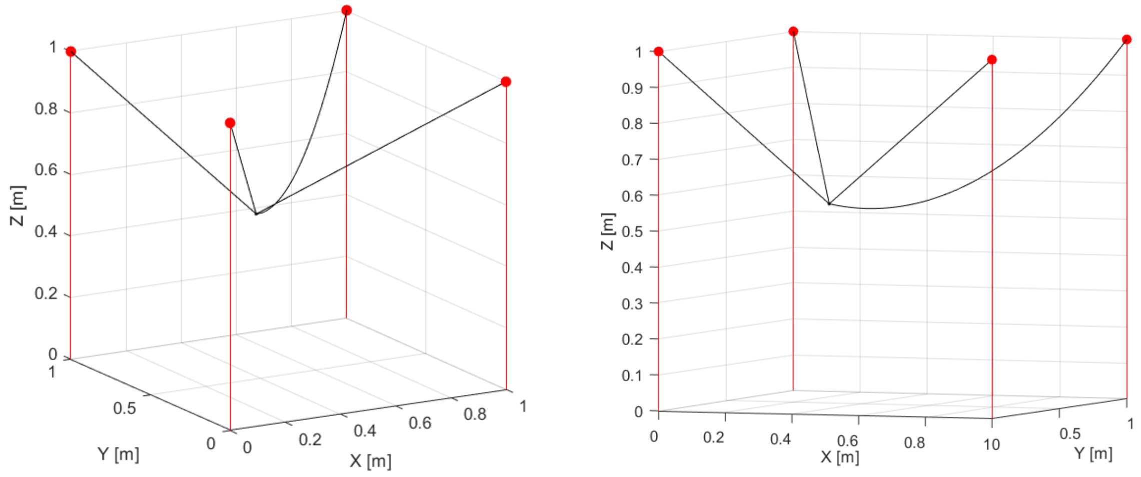


Figure 27. Example of a four-cable system where one cable has slack (two views).

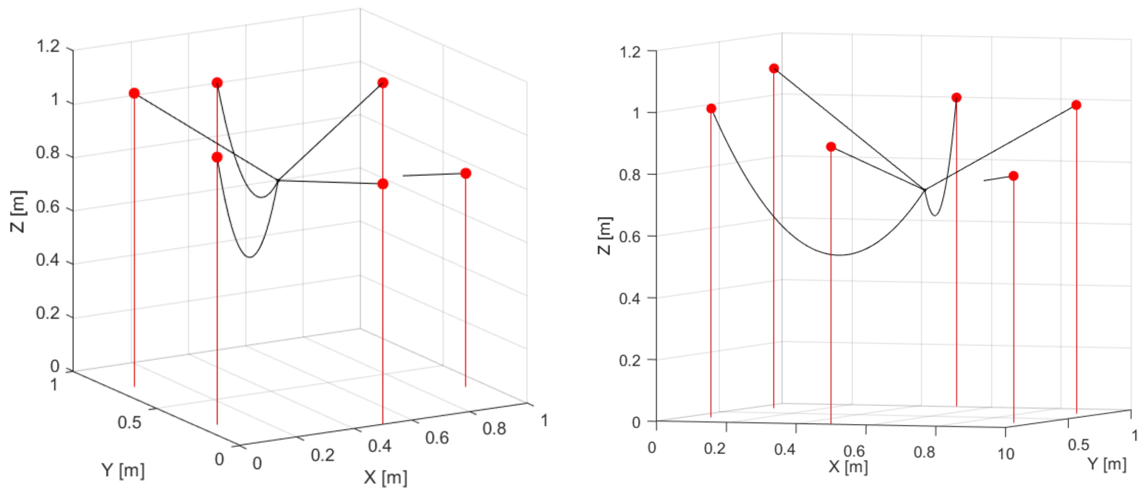


Figure 28. Example of a six-cable system where two cables have slack and one cable is too short (two views).

is excessively long. The procedure outlined above determines that the pictured position is the result of this arrangement. As expected, three of the cables are in tension, and the fourth cable has slack. A more complicated example is pictured in Figure 28. Here, six cables are arranged in a hexagon, but with the anchor points at different heights. Some of the cables are too long, and one cable is too short. The algorithm predicts that the pictured position is the result. Again, three cables are in tension. Two cables reach the end effector but have slack. The cable that is too short to connect to the end effector is shown floating in space. Disregarding the short cable, one can visually confirm the validity of the solution.

The procedure presented here is one forward-kinematics solution that addresses cable slack and short cables. The procedure is not perfect, however. In particular, the first step, which checks that the cables can all connect to each other in pairs, is a necessary-but-not-sufficient condition for a valid solution to exist. It is possible for the cables to be long enough to reach each other in pairs and, yet, not be able to reach a common point. Furthermore, the procedure is intended to be used with CSRs where the anchor points are all above end effector. The algorithm was not attempted with configurations that have additional cables at ground level.

3.1.3. Kinetics

The kinetics of a dynamical system deals with the forces that cause the system's motion. In a CSR the body of interest is the end effector. Presently, the focus is on an end effector that is much smaller than the CSR configuration, so a point-mass approximation is used. For a point mass there are no moments to consider, and the motion is entirely due to the sum of linear forces acting on the mass.

The nominal forces acting on the end effector are the cable tensions and gravity (Figure 29). Other forces can come from air drag or contact with foreign objects. Cables that have slack contribute a small force compared to the cables that are in tension, and no cables can produce a compressive force.

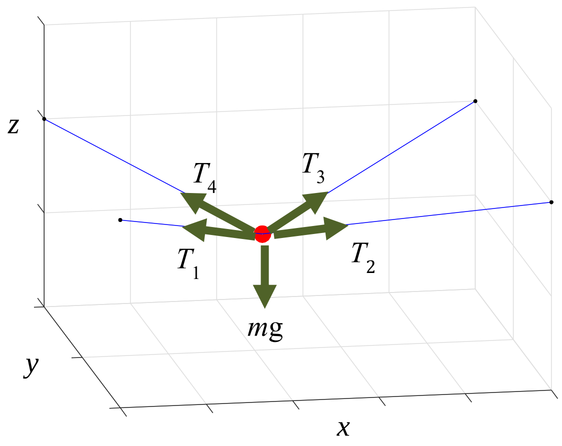


Figure 29. Nominal forces acting on a point mass suspended by four cables.

The cable tensions act along the cables. If there is negligible sagging, the cable tensions approximately point from the end effector to the cable anchor points. If there is sagging, the tension vectors tilt down, towards the horizon. If the sagging is significant, such that the cable hangs in a “U” shape, the weight of the cable produces a small force on the end effector towards the ground. For simplicity of analysis, it is assumed that any cable with slack provides a negligible tension and that any tension points exactly from the end effector to the cable anchor point. For any static equilibrium point (SEP), the sum of the cable tensions must be zero in the *x* and *y* directions and equal to the end-effector weight in the positive-*z* direction. To accelerate the end effector, the tensions must provide a net force in the direction of acceleration. Since none of the cables can push the end effector, the maximum downward acceleration is that due to gravity. Only three cables need to be in tension to support an end-effector, but it is possible for all of the cables to be in tension. This idea is explored in Section 3.1.4.

The gravity force always acts in the negative *z* direction and is equal to the weight of the end effector. The mass of the end effector could change if it carries a payload that varies. For example, the mass of an AM nozzle can vary due to the flow of material. Any hose attached to the nozzle would also add weight. The total weight of the end effector is important because it ultimately determines the tensions that the cables must withstand. On the other hand, a light end effector could more easily be displaced by disturbance forces.

Other external forces can act on the end effector (Figure 30). Drag, either from the ambient air or from energy dissipation in the cables, can dampen the end-effector movement. Wind can push the body or cables. The movements of the hose attached to the AM nozzle can produce lateral and downward forces. When material is being deposited, there will be a reaction force. Ideally, the external forces should be much smaller than the weight of the end effector and the cable tensions. Otherwise, the forces would significantly disturb the movement of the CSR.

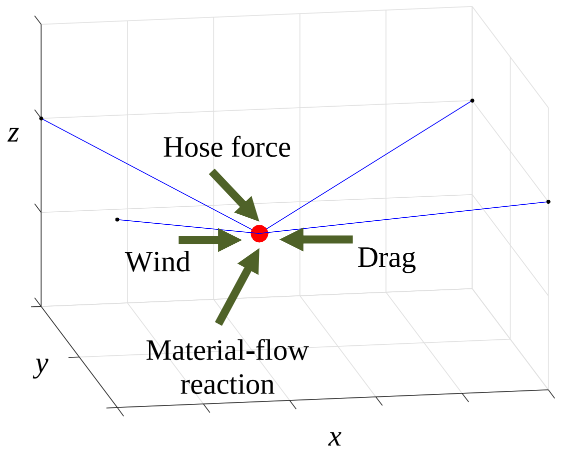
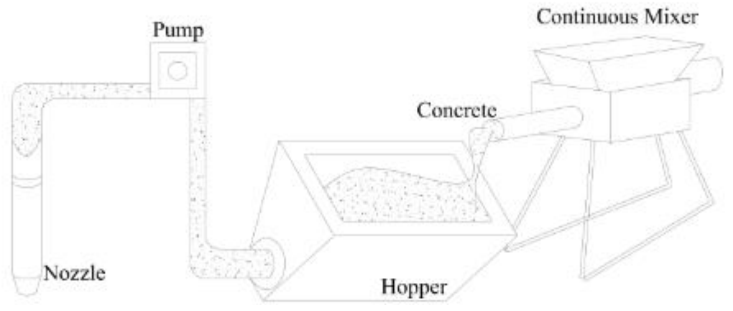
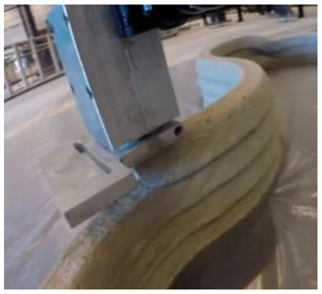


Figure 30. External forces that may act on a CSR end effector in an AM application.

One plausible application for a large-scale CSR is concrete FDM. Figure 31 depicts an FDM nozzle for concrete [59−60]. The concrete composition and material flowrate are carefully selected to ensure ease of extrusion and good layer adhesion. Many cement mixtures and nozzle geometries have been tested [61]. For example, the cement mixtures were varied in percent water content, and circular nozzles were up to several inches in diameter. The exact method of material deposition has also been modified in hopes of improving the layer adhesion. In some cases, the nozzle hovers clear above the previous layer, relying solely on gravity to deposit the new concrete. In other cases, the nozzle is purposefully lowered to provide some downward pressure on the material, helping eliminate gaps in the layers. Many studies experimented with the rheological properties of concrete for AM [62−63]. For example, a material slump experiment helps evaluate whether a fresh concrete volume will maintain its shape over time. Some studies focus specifically on the flow of the concrete to evaluate extrudability [64−65]. Despite all the previous work, it remains unclear how much reaction force an FDM extrusion



(a)



(b)

Figure 31. (a) Concrete extrusion diagram depicting the flow of material through an FDM process [59] and (b) an actual concrete nozzle depositing material layers [60].

nozzle will experience while depositing material. In previous large-scale AM studies, the nozzle reaction forces were not a major issue since the machines were rigid enough to withstand reactions. In a flexible CSR, the reaction forces can be more significant for successful operation because these forces could more easily displace the extruder. Further studies measuring nozzle reaction forces are recommended, especially since the nozzle geometry and material properties remains an active area of study. In this dissertation, the exact nozzle and material properties are not specified.

3.1.4. Tension Analysis

With the CSR configurations, kinematics, and system forces desribed, a more detailed discussion of the cable tensions can begin. In a general robotic system, the joint variables are controlled via the joint forces. In a CSR the joint variables are the cable lengths and the joint forces are the cable tensions. Aside from gravity the cable tensions are the primary forces used to control the precise movement of the end effector.

Cables have two specific properties that affect their use in control. First, they can only provide a tensile force. Second, the direction of the force depends on instantaneous position of the respective cable. Since all of the cables lead to the point mass of the end effector, the coordinates of the end effector can be used to determine the position of the cables. As in Section 3.1.2, let the coordinates of the end effector be given by (1), the coordinates of the *i*th cable-anchor point by (2), and the cable lengths by (3). Equation (3) assumes that the cables are taut. A slack cable provides no tension. Let the tension vector for the *i*th cable be denoted as

$$\boldsymbol{T}_i = [T_{ix} \quad T_{iy} \quad T_{iz}]^T. \tag{5}$$

The net force on the end effector due to the cable tensions is the vector sum

$$\boldsymbol{F}_{cables} = \sum_i \boldsymbol{T}_i. \tag{6}$$

Now, what remains is defining the Cartesian components of $\boldsymbol{T}_i$ in (5). Suppose the magnitude of $\boldsymbol{T}_i$ is known. After all, the CSR is controlled by varying this quantity for each cable. Since cable i is in tension, it runs in, approximately, a straight line from the end effector to the cable anchor point. Then, the direction of $\boldsymbol{T}_i$ is given by the unit vector

$$\boldsymbol{v}_i = \frac{\boldsymbol{c}_i - \boldsymbol{p}}{\|\boldsymbol{c}_i - \boldsymbol{p}\|_2} = \frac{[x_i - x \quad y_i - y \quad z_i - z]^T}{\sqrt{(x_i - x)^2 + (y_i - y)^2 + (z_i - z)^2}}. \tag{7}$$

In (7), the numerator is a vector from the end effector to the cable anchor point, and the denominator is the magnitude of that vector. Then, the tension vector can be expressed as

$$\boldsymbol{T}_i = t_i \frac{[x_i - x \quad y_i - y \quad z_i - z]^T}{\sqrt{(x_i - x)^2 + (y_i - y)^2 + (z_i - z)^2}}, \tag{8}$$

where t_i is the tension magnitude for cable i. Note that the denominator is just the cable length from (3).

The sum of cable tensions in the Cartesian coordinates, generally stated in (6), can be expanded, using (8), as the column vector

$$\boldsymbol{F}_{cables} = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = \sum_i \frac{1}{l_i} \begin{bmatrix} x_i - x \\ y_i - y \\ z_i - z \end{bmatrix} t_i, \tag{9}$$

where l_i was substituted for the vector norm in (8). This is the final, but still general form of the CSR cable forces that will be used for further analysis. A similar derivation was used in previous studies (Section 1.2.2). Others used a geometric approach with trigonometric functions instead of vectors.

To produce the desired end effector movement in 3D space, the three Cartesian-direction forces must be controlled during the movement. From (9) it appears that every cable can play a part in providing the three force components, but the cable-specific contributions depend on the anchor placements with respect to the end effector. Suppose that there are three cables in a CSR. Then, the sum of cable forces can be stated as the matrix equation

$$\boldsymbol{F}_{cables} = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = \begin{bmatrix} \frac{x_1 - x}{l_1} & \frac{x_2 - x}{l_2} & \frac{x_3 - x}{l_3} \\ \frac{y_1 - y}{l_1} & \frac{y_2 - y}{l_2} & \frac{y_3 - y}{l_3} \\ \frac{z_1 - z}{l_1} & \frac{z_2 - z}{l_2} & \frac{z_3 - z}{l_3} \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}. \tag{10}$$

The matrix in (10) simply transforms the cable tensions into the Cartesian directions and sums the forces component-wise. There is a unique set of tensions for any desired force vector as long as the matrix is nonsingular. To ensure that this is the case, the three cables should satisfy the following conditions:

1. The cable anchor points are distinct. This means that the cables will each pull the end effector in a different direction. If any two cables share the same anchor point, then they could be replaced by a single cable.
2. All three Cartesian coordinates appear in the vectors from the cable anchor points to the end-effector position. This means that the cable tensions can provide a force to the end effector in the required *x*, *y*, and *z* directions.

3. No combination of two anchor points plus the end effector forms a straight line. This means that no two cables pull the end effector in exactly the opposite direction. If this were the case, then the two cables could only pull the effector along a straight line, and the third cable would only increase the control to 2D.

The unique set of cable tensions is calculated by taking the inverse of the matrix in (10) and multiplying the inverse by the desired force vector. This procedure alone does not guarantee that the tensions will be positive. As will be discussed later, the placement of the cables and the force of gravity can be used to ensure positive tensions.

Now, suppose there are four cables in the system. The matrix equation becomes

$$\boldsymbol{F}_{cables} = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = \begin{bmatrix} \frac{x_1 - x}{l_1} & \frac{x_2 - x}{l_2} & \frac{x_3 - x}{l_3} & \frac{x_4 - x}{l_4} \\ \frac{y_1 - y}{l_1} & \frac{y_2 - y}{l_2} & \frac{y_3 - y}{l_3} & \frac{y_4 - y}{l_4} \\ \frac{z_1 - z}{l_1} & \frac{z_2 - z}{l_2} & \frac{z_3 - z}{l_3} & \frac{z_4 - z}{l_4} \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{bmatrix}. \tag{11}$$

Unlike (10), the inverse of the matrix in (11) does not exist because the matrix is not square. The same is true for any system with more than three cables, as the matrix has one column for each cable and three rows, regardless of the number of cables. The non-square matrix means there is not a unique set of cable tensions that will produce the desired force vector. However, there may be an infinite number of solutions. Solutions are found using basic linear algebra methods, some of which are now presented.

The general form of a matrix equation is commonly stated as

$$\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}. \tag{12}$$

In (10−11), $\boldsymbol{A}$ is the direction matrix, $\boldsymbol{x}$ is the vector of cable tensions, and $\boldsymbol{b}$ is the force vector. For convinience, let $\boldsymbol{x}$ be replaced by the symbol $\boldsymbol{t}$ and let $\boldsymbol{b}$ be replaced by $\boldsymbol{F}_{cables}$. If the inverse of $\boldsymbol{A}$ exists, $\boldsymbol{t}$ is directly calculated using

$$\boldsymbol{t} = \boldsymbol{A}^{-1}\ \boldsymbol{F}_{cables} \tag{13}$$

When the exact inverse of $\boldsymbol{A}$ does not exist because it is not a square matrix, for example, a pseudo inverse can be used. Depending on the properties of $\boldsymbol{A}$ and on the method used to calculate the pseudo inverse, (12) may be exactly or approximately satisfied. If there are no exact solutions, one selects a solution that is close to satisfying (12).

One of the common pseudo inverses is known as the Moore-Penrose inverse. If there are multiple solutions, this inverse gives the solution with the minimum 2-norm. If there are no exact solutions, this inverse gives the solution that minimizes the quantity

$$||\boldsymbol{At} - \boldsymbol{F}_{cables}||_2^2. \tag{14}$$

This is also known as the least-squares solution. Note that the closer (14) is to zero, the better (12) is satisfied. In the context of the CSR problem, the Moore-Penrose inverse does not guarantee that the cable tensions are positive.

To enforce that the cable tensions remain positive, (12) can be solved as an optimization problem. For example, one can pose a least-squares problem similar to the Moore-Penrose method but with a nonnegative constraint on the solution. That is,

$$\begin{gathered} \min\ ||\boldsymbol{At} - \boldsymbol{F}_{cables}||_2^2 \\ s.\ t. \\ \boldsymbol{t} \geq 0. \end{gathered} \tag{15}$$

If it is known that infinite solutions exist, another way to pose the problem is to minimize some objective function such that (12) is satisfied exactly. For example, the sum of tensions can be minimized. That is,

$$\begin{gathered} \min \textstyle\sum_i t_i \\ s.\ t. \\ \boldsymbol{At} = \boldsymbol{F}_{cables} \\ \boldsymbol{t} \geq 0. \end{gathered} \tag{16}$$

Again, this formulation includes a nonnegative constraint for the solution. Similarly, one could impose a minimum or maximum tension. Another possibility is to minimize the tension of a select number of cables. For example, [24] suggests minimizing the tensions of the two longest cables, since these are generally expected to experience larger tensions.

Suppose that the cable tension is created by an electric motor. Then, the tension is proportional to the motor current. In turn, the electrical power is proportional to the square of the current. Therefore, the power required to create the cable tension using an electric motor is approximately proportional to the square of the tension. This reasoning gives another method for optimizing the CSR tensions. That is, minimize the sum of the squared tensions in order to minimize the total electrical-power consumption. In standard form,

$$\begin{gathered} \min \textstyle\sum_i t_i^2 \\ s.\ t. \\ \boldsymbol{At} = \boldsymbol{F}_{cables} \\ \boldsymbol{t} \geq 0. \end{gathered} \tag{17}$$

Once again, the constraints include an exact solution to (12) and nonnegative terms for $\boldsymbol{t}$. A full discussion of the CSR actuators is presented in Section 3.1.7.

In a three-cable system, a unique tension solution is guaranteed by the proper placement of the cable anchor points. However, there is no guarantee of those tensions being positive for every resultant-force vector. When four or more cables are used, the system is over-constrained. Then, an infinite number of tension solutions is desired, and a solution with positive tensions is selected using an objective criterion, like the ones in (16−17). In any case, it is useful to guarantee that solutions with positive tensions exist. In some circumstances, this can be achieved by placing restrictions on the force vector.

In light of the discussion in Section 3.1.1, consider the circumstance where all cable anchor points are higher than the end-effector. Then, the cables can pull the effector away from the ground but not push the effector towards the ground. So, any force vector with a negative *z* component cannot be supported. This fact is easily seen in (9), as the third element of the vector in the sum will always be positive, and any set of positive cable tensions will produce a nonnegative *z* force.

Now, consider the circumstance where all cable anchor points are higher than the end effector and the CSR is in static equilibrium. For static equilibrium, the *x*- and *y*-force components must be zero, and the *z* component must be equal to the weight of the mass. If the end effector is in the convex hull of the cable anchor points, as discussed in Section 3.1.2, then it is always possible to select a set of positive tensions to produce net-zero *x* and *y* forces. To show this, suppose that all but three cable tensions are set to zero. Then, the three tensions can be set to produce a net-zero *x*-*y* force (Figure 32). Furthermore, multiplying the same cable tensions by any scalar would still result in the net-zero *x*-*y* forces. Then, an appropriate scalar can be selected to produce the required *z* force for static equilibrium. Note that any number of cables can be combined to produce a net force equivalent to that of a single cable. Therefore, a three-cable equivalent system can always

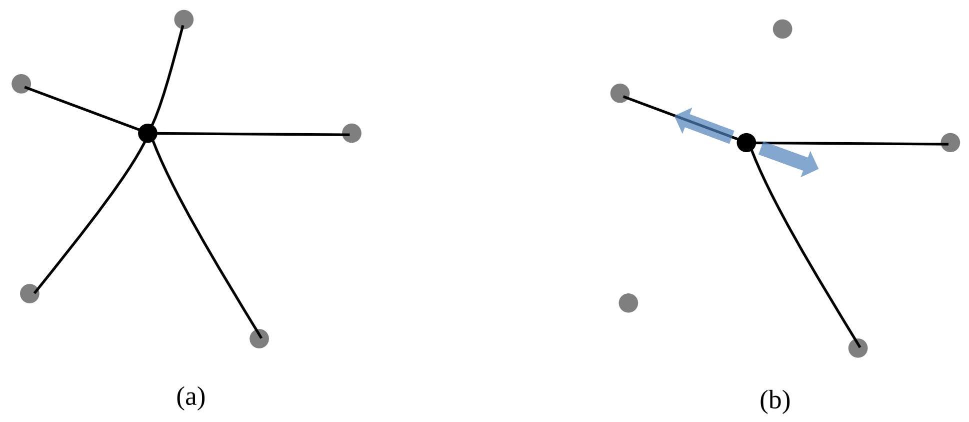
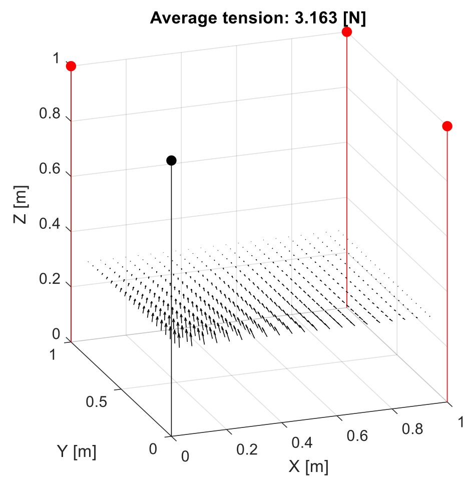


Figure 32. (a) Top view of a five-cable CSR system. (b) The same system where all but three cable tensions are set to zero. The *x*-*y* force of one of the cables, shown as an arrow pointing to the left, is exactly counteracted by the net *x*-*y* force of the other two cables, shown as an arrow pointing to the right.

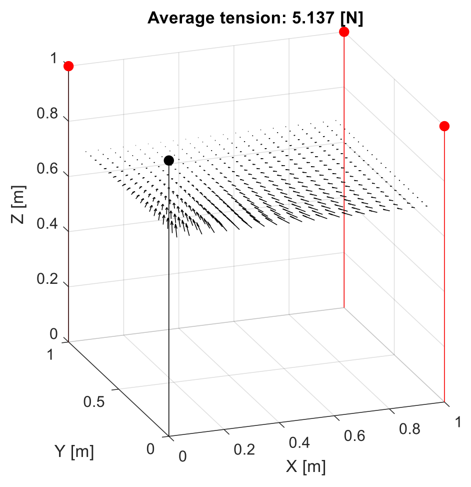
be created, regardless of the number of cables in the system, and the above result always applies. It is not necessary for only three cables to be in tension.

The requirement for the end effector to be in the convex hull of the anchor points ensures that the cable tensions can counteract each other in the *x*-*y* directions to maintain a static equilibrium. In light of the result described in Figure 32, this requirement can be made even more specific. That is, to allow for static equilibrium of the CSR the end effector should be in the convex hull of at least three anchor points.

Using the information presented here, tension simulations at various SEPs were completed in MATLAB. Figure 33 shows the first set of plots that compares the cable tensions for a four-cable CSR at two heights. The configuration is a cube with 1-m sides, and gravity acts on the 1-kg point mass. Method (15) was used to calculate the cable tensions at equally-spaced points for each height. For clarity, the tension vectors for only one of the cables is shown. Due to the symmetry, the other three cables exhibit the same



(a)

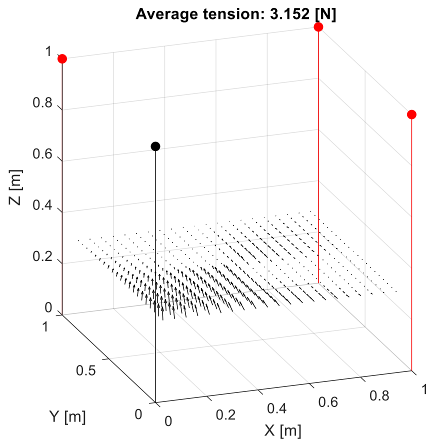


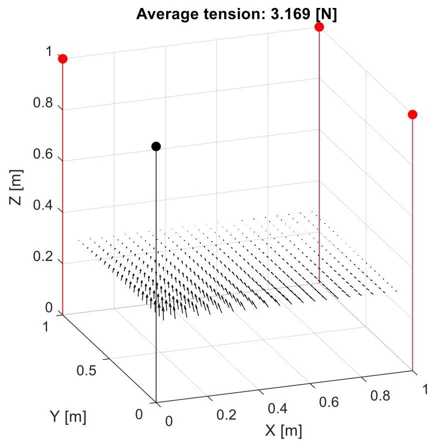
(b)

Figure 33. Cable-tension vectors for a four-cable CSR at static equilibrium. Equally-spaced points are sampled along a 2D plane at (a) 0.3-m height and (b) 0.7-m height. For clarity, only the tension vectors for the cable originating above the origin are shown.

tensions, from the viewpoint of the anchor points. In general, the cable tensions are larger at the greater height. This is because the *z* component of the cable tensions decreases as the mass is raised, so the tension magnitudes must increase to produce the necessary *z* force that supports the mass. Also, the points closer to the anchor point have a larger tension. Thus, the cable shares a higher percentage of the load as the mass moves closer to the anchor point. If the mass were directly under the anchor point, that cable would support the entire load. Also, the cable has a positive tension everywhere along the grid of points, meaning that it always supports the load. Finally, the trend of cable tensions is smooth. This is significant because if the mass were to slowly transition from one SEP to another the cable tension would not change suddenly.

Figure 34 shows the cable tensions at static equilibrium using (16−17). Again, only the tension vectors for one cable are shown, since the other cables exhibit the same results. As in Figure 33, the tension magnitude increases as the point mass gets closer to the anchor point. However, there is a clear difference in the tensions elsewhere. When (16) is used, the tension sum is minimized. This results in regions where the cable has zero tension, as shown in Figure 34(a). Tensions of zero magnitude could be detrimental to the performance of the system, since a cable with zero tension could have a little or a lot of slack, and there may be a noticeable delay in increasing the tension if required. The cable tension when using (17), shown in Figure 34(b), looks similar to Figure 33 but with slightly larger tensions near the cable anchor point and some zero-tension points in the opposite corner. This calculation method minimizes the sum of the squared tensions, thereby the power consumption by the motors, and it produces a smooth tension trend without large regions of zero cable tension. Thus, this method is also favorable for control.

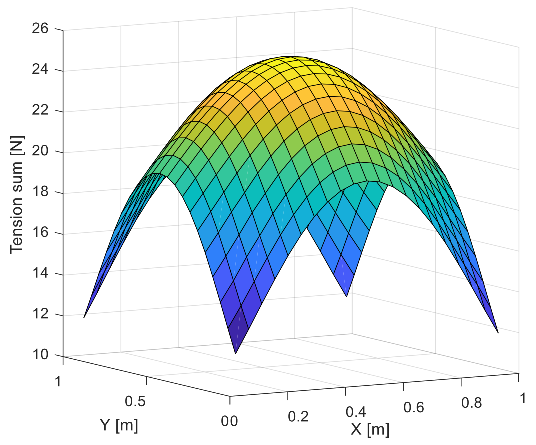




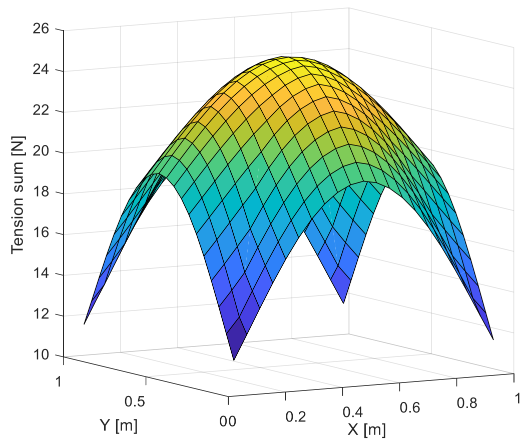
(a) (b)

Figure 34. Cable tension (a) minimizing the sum of tensions and (b) minimizing the sum of squared tensions.

The results from using (16) or (17) to calculate the cable tensions is more clearly seen by calculating the optimized objective function at each tension-sample point. Method (15), which is the ordinary least-squares solution with a nonnegative constraint, is used as a baseline to compare against. In Figure 35, the tension sums are plotted using (15) and (16) for the height of 0.7 m. Method (16) does decrease the tension sums, especially near the anchor points, but the result is nearly indistinguishable from using (15). The difference between the two methods is even smaller at lower heights. Since minimizing the tension sum does not significantly change the results, and since (16) gives large regions of zero cable tension, (15) is the preferred method. Figure 36 shows the sum of squared tensions using (15) and (17) at the height of 0.3 m. Again, minimizing the sum of squared cable tensions does have an effect, but the effect is hardly noticeable. The same is true at greater heights. However, (17) does produce smooth tension trends (Figure 34(b)) so it can still

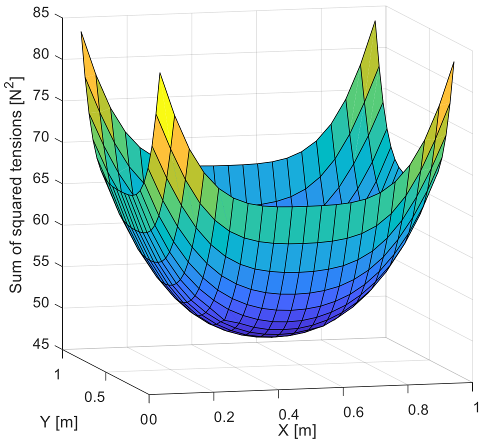


(a)

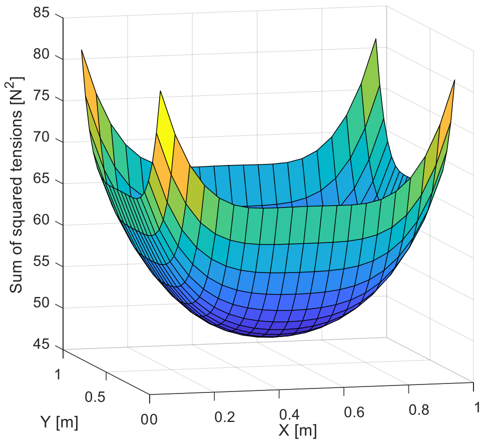


(b)

Figure 35. The sum of the four cable tensions for a CSR at static equilibrium. The configuration is the same as that in Figure 34, and the sampled points are at 0.7-m height. Plot (a) uses method (15) to calculate the tensions, and (b) uses method (16).



(a)

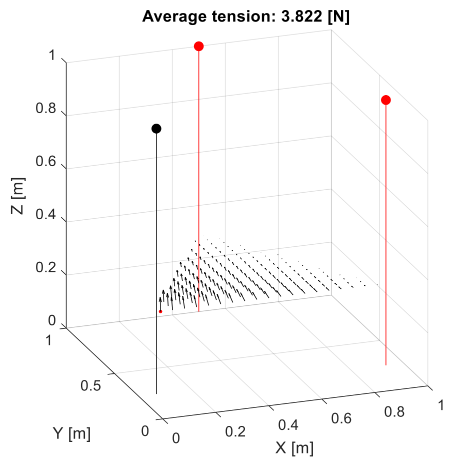


(b)

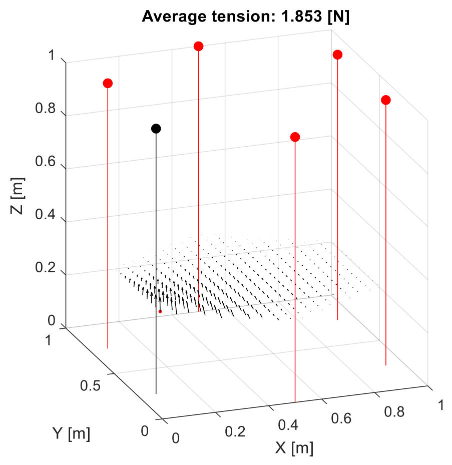
Figure 36. The sum of the four squared cable tensions for a CSR at static equilibrium. The configuration is the same as that in Figure 34, and the sampled points are at 0.3-m height. Plot (a) uses method (15) to calculate the tensions, and (b) uses method (17).

be used in hopes of slightly reducing electrical power consumption in the CSR.

For comparison, static-equilibrium tensions were also calculated for three- and six-cable systems (Figure 37). The configurations were set so that they fit within the same area as the square configuration of Figure 33, and the same height of 0.3 m was used. In the case of three cables the tensions are unique for every position, and the calculation involves solving (10) using the true matrix inverse. Since less cables carry the weight of the mass, the tensions are generally higher than that of the four-cable system. All three cables are in tension for every point. As before, the cable tension is larger as the point mass gets closer to the anchor point. The six-cable configuration exhibits the same trend. In that system, the larger number of cables means that, on average, the cable tensions are smaller than the three- and four-cable systems. The method in (15) was used to calculate the tensions in the six-cable system. Once again, the results show that, using this method,



(a)



(b)

Figure 37. The cable tensions for a CSR in static equilibrium at 0.3-m height. (a) A three-cable system where there are no redundant cables and the cable tensions are unique. (b) A six-cable system where the cable tensions were calculated using (15).

the cable remains in tension for all of the sample points. This is a favorable result, as the cable tensions are more predictable when the cables remain without slack.

One may also be interested in the circumstance where the system is not in static equilibrium. After all, a net force is required to produce movement of the mass. Calculating the cable tensions is the same as before. However, the guarantee nonnegative cable tensions is more complicated. To guarantee a solution, restrictions can be placed on the desired force vector. As before, consider the case where all cable anchor points are higher than the end effector. One restriction already mentioned is that the *z* force must be nonnegative. Now, suppose that there is some desired *x*-*y* force. In general, only two cables are needed to create this force (Figure 38). If only those two cables are in tension there will still be a net-positive *z* force on the mass. This is the minimum allowable *z* force. There is no way to produce a smaller *z* force without having some negative tensions.

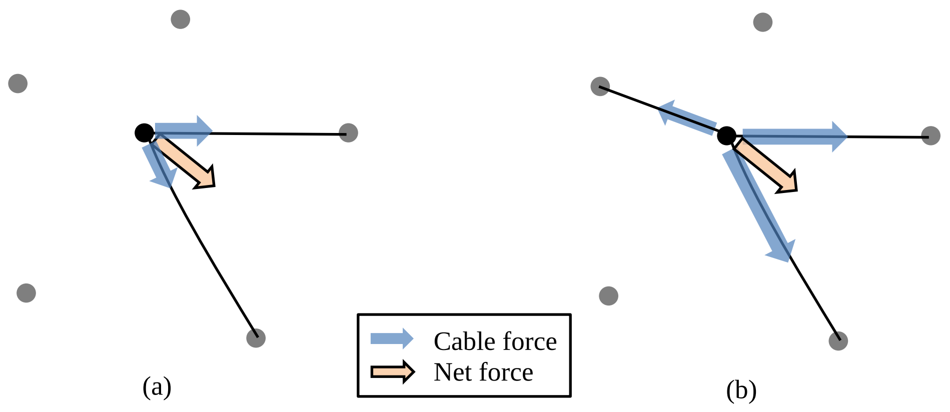


Figure 38. Top view of a five-cable CSR. (a) A general *x*-*y* force can be produced by two cables in the system, with all other cables having zero tension. (b) The same net *x*-*y* force can be produced by increasing one of the other cable tensions and adjusting the cable tensions of the original two cables. The *z* force will be greater than before.

However, the *z* force could be increased by raising the original two tensions along with some of the other cable tensions, all while maintaining the required *x*-*y* force.

As previously mentioned, a general CSR can always be simplified to an equivalent three-cable system by combining some cable tensions. Therefore, the result of Figure 38 can be applied to a CSR of any number of cables. However, different cable combinations may yield a different minimum *z* force. One may be interested in the absolute minimum-allowable *z* force among all the possible cable combinations. This problem is solved by

$$\begin{gathered} \min \boldsymbol{A}_z \boldsymbol{t} \\ s.\ t. \\ \begin{bmatrix} \boldsymbol{A}_x \\ \boldsymbol{A}_y \end{bmatrix} \boldsymbol{t} = \begin{bmatrix} F_x \\ F_y \end{bmatrix} \\ \boldsymbol{t} \geq 0, \end{gathered} \tag{18}$$

where $\boldsymbol{A}_x$, $\boldsymbol{A}_y$, and $\boldsymbol{A}_z$ are the first, second, and third rows of the cable direction matrix, respectively. The objective function in (18) is the net *z* force of the cable tensions, and the first constraint restricts the tensions to produce the desired *x*-*y* force. As the magnitude of the *x*-*y* force increases, the minimum *z* force also increases. As the *x*-*y* force decreases, the minimum *z* force also decreases. When the *x*-*y* force is zero, the minimum *z* force is zero.

Figure 39 shows one example of the maximum *z* force for the four-cable robot pictured in Figure 33. To create this plot, (18) was repeatedly applied at the arbitrary robot position (0.25, 0.75, 0.3) m using different positive and negative *x*-*y* forces each time. The force values are plotted as a percent weight of the point mass. The result confirms that the minimum *z* force is zero when the *x*-*y* force is zero. Furthermore, the slope of the plot in the *x* and *y* directions is indicative of the cables that, when in tension to produce the required *x*-*y* force, also give the minimum *z*-force allowed. For example, for positive-*x* forces the slope of the plot is nearly one (Figure 39(b)). This is because a positive *x* force

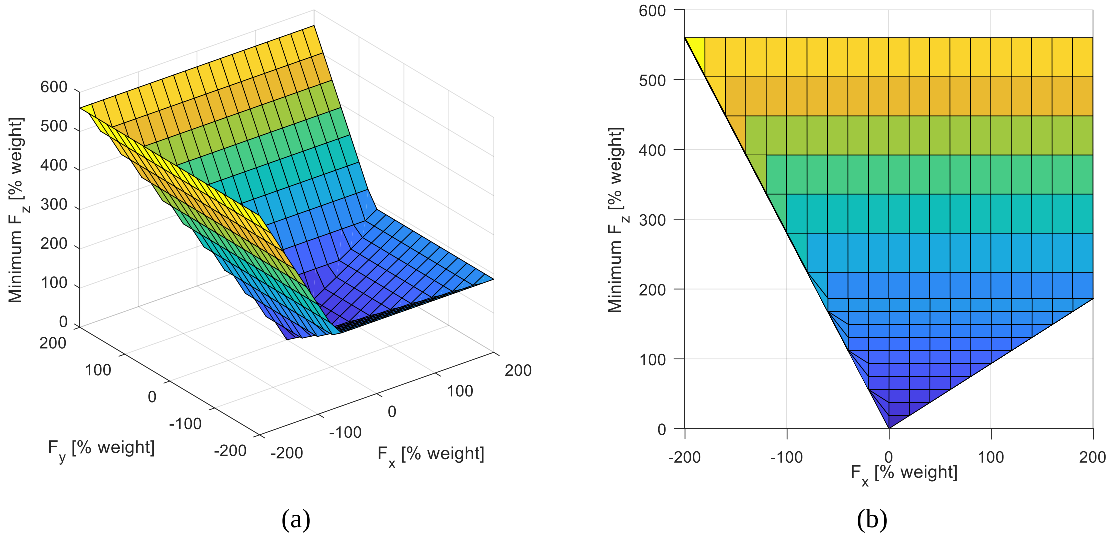


Figure 39. (a) Minimum-required *z* force for a four-cable CSR vs. the desired *x* and *y* forces. (b) Side view of the same result. These results were calculated at the position (0.25, 0.75, 0.3) m for the same configuration as Figure 33.

must be supported, at minimum, by the two cables that pull the mass to the right. From the cable direction matrix of the CSR, one can calculate that the ratio of *z*-to-*x* force provided by those two cables is 0.933, which is exactly the slope of the plot in the positive-*x* direction. In the opposite direction, the two left cables are required to provide a negative-*x* force. For these cables, the ratio of *z*-to-*x* force is −2.8, which is the slope of the plot in the negative-*x* direction. In practical terms, (18) gives the minimum *z* force that must be applied on the end effector given a required *x*-*y* force. Furthermore, the answer to this question is a linear relationship encoded in the cable-direction matrix. For example, for the test point in Figure 39, every 1 N of *x*-direction force requires an additional *z* force of 0.933 N to be applied to the mass. Again, all of this is to ensure that the cable tensions remain nonnegative. Note that for small *x*-*y* forces the minimum *z* force is also small. Therefore, it is expected that slow movements, such as those typical of an AM application, will not have a problem with maintaining positive cable tensions, since the CSR *z* force will remain near 100% of the end effector weight and the minimum *z* force for slow *x*-*y* movements will be near zero.

The problem of restricting the force vector to ensure positive cable tensions can be approached the other way around. That is, if the required *z* force is known then restrictions can be placed on the *x*-*y* force components. Since the *x*-*y* forces can be positive or negative, restrictions are placed on the magnitude of the *x*-*y* force vector. It was already stated that a system with net-zero force in the *x* and *y* directions can support any *z* force, so the minimum *x*-*y* force magnitude is always zero. The maximum is found by solving

$$\max \left\| \begin{bmatrix} \boldsymbol{A}_x \\ \boldsymbol{A}_y \end{bmatrix} \boldsymbol{t} \right\|_2 \tag{19}$$

$$\begin{gathered} s.\ t. \\ \boldsymbol{A}_z \boldsymbol{t} = F_z \\ \boldsymbol{t} \geq 0 \end{gathered}$$

Here, the objective function is the magnitude of the *x*-*y* force, and the first constraint is that the tensions produce the required *z* force. As F_z decreases, the cable tensions must be smaller and, as a result, the maximum *x*-*y*-force magnitude also decreases. In the extreme case, when F_z is zero, the maximum *x*-*y* force is also zero. Physically, a zero F_z means that the cables provide no tension at all, and the mass is in free fall. It is obvious that no *x*-*y* force can be applied in that situation. On the other hand, a large F_z requires large cable tensions, which provides the opportunity for the cables to simultaneously produce a large *x*-*y* force.

Figure 40 gives an example of the maximum *x*-*y* force of a four-cable CSR given the required *z* force. This plot was created by applying (19) at the arbitrary position (0.25, 0.75, 0.3) m for the CSR pictured in Figure 33 and varying the required *z* force. The forces

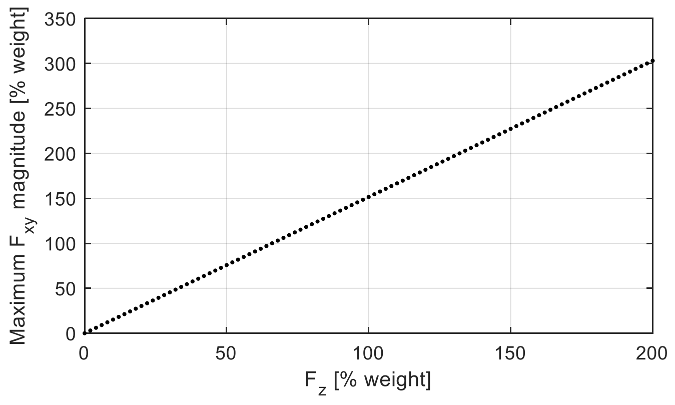


Figure 40. The maximum allowable *x*-*y* force for a four-cable CSR given the required *z* force. These results were calculated for the same configuration and test position as Figure 40.

are plotted as percentages of the mass weight. As the *z* force increases, so does the maximum *x*-*y* force magnitude. When the required *z* force is zero, which is to say that none of the cables are in tension, the *x*-*y* force must also be zero. The slope of the plot indicates the cable that gives the maximum *x*-*y* force when that cable, alone, provides the required *z* force. In this case, the cable anchored at (1.0, 0, 1.0) m has the maximum x-y-to-z force ratio of 1.51, which is the slop of the plot. In practical terms, this result means that for every 1 N of *z* force required, the maximum additional *x*-*y* force that can be provided by the CSR is 1.51 N in the direction of the cable anchored at (1.0, 0, 1.0) m. For a CSR used in an AM application, the movements are slow and mostly along a constant height, as layers of material are deposited. For this type of movement, the required-*z* force is nearly 100% of the end-effector weight. Then, the maximum *x*-*y* force is also a sizeable number. Since only small forces are required to produce the slow movements, these results suggest that a CSR for AM should have no trouble in maintaining positive cable tensions.

In summary, (15) or (17) can be used to calculate the required CSR cable tensions given an instantaneous position and desired net force on the end effector. Furthermore, the slow movements typical of an AM application mean that the cable tensions are expected to remain positive during nominal CSR operation.

3.1.5. Stiffness Analysis

In simple words, stiffness is resistance to deflection. In order to resist external disturbances, a robot must have some stiffness. A higher stiffness is usually desirable since it means that it takes a larger force to deflect the robot by a certain amount. A stiffness analysis reveals some of the consequences of using flexible cables in a CSR.

The basic definition of stiffness can be stated as

$$K = \frac{F}{\delta}, \tag{20}$$

where F is a force that causes the displacement, δ. If one was to experimentally determine the stiffness of a robot at its end effector, a force could be applied to the end effector and the resulting displacement measured. Then, (20) would give the stiffness value. However, the value can depend on the direction of the force and on the position of the effector. Therefore, a more specific stiffness definition is needed. For example, the stiffness in the x direction can be stated as

$$K_x(x, y, z) = \frac{F_x}{\delta_x}. \tag{21}$$

The stiffness in the y and z directions are similarly defined. This equation recognizes that the stiffness depends on the position of the end effector and on the direction of the force. Since the stiffness varies with position, the displacement, δ_x, is an infinitesimal quantity.

The CSR stiffness at static-equilibrium will now be investigated. Let the CSR point-mass begin at rest at some initial position. A method like those in Section 3.1.4 can be used to calculate the cable tensions at this initial position. Then, let the mass be displaced by a small amount. Some force will be required to produce this displacement. The approximate stiffness is the applied force divided by the displacement. The exact stiffness is the limit of this calculation as the displacement goes to zero. The immediate task at hand is finding an expression for the force in terms of the displacement.

At any equilibrium position, the mass is supported by a set of cable tensions. These tensions are caused by the individual cables stretching by some small amount. A displacement of the end effector will produce a change in the cable stretch amounts and, consequently, the cable tensions. Therefore, the displacement amount can be related to the change in cable tensions. The key to this relationship is the cable-tension function, which is discussed in Section 3.2. For now, the function is assumed to have form

$$T(l, l_0) = \alpha(l - l_0), \tag{22}$$

where α is some stiffness coefficient, l is the stretched cable length, and l_0 is the unstretched cable length. If the cable tension is not a linear function, it can be linearized locally. At the initial equilibrium point, the stretched-cable length is calculated using (3) and the cable tension is known. Therefore, the unstretched-cable length can be found using (22). After a small displacement the new stretched-cable length is l'. The unstretched-cable length remains the same as before, and the new cable tension, T', can be calculated. Finally, the force required to produce the small displacement can be calculated from the difference between the original and new cable tensions.

One example, using the *x*-direction stiffness, is now presented. A point mass at some initial position is supported by a cable that is anchored at (x_c, y_c, z_c), as pictured in Figure 41. The initial, stretched-cable length is

$$l = \sqrt{(x_c - x)^2 + \left(y_c - y\right)^2 + (z_c - z)^2}. \tag{23}$$

After displacing the mass in the positive-*x* direction, the new stretched-cable length is

$$l' = \sqrt{(x_c - x - \Delta x)^2 + \left(y_c - y\right)^2 + (z_c - z)^2}. \tag{24}$$

The cable's tension before and after the displacement is T and T', respectively. To calculate the stiffness in the *x* direction, the *x*-direction force is needed. Applying (9), the *x* force is

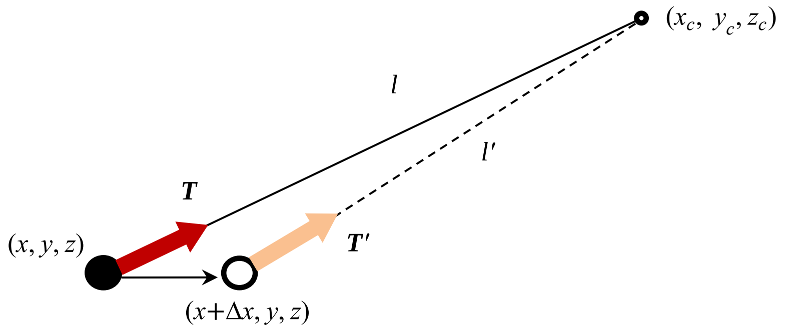


Figure 41. A point mass supported by a cable. The solid line is the original placement of the cable, and the dotted line is the same cable after displacing the mass in the positive *x* direction. The tension vectors before and after the displacement are shown as arrows.

found to be

$$F_x = T_x - T'_x = T\left(\frac{x_c - x}{l}\right) - T'\left(\frac{x_c - x - \Delta x}{l'}\right). \tag{25}$$

Now, applying (22), the force is written as

$$F_x = \alpha(l - l_0)\left(\frac{x_c - x}{l}\right) - \alpha(l' - l_0)\left(\frac{x_c - x - \Delta x}{l'}\right). \tag{26}$$

Equation (26) is the expression that relates the *x* force to the *x* displacement. According to the definition in (21), the approximate *x*-direction stiffness for the displacement, Δ*x*, is

$$\widetilde{K_x} = \frac{F_x}{\Delta x} = \frac{\alpha(l - l_0)}{\Delta x}\left(\frac{x_c - x}{l}\right) - \frac{\alpha(l' - l_0)}{\Delta x}\left(\frac{x_c - x - \Delta x}{l'}\right) \tag{27}$$

and the exact stiffness is the limit of (27) as Δ*x* goes to zero. After performing some algebra and taking the limit, the result is

$$K_x = \alpha\left(1 - \frac{l_0}{l} + \frac{l_0(x_c - x)^2}{l^3}\right). \tag{28}$$

Note that (28) depends on the stretched and unstretched cable lengths. The expression also depends on the *x* coordinates of the point mass and cable anchor point. At first glance, it appears that the expression does not depend on the *y* and *z* coordinates. However, all three

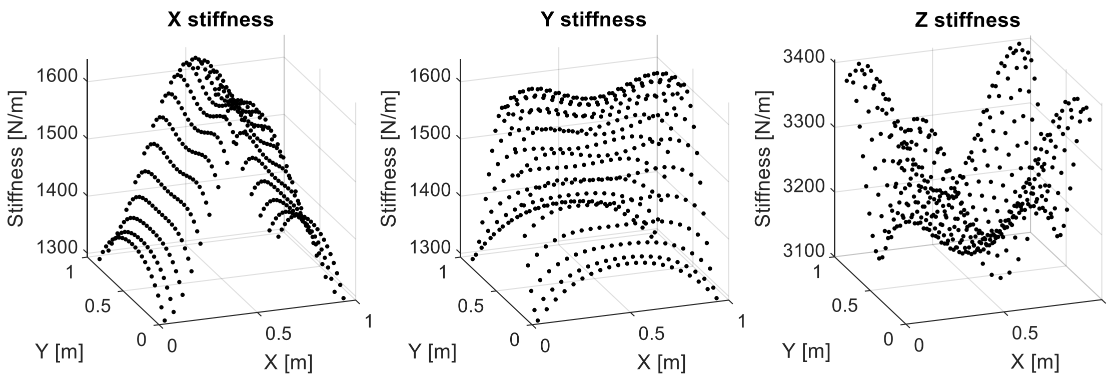
Cartesian coordinates are needed to determine l and l_0 for a given CSR position. Furthermore, (28) is the stiffness contribution for just one cable in the system. The expression must be applied for each cable, and the total stiffness is the sum of the individual contributions. In summary, the x stiffness of the CSR end effector at a certain point in the 3-D space is calculated by following these steps:

1. Calculate the cable tensions using a method like those described in Section 3.1.4.
2. Find the unstretched cable lengths. To do this, apply the cable-tension function to each cable. The cable tensions are known from step 1, and the stretched cable lengths are calculated using (23). Then, the unstretched cable length is the only unknown in the cable-tension function and can be solved for.
3. Calculate the x-direction stiffness contribution for each cable using (28). Sum all of the contributions to find the total x-direction stiffness.

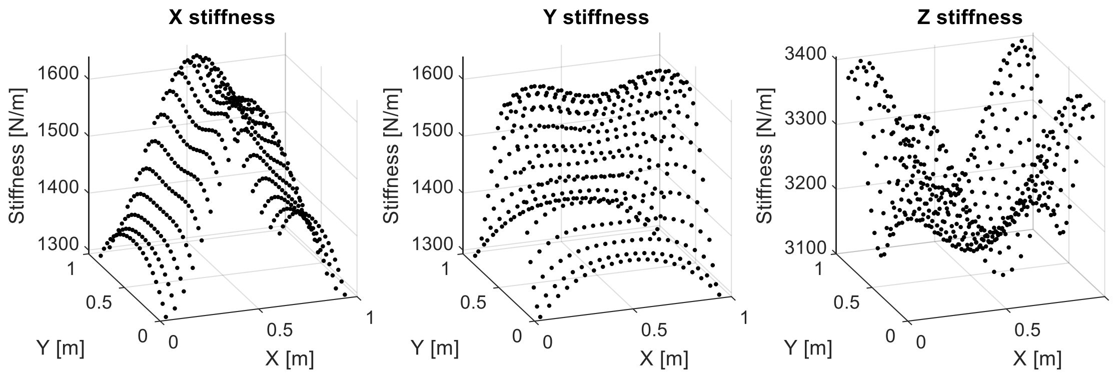
If the displacement in Figure 41 had been in the negative-x direction, the final stiffness equation would be the same, owing to the fact that the limit of (27) is the same from the left and the right of Δx equaling zero. The y- and z-direction stiffnesses are like (28) but with the y and z coordinates in the squared term, respectively. Figure 42 gives an example of the directional stiffness for a four-cable CSR. The same configuration as Figure 33 is used, where the work region is a cube with 1-m sides and the end effector is a 1-kg point mass. The cable stiffness was modeled using Hooke's Law for a rod, such that a longer or thinner cable has a smaller stiffness. For this example, the elastic modulus of A36 steel, which is 200 GPa, was used along with a circular cable diameter of 0.1 mm. The stiffness of each cable is calculated using

$$\alpha = \frac{EA}{l}, \tag{29}$$

where E is the cable elastic modulus, A is the cross-sectional area, and l is the length. Note that the stiffness is proportional to the length and the area of the cable, so changing either of these values would simply scale the results in Figure 42. Nevertheless, these plots are used to identify some trends in the CSR stiffness. The stiffness values were calculated at the height of 0.3 m, the same as Figure 34. Also, two different methods for calculating the cable tensions were tested. Recall that the cable tensions are calculated first to determine



(a)

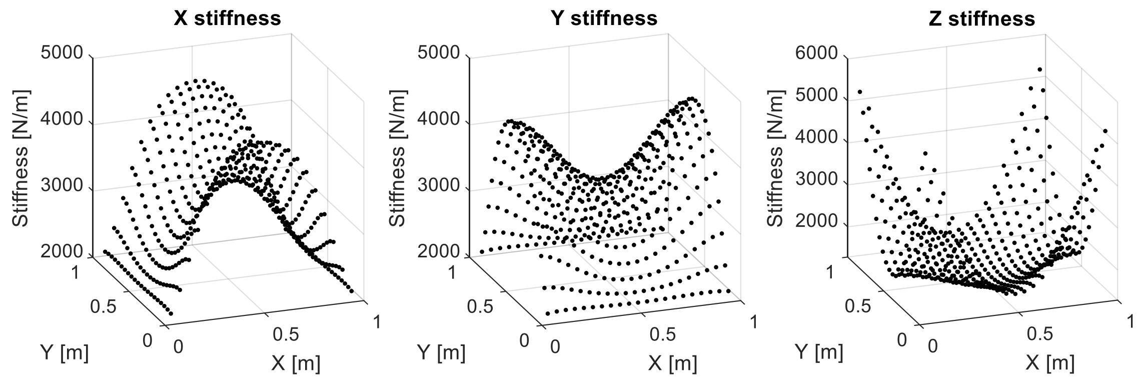


(b)

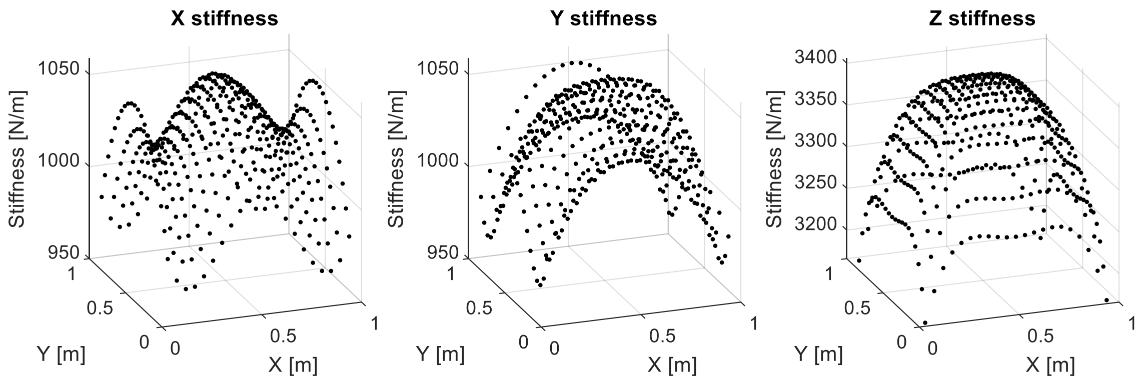
Figure 42. (a) Directional stiffness values for a four-cable CSR at 0.3-m height, using tension method (15). (b) The stiffness values for the same configuration but using method (17). The results are visually identical.

the cable stretch. Then, the stiffness can be found. In Figure 42(a), the cable tensions were calculated at static equilibrium using method (15), which is the least squares solution with the nonnegative tension constraint. In Figure 42(b), the tensions were calculated using method (17), which minimizes the sum of squared tensions. It was shown in Section 3.1.4 that these two methods gave similar results. Therefore, the two sets of stiffness plots in Figure 42 are visually identical. For the remaining stiffness plots, method (15) is used.

Some trends in Figure 42 are now described. First, the *x*- and *y*-direction stiffnesses are the same but rotated. This is expected since the configuration is symmetric with respect to the *x* and *y* directions. For the *x* and *y* directions, the stiffness is generally higher near the center of the 3-D space. Nevertheless, the stiffness remains within the same order of magnitude. The *z*-direction stiffness is about twice as much as the *x* and *y* directions, and the maximum values appear close to the anchor points. Again, the range between the maximum and minimum stiffness values for the *z* direction is small, only about 10% of the average value. From (28), one can see that, in general, the stiffness is increased by shorter cable lengths, a position farther from the anchor points, and a smaller cable stretch. Figure 43 shows the stiffness values at a higher and lower height. At the 0.7-m height, the cables are much shorter and all of the stiffnesses increase substantially. However, the cable stretch is greater near the center of the *x*-*y* area and the stiffness decreases noticeably. At the 0.3-m height, the cable lengths are larger, so the stiffness values are lower. However, near the center of the *x*-*y* area, the cable stretch is small and the stiffness increases. The amount of increase is not significant, however. The greater height of 0.7 m produced a higher variation in stiffness values, as well as the highest values overall. For the purpose



(a)

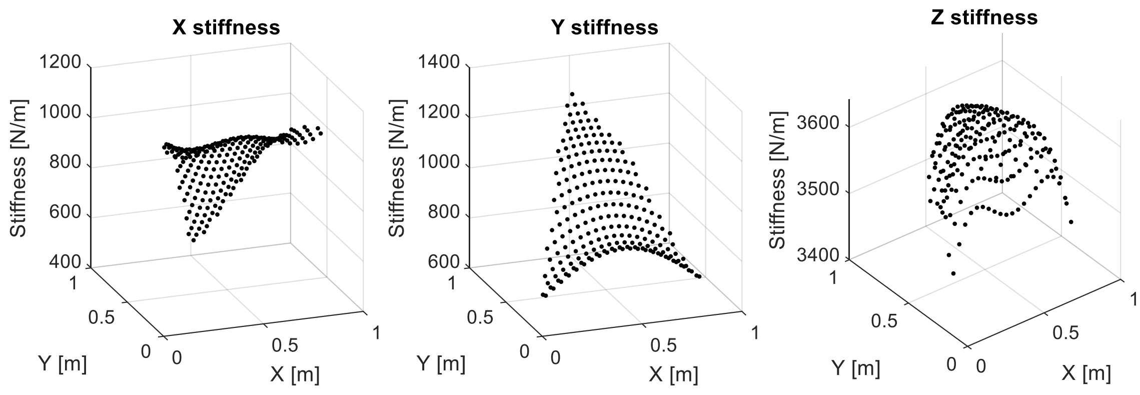


(b)

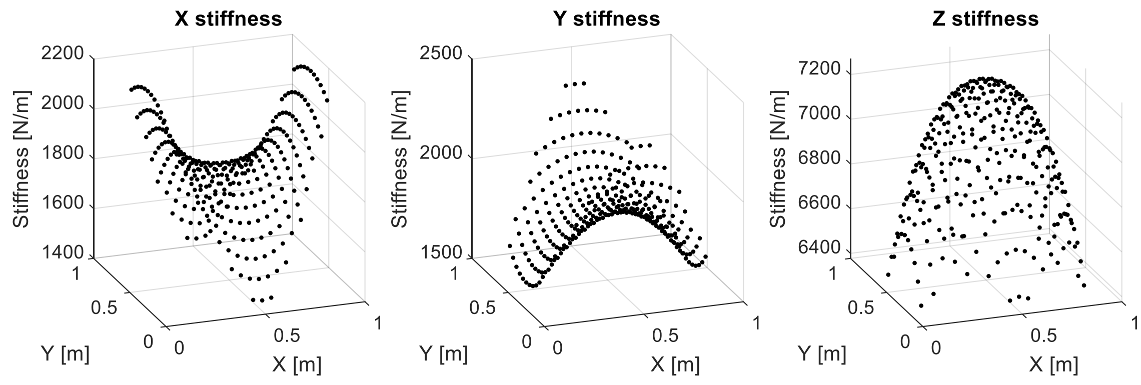
Figure 43. (a) Directional stiffness values for a four-cable CSR at 0.7-m height. (b) The stiffness values for the same configuration but at 0.1-m height.

of design, if a minimum CSR stiffness is desired, a group of test points at a lower height can be used to ensure that the minimum stiffness is met. Then, the other points in the workspace can be expected to have a higher stiffness.

For comparison, Figure 44 shows the directional stiffness values for a three- and six-cable system. The configurations are the same as in Figure 37, and the cables are modeled the same as those in the four-cable system. The stiffness values are calculated at the 0.3-m height for direct comparison with results for the four-cable configuration in Figure 42. The stiffness values are smaller for the three-cable system owing to the larger



(a)



(b)

Figure 44. (a) Directional stiffness values for a three-cable CSR at 0.3-m height. (b) Directional stiffness values for a six-cable CSR at 0.3-m height.

cable stretch amounts necessary for less cables to support the same weight. The three-cable configuration also exhibits a much more irregular trend in the *x* and *y*-direction stiffnesses, with a large range between the maximum and minimum values and a lack of symmetry. The six-cable system is more regular and symmetric, with larger stiffness values, due to the smaller cable stretch. In fact, the average *z* stiffness is approximately twice as large for the six-cable configuration as it is for the four-cable system.

In summary, the CSR generally exhibits the lowest directional stiffnesses at lower heights, and the overall stiffness can be increased by increasing the cable stiffness, increasing the number of cables or decreasing the end effector mass.

3.1.6. Sensors

Effective control of a CSR, like any other robot system, depends on the use of high-quality sensors. The sensors can be placed into one of three groups. The first group includes the sensors that are used to monitor the cable states, like their lengths or their tensions. The second group includes the sensors that are used to directly measure the position of the end effector. The final group includes any sensors that detect changes in the robot configuration. For example, these sensors could measure the exact position of the cable anchor points.

The first group of sensors includes those that monitor the cables states. These sensors are the most essential for basic operation of the CSR because they are closely associated with the robot kinematics and dynamics. As discussed in Section 3.1.2, the robot kinematics describes the relationship between the cable lengths and the end-effector position. One of the main functions for this group of sensors is to accurately measure the cable lengths so that the end-effector position can be estimated.

The sensor of choice for measuring the cable lengths is the rotary encoder. One of the most practical ways to manage the CSR cables is to store them in a spool, and then unwind the spool as an additional length is needed. The cables can also be directed using low-friction pulleys. Any such spool or pulley can serve as an attachment point for a rotary encoder (Figure 45). The encoder registers the shaft rotation, converting the movement to

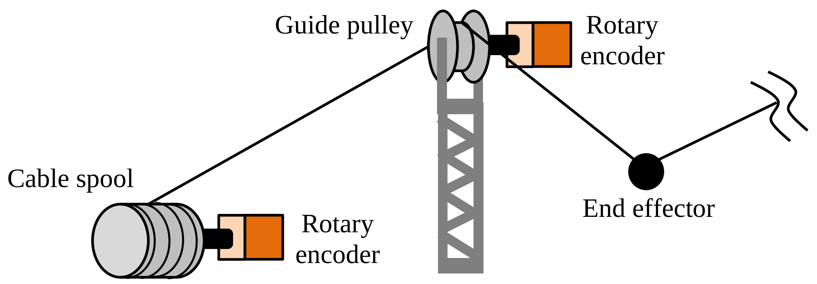


Figure 45. Some locations for a rotary encoder. One location is at the cable spool and the other is at the guide pulley. The rotary encoder is used to measure the cable length.

an electrical signal that can be decoded by a microcontroller board. Encoders vary in their internal mechanism and the electrical signals they produce. Most commonly, a relative (or incremental) encoder with a digital output is used. In this type of encoder, incremental changes in the angular position of the shaft are converted to binary electrical pulses. Then, a specialized circuit keeps the count of the pulses to calculate the rotation amount.

One quantity of interest relevant to an encoder is its resolution. The sensing resolution is the smallest-detectable amount of the shaft rotation. Ultimately, this quantity dictates how accurately the changes in the cable length can be measured since the change in the cable length is proportional to the shaft rotation. The encoder resolution is commonly provided in terms of pulses per revolution (PPR) of the encoder shaft. For example, an encoder with 360 PPR provides one electrical pulse for each degree of the encoder rotation. Moreover, some encoders provide multiple channels of such pulses, so the maximum detectable count for each shaft rotation can be multiple times the PPR. The encoder channels are typically out of phase by 90°, or one quadrature, to allow for detecting the direction of the shaft rotation. For a CSR, it is necessary to accurately detect the shaft rotation and direction. Then, the change in cable length is the rotation amount times the spool (or pulley) radius. In this respect, the radius should remain constant or be accurately known at all times. Otherwise, there will be systematic errors in the cable-length calculation.

Another important quantity is the encoder slew rate. In short, the slew rate is the maximum rotation speed that the encoder can accurately register. If not properly selected, the encoder could give false readings of the shaft movement, which will result in an inaccurate calculation of the cable lengths. In some cases, it may be necessary to sacrifice sensing resolution in order to achieve the desired encoder slew rate. Such determination should be made considering the expected working conditions of the CSR. For example, in a CSR for AM all of the movements must be relatively slow. Then, the required slew rate will be lower, and a high sensing resolution will be prioritized. In any case, it is also important to consider the frequency limitations of other system components, such as the microcontroller. The microcontroller's sampling frequency should be high enough to read the encoder pulses, and the microcontroller's clock speed should be fast enough to perform calculations between encoder pulses.

In some cases, commercial encoders can come with optional signal decoders or associated electronics that ensure their proper operation. The selection of the encoders for a specific CSR design is a task that is left for future work. Ultimately, the objective is to accurately measure the cable lengths under all of the expected working conditions of the CSR. One example of encoder selection and implementation is presented in the experiment of Chapter 4.

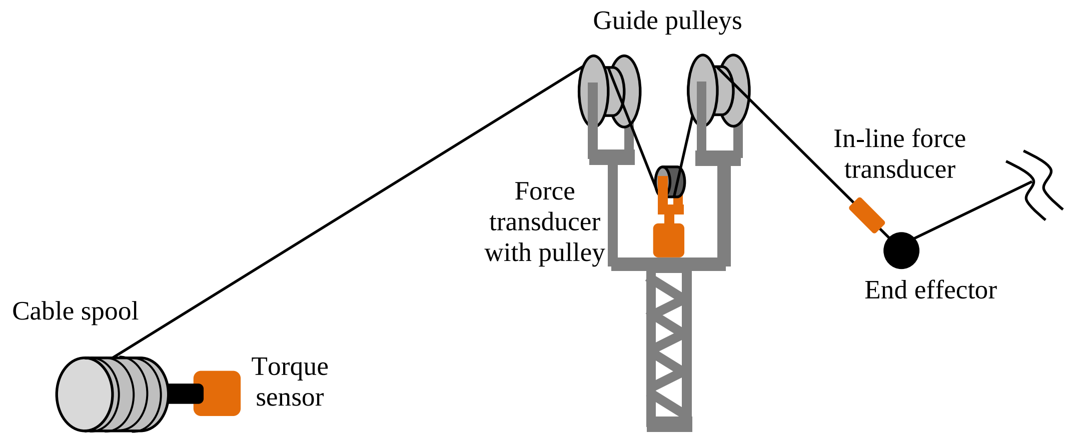


Figure 46. Some possible locations for a cable-tension sensor. One location is at the cable spool, where a torque sensor measures the torque on the spool. Another location is at a specialized pulley location. The third location is by the end effector, where a force transducer is placed in series with the cable.

Another quantity that can be measured for each cable is its tension. Besides gravity, the cable tensions are the main forces used to move the end effector. As discussed in Section 3.1.4, accurate tension control is also beneficial when there are redundant cables in the CSR. The tensions can be optimized, cable slack can be avoided, and dangerously-high tensions can be prevented. One sensor that can be used for this purpose is a force transducer. These devices convert linear force into an electrical signal that can be measured using an analog-to-digital converter (ADC). These sensors come in many varieties such as a load cell, strain gage, or pressure plate. Depending on the form factor of the sensor, it can be placed in various locations of the CSR (Figure 46).

One possible location for a tension sensor is at the cable pulley. Here, the tension is indirectly calculated by measuring the pulley torque. If an electric motor drives the pulley, the torque can be estimated by monitoring the current of the motor. Otherwise, a specialized tension sensor can be attached to the spool. Another location for a cable-tension sensor is at one of the pulleys. A force transducer is attached to the pulley in order to measure the linear force produced by the cable on the pulley. To ensure accurate readings, it may be beneficial to fabricate a specialized-pulley fixture so that the direction of the cable tensions on the pulley is always the same. Care should be taken so that this fixture is rigid and does not unexpectedly affect the effective cable length. Finally, a third location for a tension sensor is at the end effector. Here, the force transducer is placed in line with the cable, so that the cable tension is measured directly. This method is the most accurate, since the true tension at the end effector is measured, but it may be more difficult to manage the sensor because it will be moving with the end effector.

As with the encoder, some performance quantities for the tension sensor should be carefully selected. For example, the sensor should be able to measure the full range of expected cable tensions, and the measurement resolution should be good enough for the intended use of the tension data. The sampling rate and frequency response of the sensor is also important to ensure that sudden cable tensions will be detected. Finally, the power-source and signal processing requirements for the tension sensor should be considered. Some sensors require special signal conditioners or power supplies to function correctly, and these requirements may constrain the placement of the sensor.

The second group of CSR sensors includes those that are used to directly measure the position of the end effector. These sensors are not critical for nominal operation of the robot but can be used to enhance performance in adverse conditions. For example, a laser system can be used to record the 3-D position of the end effector with high resolution and verify that the effector is moving as expected, even if there are errors in the cable-length measurements. Alternatively, the 3-D position can be recorded and estimated using a camera with computer vision (CV). Some disadvantages of these two sensors are that they require a clear line-of-sight to the target and some level of calibration. As such, the sensors should be placed in a fixed location where they will not be disturbed and their view will remain unobstructed. The target must also be clearly identified so that it can easily be distinguished from foreign objects. For example, special reflectors or transmitters can be fixed to the end effector. Another factor to consider when using these sensors is that their use may require costly equipment or computation power. Nonetheless, the independent measurement of the true end-effector position allows for some advanced control options that increase the CSR’s tolerance of unknown or unmodeled system kinematics or dynamics. Some control schemes that make use of an independent position-measurement sensor are presented in Section 3.2.

The third and final group of sensors are those that are used to monitor the state of the CSR configuration. The configuration consists of the number and placement of the cables. These details are important because they determine the kinematics of the robot. If there is any chance of the configuration changing, for example, due to external disturbances, it may be useful to detect the changes. In this dissertation, a portable CSR for on-site AM is envisioned. Then, one of the main tasks for setting up the CSR in a new build site is to accurately place the cable anchor points in the predetermined locations. In Figure 22, the four cables are anchored in a square pattern using tall construction towers. Ideally, these towers are placed perfectly and remain fixed. In reality, there can be placement errors or movements due to environmental disturbances. Measurement sensors can be used during the CSR installation to accurately position the cable anchor points or during the CSR operation to detect disturbances to the anchor point positions. Some examples of sensors that can be used for this purpose are lasers, radars, or cameras.

3.1.7. Actuators

The main function of the CSR is to control the position of the end effector. To this end, the robot depends on actuators that can accurately vary the cable lengths. One practical way to manage the cable lengths is to store each cable in its own spool, and then, unwind the cables as needed. Each spool is rotated by an independent servo motor. The majority of the CSRs in the literature employ this method. As an example, [55] presents the detailed design of a cable winch that incorporates the motor and spool in one package.

A servo motor is a closed-loop electromechanical system that incorporates an electric motor, encoder, and controller (Figure 47). Sometimes, a gearbox is included to increase the output torque. The servo motor is an all-in-one package that implements closed-loop control to produce controlled movements of the motor shaft. To use the servo motor, one simply provides power and a proper command signal. For example, the servo

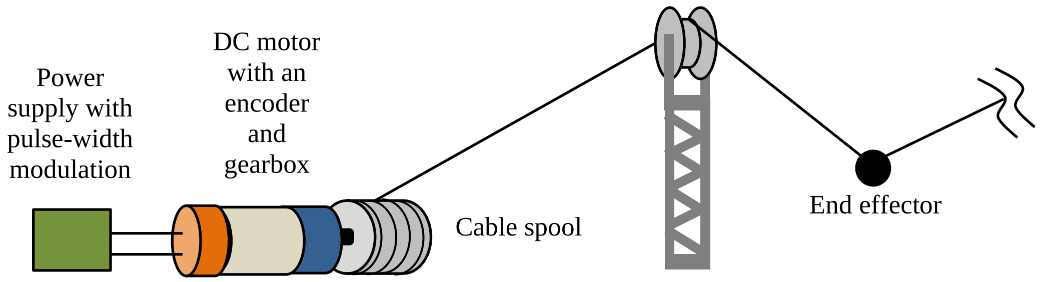

Power supply with pulse-width modulation
DC motor with an encoder and gearbox
Cable spool
End effector

Figure 47. A servo motor is used to drive the rotation of each cable spool.

motor may be programmed to rotate to whatever angular position the user selects. Or, the servo motor may be programmed to rotate at the angular velocity specified by the user. In the case of the CSR, a servo motor with position control is required. Once the desired cable length is known, the corresponding angular position of the cable spool is calculated and given to the servo motor as a reference signal. A high-quality servo motor will quickly move to the desired position with little overshoot, regardless of the cable tension. One example of such a servo motor is presented in Section 5.1.

There are several options for the type of electric motor to be used in the servo. One option is a stepper motor. This kind of motor is commonly used in small robots because it is designed to rotate in fine angular increments, or steps, that allow for fine position control, even in the absence of an encoder. However, these motors tend to be heavy, and the shaft motion can be jerky at some speeds. Any non-smooth shaft rotation due to saliency can produce undesirable cable vibrations. There is also a risk of missing steps when sudden movements are attempted. If this type of motor is used, a model with sufficient torque and stepping resolution should be selected. It would also be beneficial to include a rotary encoder in case there are any missed steps.

Another common motor type is the alternating-current (AC) motor, of which there are different variations. Although these are widely used in industry, they are best suited for operation at constant speeds. Therefore, AC motors are not a good choice for CSRs.

Another type of motor, which is the type usually found in servo applications, is the direct-current (DC) motor. As the name suggests, this type of motor is powered using a DC power source although multiple phases may be needed depending on the motor type.

Generally, the output torque is proportional to the motor armature current, and the unloaded speed of the motor shaft is proportional to the input voltage. To drive a DC motor, one may use a variable power supply or a constant power supply with pulse-width-modulation (PWM). In the PWM method, the constant voltage is switched between the on and off states at a frequency on the order of tens of kilohertz. By varying the fraction of on and off times, different voltage levels can be approximated. The PWM signal can be generated by many common microcontroller boards.

There is some other useful information about DC motors that should be noted. First, it is common for DC motors to be paired with a gearbox. The gearbox increases the output torque and decreases the rotation speed. The gear ratio of the gearbox should be selected based on the expected movement speed of the CSR. Another property of DC motors is that the torque is highest when the motor shaft is forcefully held from rotating. This condition is also called stall, and it is important because a CSR may be held at stationary positions for extended periods of time, during which the DC will continue to consume energy. If this energy expenditure is an issue, a braking system can be installed to assist in supporting stationary positions. On the other hand, the highest motor speed is achieved when there is no load torque. Therefore, cable slack can result in high-speed motor rotations if proper care is not taken. Finally, many commercial DC motors can be purchased with rotary encoders already mounted on the motor shaft.

Additional actuators may be used elsewhere in the CSR. In particular, it may be necessary to add actuators to the end effector in order to control its orientation. In an AM application, the material nozzle should remain at a proper angle to deposit material on top of previous layers. Figures 20 and 22 give some ideas of the mechanisms that may be used to stabilize the end effector. A full solution is not designed in this dissertation since the end effector is treated as a point mass. Nevertheless, the full design of the end effector will require a selection of actuators and associated hardware. It is likely that DC motors with custom gears and fixtures will be used in any design. Then, an additional consideration is how to power the actuators. One suggestion is to run power cables along with the material hose that is shown in Figure 22.

Finally, one may consider the actuators necessary to deliver the AM material to the end effector, since this is the intended use of the CSR. At minimum, a pump must be used to move the semi-fluid material through a hose that leads to the nozzle. If the material's viscosity is high, a screw-type pump can be used. If the viscosity is low, a piston may be another solution. To prevent the hose from weighing down the end effector, a support structure can guide the hose above the CSR.

3.2. Position Control

Now that the overall design and a basic understanding of the CSR properties were established, the focus turns to position control for an AM application. An immediate objective is to enable the movement of the end effector in straight and curved paths while maintaining a constant height. This is the type of movement required for depositing layers of material in AM. One may also consider what will happen if there are external disturbances to the system and whether any adverse effects can be corrected.

First, two CSR system models and disturbances are presented. Then, some control strategies based on tension and cable-length measurements are designed and simulated. Finally, a method for estimating anchor-point placement errors is described and tested.

3.2.1. Cable Models

Before discussing the system-wide models for the CSR, the cables themselves are modeled. After all, it is the properties of the cables, such as their flexibility and their inability to provide compression, that differentiate a CSR from other types of robots. A few cable models are included in this work, and they are presented in order of increasing complexity. However, all of these models are relatively simple and do not fully account for some complicated behaviors, like sagging and vibrations. Therefore, these models are best at representing light cables where the overwhelmingly-dominant force is tension.

The first cable model, which is the simplest but also the least accurate for a cable with varying length, treats the cable like a linear spring. For a linear spring, the tension is proportional to the spring deflection, and the slope of the linear relationship is called the spring constant. In this dissertation, the cable deflection is also called the cable stretch. The common-sense expectation is that even a small cable stretch produces a large tensile force, and that a cable with slack produces zero tension. To match these expectations, the spring model for a cable features a large spring constant and a piecewise discontinuity at zero stretch. This model can be summarized in the mathematical expression

$$t = \begin{cases} f(l - l_0) & \text{for } (l - l_0) > 0 \\ 0 & \text{otherwise} \end{cases}, \tag{30}$$

where t is the cable tension, l is the stretched cable length, and l_0 is the unstretched cable length. The tension is proportional to the cable stretch amount with the exception that

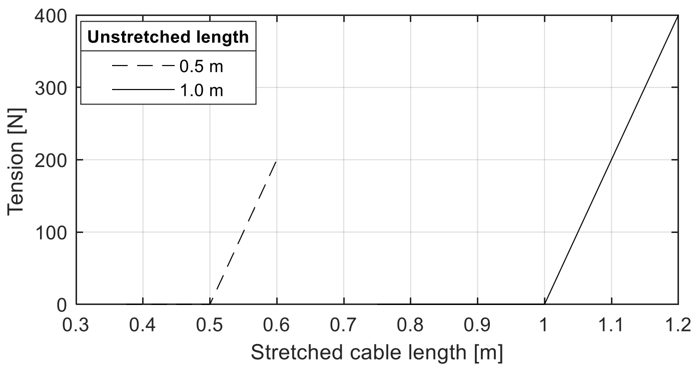


Figure 48. Some examples of the linear-spring model for the cable tension. In this model, the cable tension is proportional to the cable stretch, with the same stiffness regardless of cable length.

negative stretch, or cable slack, produces zero tension. Two examples of this cable model are plotted in Figure 48, where the spring constant of 2000 N/m was arbitrarily selected as an example. The spring constant is the slope of the line for both cable lengths.

The constant-stiffness assumption of the first model may be accurate for a small range of cable lengths, but it is certainly not true for all cable lengths. Experience dictates that a longer cable stretches more than a short cable when a tension is applied, so the model can be improved by making the stiffness dependent on the unstretched length. One way to accomplish this is to model the taut cable like a thin rod. The cable model becomes

$$t = \begin{cases} \dfrac{EA}{l_0}(l - l_0) & \text{for } (l - l_0) > 0 \\ 0 & \text{otherwise} \end{cases}, \tag{31}$$

where *E* is the elastic modulus and *A* is the cross-sectional area of the cable. The stiffness is proportional to these two variables, but it is inversely proportional to the unstretched cable length. Two examples of this model, using the same cable lengths as Figure 48 for

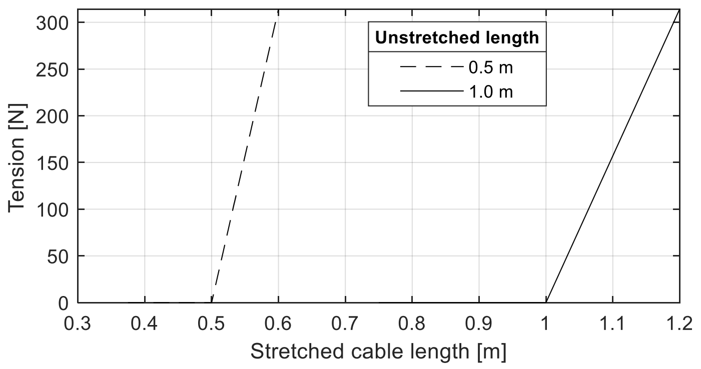


Figure 49. Some examples of the rod model for the cable tensions. In this model, the cable tension is proportional to the cable stretch but the stiffness of the cable is inversely proportional to the unstretched cable length.

comparison, are plotted in Figure 49. The elastic modulus of A36 steel, which is 200 GPa, and a circular diameter of 0.1-mm were used as example values. Note that the slope of the tension for the 1-m cable is half of that for the 0.5-m cable. As before, negative tensions are not permitted. This cable model makes more physical sense than the linear-spring model and agrees with the tensile tests of real cables.

When a cable is stretched, some of the cable tension is due to the cable weight. However, the tension contributed by the cable weight may be insignificant compared to the tension due to the cable stretch. In this dissertation, it is assumed that the cable is thin and can be treated as massless. However, the cable model may be improved by acknowledging that the cable weight can result in a small tension even when the cable has some slack. One simple way to incorporate this into the cable model is to shift the tension function. Furthermore, the shift amount can be proportional to the cable length, since a longer cable has more mass. Mathematically, this tension model can be expressed as

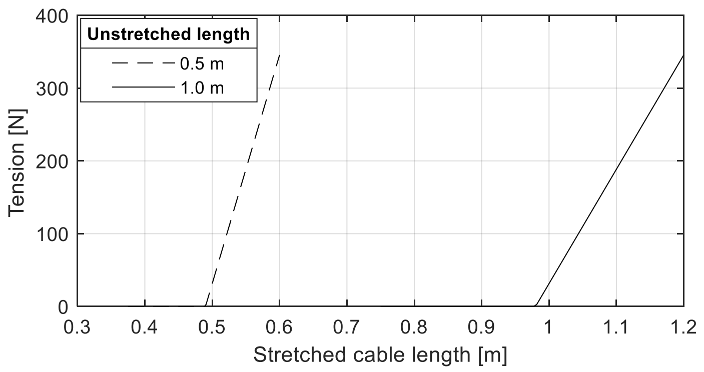


Figure 50. Some examples of the shifted-rod model for the cable tensions. This is the same model as Figure 49, but the tension functions are slightly shifted to the left, allowing for a small tensile force for small cable slacks. The amount of shift is proportional to the unstretched-cable length.

$$t = \begin{cases} \frac{EA}{l_0}(l - l_0 + sl) & \text{for } (l - l_0 + sl) > 0 \\ 0 & \text{otherwise} \end{cases}, \tag{32}$$

where s is a positive number. Some examples of this model are plotted in Figure 50. The lines in this plot are the same as in Figure 49 but are slightly shifted to the left along the horizontal axis, allowing for a small tensile force even when there is a small cable slack. The s value used in this example was 0.02, meaning that the shift to the left is 2% of the cable length. The shift amount could be adjusted using data from experimental measurements, but the exact cable tension due to the cable weight can vary based on other factors, such as the orientation of the cable. Thus, this model is a simple approximation.

There is a significant amount of work in the literature related to the modeling and practical use of cables. In this dissertation, the simple model of Figure 49 is used since it makes physical sense and since the cable lengths used in simulation and experiments are relatively short, such that cable weight is small. Then, any slack results in negligible tension. Cable vibrations due are also ignored. However, the issue of cable vibrations is a significant one that deserves further investigation or experimentation, especially for a large-scale CSR, since vibrations lead to varying cable tensions.

3.2.2. System Models

The CSR system can essentially be described as an interaction between the end effector and the cable tensions. In this section, two variations of the CSR-system model are presented as mathematical equations. The main difference between the two models is that one assumes direct knowledge of the cable tensions while the other only assumes knowledge of the cable lengths. Internally, the second model contains and the cable-tension equation so that the cable tensions can be calculated. The first model does not include such equation. The second model is ultimately preferred in this dissertation because it paves the way for some servo-based control strategies.

The CSR dynamic equations are derived using the Newtonian method in the Cartesian coordinate system. The end effector is assumed to be a point mass with mass *m*, and the forces acting on it are gravity, the cable tensions, and a small amount of damping. The position of the mass is defined in (1), and the Cartesian coordinates are defined in Figure 29. Then, the acceleration of the end effector is related to the forces by

$$\begin{aligned} m\ddot{x} &= F_x - c\dot{x} \\ m\ddot{y} &= F_y - c\dot{y} \\ m\ddot{z} &= F_z - c\dot{z} - mg, \end{aligned} \tag{33}$$

where F_x, F_y, and F_z are the summation of cable tensions along the *x*, *y*, and *z* directions, respectively. These forces are defined in (8−9) and come from the individual cable tensions as well as the position of the end effector relative to the cable anchor points. The *mg* term is the weight of the end effector. The damping coefficient, *c*, is a lumped parameter that is included for some energy dissipation in the system. In reality, the energy dissipation can come from many sources, like air drag or heating of the cables as they flex. Whatever their source is, energy-dissipation forces can be modeled as damping forces that are resolved into the three Cartesian directions, and their magnitudes generally increase with faster end-effector movements. Therefore, as long as the damping forces are much smaller than the cable tensions, the lumped-parameter damping coefficient assumption is not a critical choice. However, the value of *c* could be debated or based on experimentation. If the movement speed of the CSR is slow, then these damping forces will also be small, regardless of the exact choice of *c*.

The cable-force expressions from (9) are now substituted into (33) to give

$$\begin{aligned} m\ddot{x} &= \sum_i \frac{x_i - x}{l_i} t_i - c\dot{x} \\ m\ddot{y} &= \sum_i \frac{y_i - y}{l_i} t_i - c\dot{y} \\ m\ddot{z} &= \sum_i \frac{z_i - z}{l_i} t_i - c\dot{z} - mg. \end{aligned} \tag{34}$$

Recall that t_i is the tension of the i^{th} cable, l_i is its length, and (x_i, y_i, z_i) are the coordinates of its anchor point. The three equations are coupled and nonlinear, since the l_i term expands to (3).

The first system model assumes that the cable tensions, t_i, are the inputs. This implies that the tensions are directly controlled. Then, (34) is the entire model. The same equations can be restated in the matrix form

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix} \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} + \begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} = \boldsymbol{AT}, \tag{35}$$

where ***A*** is a cable-direction matrix, like those in (10−11), and ***T*** is a column vector of the cable tensions. In this form, the simplicity of the model is clearly seen. The left side of (35) is three decoupled equations of motion and the right side are the input forces for the three equations. The matrix ***A***, which is a function of the mass position with respect to the cable anchor points, is solely responsible for coupling the input cable tensions. Section 3.1.4 discussed exactly how ***T*** can be found to produce the desired Cartesian forces. The steps involve finding an inverse of ***A***. Once a suitable inverse is found, the right side of (35) is effectively replaced by the desired Cartesian forces and control of the system is trivial because it is equivalent to three decoupled 1-DOF systems. Some examples of this tension-based control are provided in Section 3.2.4.

It may be difficult or impractical to control the cable tensions directly. This is especially true if the cables are very stiff. For this reason, another model, which does not assume direct tension control, is developed. To do this, the cable tensions in (34) are replaced by the cable-tension model from (31). The result is

$$\begin{aligned} m\ddot{x} &= \sum_i \frac{x_i - x}{l_i}\frac{EA}{l_i}(l_i - l_{i0}) - c\dot{x} \\ m\ddot{y} &= \sum_i \frac{y_i - y}{l_i}\frac{EA}{l_i}(l_i - l_{i0}) - c\dot{y} \\ m\ddot{z} &= \sum_i \frac{z_i - z}{l_i}\frac{EA}{l_i}(l_i - l_{i0}) - c\dot{z} - mg. \end{aligned} \tag{36}$$

Here, the discontinuity in (31) is not explicitly stated, but it still applies. After some rearranging, the equations can be rewritten as

$$
\begin{aligned}
m\ddot{x} &= EA\sum_i \frac{x_i - x}{l_i} - \frac{x_i - x}{l_i^2} l_{i0} - c\dot{x} \\
m\ddot{y} &= EA\sum_i \frac{y_i - y}{l_i} - \frac{y_i - y}{l_i^2} l_{i0} - c\dot{y} \\
m\ddot{z} &= EA\sum_i \frac{z_i - z}{l_i} - \frac{z_i - z}{l_i^2} l_{i0} - c\dot{z} - mg.
\end{aligned}
\tag{37}
$$

Finally, in matrix form, the model is

$$
\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix}
\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix}
+
\begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & c \end{bmatrix}
\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}
+
\begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}
- EA\boldsymbol{A}\boldsymbol{1} = -EA\boldsymbol{H}\boldsymbol{L},
\tag{38}
$$

where ***1*** is a column vector of ones with the same number of columns as ***A***, ***H*** is a matrix like ***A***, but with l_i^2 in the denominators instead of l_i, and ***L*** is a vector of unstretched cable lengths. Some examples of the ***A*** matrix were provided in (10−11).

In this second model, the inputs are the unstretched cable lengths. Any positive cable stretch creates a tension that acts on the mass. Thus, the cable tensions are indirectly controlled by controlling the cable lengths. Section 3.2.5 presents some control laws based on varying the cable lengths.

One final improvement to the model is to include the actuator dynamics. After all, the cable lengths are varied using actuators that have some associated response characteristics. Section 3.1.7 describes, for example, how the CSR cables can be wound in spools using servo motors. The servo motors are expected to quickly follow commands, but their response cannot be instantaneous. The dynamic behavior of the motor can be added to the model by adding yet another layer to the model inputs. That is, the vector of unstretched cable lengths in (38) is not directly controlled but is actually the output of another set of differential equations. For instance, a well-tuned actuator can accurately be modeled using a second-order differential equation with a select natural frequency and damping factor. Then, each of the cable lengths in the vector ***L*** can be considered as the output from such a second-order system. The input to these second-order systems becomes the new inputs for the overall CSR model. An example of this kind of modeling is presented in Section 3.2.5.1.

3.2.3. Disturbances

Another element in the CSR-system model that merits consideration is the disturbances. In general, these are any external forces or changes to the system that disrupt the ideal operation. Their origin could be human error or environmental factors. In other words, the disturbances are due to systematic or random errors. In any case, it is useful to identify and accurately model the errors so they can be simulated and dealt with. A number of possible system errors are now presented.

The most general external disturbance is one that acts directly on the end effector itself. Some examples are forces due to wind, weight from a payload, or reactions from contact with other objects. In the case of AM, the deposition of material can cause a reaction force upward on the nozzle and, also, a force that opposes the nozzle motion. Cable vibrations will also produce a force on the end effector. These types of disturbances could be modeled as one or more functions added to the *x*-*y*-*z* Newtonian equations (33). A reaction force from the steady deposition of material is modeled as steady force in the positive-*z* direction, and possibly additional damping forces in the *x*-*y* directions. A wind force is modeled as a sinusoidal force with multiple frequency components that act in the direction of the wind. Random noise components can also be added to the force magnitude.

Forces from the cables are modeled either as Cartesian-direction forces or as forces along the cable directions. In any case, the forces can be modeled as step inputs, periodic functions, or noisy signals. There are many decisions to be made about the magnitude of the forces as well as the period of oscillation. In general, the force magnitudes should be less than the cable tensions if there is any hope of maintaining good operation. A good operation is observed when the end-effector position stays within a small error bound in the *x*-*y*-*z* directions.

Another type of disturbance is one that changes the robot configuration and, consequently, the kinematics. One example is a change in the cable anchor points. This can be a static error, like the incorrect placement of the cables, or a dynamic error, like the cable anchor points swaying during the operation. These types of errors are modeled by varying the system parameters in (33). The parameters can vary randomly, by a constant amount, or periodically. A novel technique for estimating unknown anchor-point position errors of the constant type is presented in Section 3.2.6.

Other model parameters, related to the kinematics, can also be varied and their effect studied. For example, the end effector mass can be varied to ensure that the system performance is satisfactory for various payload weights. The linear-damping coefficient is varied to test different amounts of overall damping on the movement of the mass. The cable modulus of elasticity or cable-tension function are varied to test the effect of different cable stiffness values.

There is also another group of errors that are specifically related to the system sensors. These types of errors are important because the feedback control is affected by the measurement inaccuracies. The cable-length measurements are one example. From the start of operation, the cable lengths should be known. In practice, this can be accomplished by bringing the CSR to some initial starting position where the cable lengths are known (a step called "homing"). Then, the cable lengths are continuously updated by measuring any length changes over time. Any errors in the initial cable lengths or in the measurements of the length changes will remain in the system memory. This kind of error is accounted for in the model by manually changing the cable lengths at some time step in the simulation. Therefore, a CSR model that accounts for the cable lengths, such as (36) must be used. A model based on direct tension control, such as (34), does not provide information about the cable lengths. Another quantity that may or may not be used for operation is the direct measurements of the of the end-effector position. This measurement can come from an external sensor, like a camera. Errors in the camera images will have an effect on the performance if the measurements are used for closed-loop control. For example, if the camera is not calibrated correctly, there could be a systematic error in the *x*-*y*-*z* position of the mass. The mathematical equations used in modeling the errors are based on some physical reasoning or experimental results.

When sensor errors are modeled, it is important to realize that a measurement error does not affect the true value of the quantity being measured. For instance, an error in the measurement of the mass *x*-*y*-*z* position does not mean that the true *x*-*y*-*z* position is changed. Therefore, this kind of measurement error is modeled by creating a copy of the true mass position and adding the error to the copy. Then, any closed-loop control that depends on the sensor measurement utilizes the variable copy for calculations, and the true mass position is not affected. The only way for the true mass position to change is to interact with the mass through the cable tensions.

Various disturbance and error types are modeled and simulated in the forthcoming sections. Some of the errors are recreated in the CSR experiment of Chapter 5. Whenever possible, the simulations are compared to the experimental results.

3.2.4. Tension-Based Control

Conceptually, one of the simplest forms of control of the CSR is tension-based control. If one can accurately control the cable tensions an arbitrary force vector can be applied to the end effector, within the limits already discussed. Then, control is a matter of calculating the force vector necessary to produce different movements.

The tension-based control begins with the model in (35) where the control inputs are the individual cable tensions. This model is applied to the four-cable CSR depicted in Figure 51, where the side lengths are 10 m and the height is 10 m. The point mass is 10 kg and the damping is 10 N·s/m. These values and dimensions were arbitrarily selected as an example. For comparison, the SkyCam system from Figure 19 has a mass of about 13 kg and is suspended over stadiums with lengths exceeding 100 m [56]. The test trajectory is a circular helix with a radius of 3 m, centered at $x = y = 5$ m. The trajectory starts at the position (8, 5, 2) m and goes counter-clockwise with an upward velocity of 1/15 m/s. The trajectory ends at 60 s when the point mass has completed two circular rotations and reaches the finishing position of (8, 5, 6) m. The following subsections use this trajectory to test some tension-based control laws via simulation. For this task, it is useful to re-state the robot model from (35) as

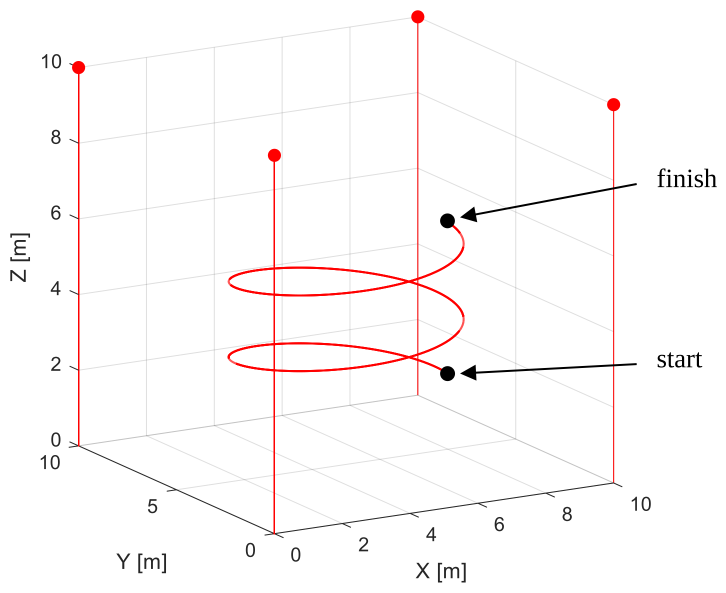


Figure 51. A four-cable CSR configuration and a helical reference path. The cable anchor points, but not the cables themselves, are shown. This is the setup used to test the tension-based control.

$$\boldsymbol{M}\ddot{\boldsymbol{q}}+\boldsymbol{C}\dot{\boldsymbol{q}}+\boldsymbol{g}=\boldsymbol{A}\boldsymbol{T}, \tag{39}$$

where $\boldsymbol{M}$, $\boldsymbol{C}$, and $\boldsymbol{g}$ represent the system matrices and $\boldsymbol{q}$ is a vector of the end-effector Cartesian coordinates. This form of the model is sometimes called the "standard form" of a robotic system. Furthermore, the error state is defined as

$$\boldsymbol{e}=\boldsymbol{q}_r-\boldsymbol{q}, \tag{40}$$

where $\boldsymbol{q}_r$ is the coordinates from the reference trajectory. Then, the goal of control is to bring the error to zero, or near zero, and keep it there.

3.2.4.1. Feedback Linearization

One of the common control methods for robotic systems is feedback linearization. In this method, an inverse of the robot model is included in the control law to effectively cancel the robot dynamics. At the same time, a new linear dynamics is imposed on the system to control the robot state and achieve a desired performance. Mathematically, one example of a feedback control law is

$$\boldsymbol{A}\boldsymbol{T} = \boldsymbol{M}(\ddot{\boldsymbol{q}}_r + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}) + \boldsymbol{C}\dot{\boldsymbol{q}} + \boldsymbol{g}, \tag{41}$$

where k_d and k_p are constant parameters. Substituting this control law into (39) gives

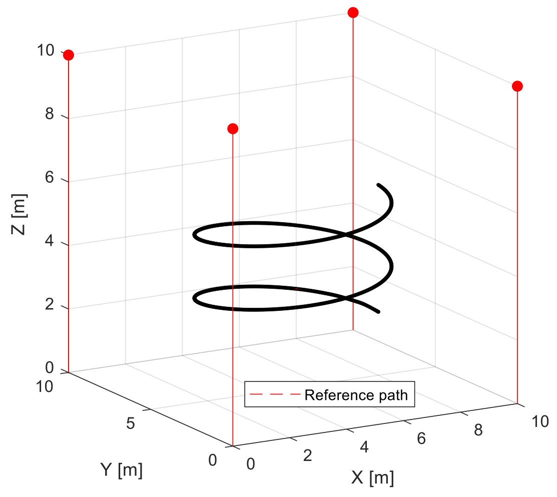
$$\begin{aligned} \boldsymbol{M}\ddot{\boldsymbol{q}} + \boldsymbol{C}\dot{\boldsymbol{q}} + \boldsymbol{g} &= \boldsymbol{M}(\ddot{\boldsymbol{q}}_r + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}) + \boldsymbol{C}\dot{\boldsymbol{q}} + \boldsymbol{g} \\ 0 &= \boldsymbol{M}(\ddot{\boldsymbol{q}}_r - \ddot{\boldsymbol{q}}) + \boldsymbol{M}k_d\dot{\boldsymbol{e}} + \boldsymbol{M}k_p\boldsymbol{e} \\ 0 &= \ddot{\boldsymbol{e}} + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}. \end{aligned} \tag{42}$$

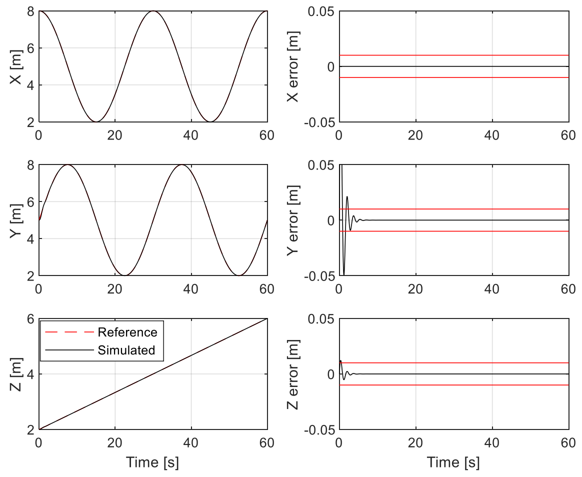
Therefore, the error state will exhibit an unforced, damped, second-order response that is shaped by the values of k_d and k_p. The error will go to zero, over time.

To implement (41) it will be necessary to solve for the cable tensions, $\boldsymbol{T}$. The final expression for cable tensions is

$$\boldsymbol{T} = \boldsymbol{A}^{-1}(\boldsymbol{M}(\ddot{\boldsymbol{q}}_r + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}) + \boldsymbol{C}\dot{\boldsymbol{q}} + \boldsymbol{g}), \tag{43}$$

where the inverse of $\boldsymbol{A}$ was used to solve for $\boldsymbol{T}$. As discussed in Section 3.1.4, $\boldsymbol{A}$ may not have an exact inverse, so a pseudo inverse is used instead. Various pseudo inverse methods are tested in Section 3.2.4.3.



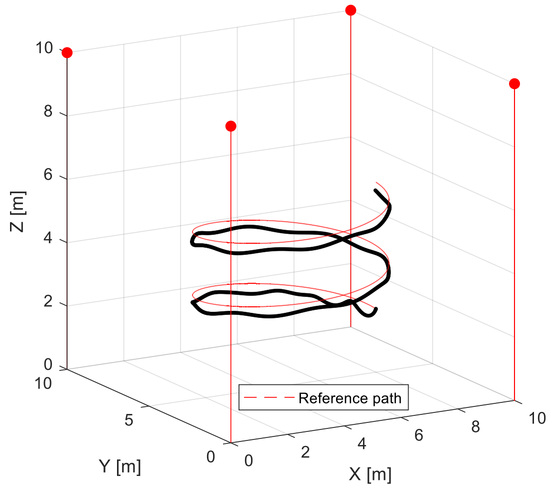


(a) (b)

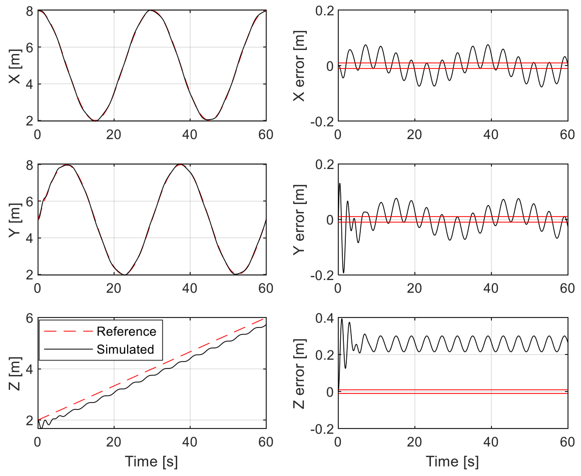
Figure 52. (a) 3-D plot of the simulation results using a feedback-linearization controller. (b) The Cartesian position and position error for the end effector. The horizontal lines mark the ±1-cm error bounds.

Figure 52 shows the simulated response of the CSR using the feedback-linearization controller. The CSR follows the reference path almost perfectly, such that the reference path is not visible in the plots. There is some initial error in the *y* and *z* directions because the mass starts from rest, but the position quickly converges to the trajectory. The k_p and k_d values used were 15 and 2, respectively, which produced the lightly-damped response in the error plots. The error plots include a ±1-cm error bound that the error remains well within after a few seconds.

The same simulation was repeated and plotted in Figure 53. This time, model uncertainties and an external disturbance were added. The mass was increased by 4 kg and the damping decreased by 8 N·s/m, with respect to the nominal values. In addition, an external force of 5 N was applied directly to the mass. The force is sinusoidal with 0.25-



(a)



(b)

Figure 53. (a) 3-D plot of the simulation results using a feedback-linearization controller, including some model errors and an external force disturbance. (b) The Cartesian position and position error for the end effector. The horizontal lines mark the ±1-cm error bounds.

Hz frequency, and it acts equally in the *x*, *y*, and *z* directions. The model uncertainties and external disturbance have a clear effect on the response. The model parameters cause a steady offset in the *z* error and a low-frequency oscillation in the *x*-*y* error, centered around zero. The error in the mass means that the end effector is heavier than expected, so the *z* forces are not large enough to eliminate the *z* error. The error in the damping term means that there is much less damping than expected, which adds to the low-frequency oscillations. The external force disturbance causes a movement in the mass with the same frequency as the disturbance. This additional movement is about the same magnitude in all three directions. The total result is that the Cartesian errors are not within the ±1-cm error bounds.

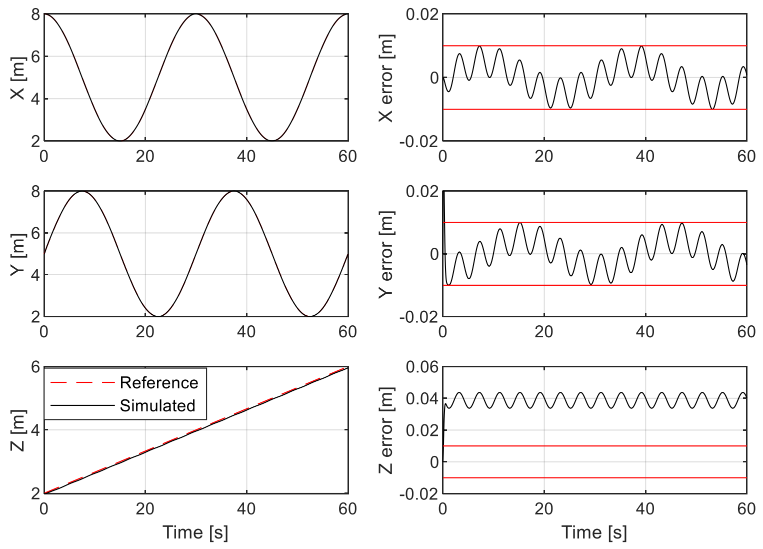


Figure 54. Simulation results using a feedback-linearization, including some model errors and an external force disturbance. This time, the values for k_p and k_d are much higher than in Figure 53. While the *x*- and *y*-direction errors are smaller, the *z* error remains large.

Figure 54 shows one final simulation of the system using the same controller. The same model uncertainties and external-force disturbance as in the simulation of Figure 53 were used. However, the k_p and k_d values were increased substantially in hopes of reducing the position errors. The new values, arbitrarily chosed, were $k_p = 100$ and $k_d = 20$. Despite showing similar effects from the model uncertainties and external disturbance, such as noticeable oscillations, the *x* and *y* errors are reduced to within the error bounds. However, the *z* error remains large and outside of the error bound. This suggests that the feedback linearization approach, alone, is not sufficient for robust control of the system. This is especially true if the model parameters are not exactly known or are likely to vary, since the control law (41) relies on an accurate system model. In an AM application, the effective mass of the end effector can vary due to the flow of material. The damping could also change due to environmental conditions or the cable material properties. Increasing the controller gains is somewhat helpful in reducing errors, but only to a certain extent.

3.2.4.2. Sliding-Mode Control

To increase the robustness to model uncertainties, a sliding-mode control law can be used. The starting point for this control law is the same feedback controller in (41). An additional, switching function is added to the controller. The new control law is

$$\boldsymbol{A}\boldsymbol{T} = \widehat{\boldsymbol{M}}(\ddot{\boldsymbol{q}}_r + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}) + \widehat{\boldsymbol{C}}\dot{\boldsymbol{q}} + \hat{\boldsymbol{g}} + \boldsymbol{K}\,\mathrm{sign}(\boldsymbol{s}), \tag{44}$$

where the hat accents indicate that the corresponding matrices are only estimates of the true values. The ***K*** is a diagonal gain matrix with the values selected in the forthcoming derivation. The ***s*** is a new variable of a sliding surface. The sliding surface is defined as

$$\boldsymbol{s} = \dot{\boldsymbol{e}} + k_d\boldsymbol{e} + k_p\int \boldsymbol{e}\,dt\,, \tag{45}$$

and its time derivative is

$$\dot{\boldsymbol{s}} = \ddot{\boldsymbol{e}} + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}. \tag{46}$$

When ***s*** is zero, and remains there, $\dot{\boldsymbol{s}}$ is zero and the tracking error is confined to the same dynamics as given by the feedback-linearization controller in (42). This is why (44) has the same form as (41). The new sign term is added to ensure that the system will move towards $\boldsymbol{s} = \boldsymbol{0}$ even if there are some model uncertainties. The values of the gain matrix ***K*** are selected using the Lyapunov stability approach and including some knowledge of the model uncertainties.

Let the candidate Lyapunov function be

$$V = \frac{1}{2}\boldsymbol{s}^T\boldsymbol{M}\boldsymbol{s}, \tag{47}$$

which is positive definite. Then, the time derivative of the Lyapunov function is

$$\dot{V} = \frac{1}{2}\dot{\boldsymbol{s}}^T\boldsymbol{M}\boldsymbol{s} + \frac{1}{2}\boldsymbol{s}^T\boldsymbol{M}\dot{\boldsymbol{s}} = \boldsymbol{s}^T\boldsymbol{M}\dot{\boldsymbol{s}}. \tag{48}$$

Substituting (46) into this expression gives

$$\begin{aligned}\dot{V} &= \boldsymbol{s}^T\boldsymbol{M}\left(\ddot{\boldsymbol{e}} + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}\right) \\ &= \boldsymbol{s}^T\left(\boldsymbol{M}\ddot{\boldsymbol{q}}_r - \boldsymbol{M}\ddot{\boldsymbol{q}} + \boldsymbol{M}k_d\dot{\boldsymbol{e}} + \boldsymbol{M}k_p\boldsymbol{e}\right).\end{aligned} \tag{49}$$

Now, substituting the CSR dynamics for the $\boldsymbol{M}\ddot{\boldsymbol{q}}$ term gives

$$\dot{V} = \boldsymbol{s}^T\left(\boldsymbol{M}\ddot{\boldsymbol{q}}_r - \boldsymbol{A}\boldsymbol{T} + \boldsymbol{C}\dot{\boldsymbol{q}} + \boldsymbol{g} + \boldsymbol{M}k_d\dot{\boldsymbol{e}} + \boldsymbol{M}k_p\boldsymbol{e}\right). \tag{50}$$

Substituting the control law, (44), gives

$$\dot{V} = \boldsymbol{s}^T\left(\boldsymbol{M}\ddot{\boldsymbol{q}}_{\boldsymbol{r}} - (\widehat{\boldsymbol{M}}(\ddot{\boldsymbol{q}}_r + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}) + \widehat{\boldsymbol{C}}\dot{\boldsymbol{q}} + \hat{\boldsymbol{g}} + \boldsymbol{K}\,\mathrm{sign}(\boldsymbol{s})) + \boldsymbol{C}\dot{\boldsymbol{q}} + \boldsymbol{g} + \boldsymbol{M}k_d\dot{\boldsymbol{e}} + \boldsymbol{M}k_p\boldsymbol{e}\right). \tag{51}$$

After some algebra, this equation simplifies to

$$\dot{V} = \boldsymbol{s}^T\left(-\left(\widehat{\boldsymbol{M}} - \boldsymbol{M}\right)\left(\ddot{\boldsymbol{q}}_{\boldsymbol{r}} + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}\right) - \left(\widehat{\boldsymbol{C}} - \boldsymbol{C}\right)\dot{\boldsymbol{q}} - (\hat{\boldsymbol{g}} - \boldsymbol{g}) - \boldsymbol{K}\,\mathrm{sign}(\boldsymbol{s})\right). \tag{52}$$

For stability, this expression must evaluate to a negative number for every $\boldsymbol{s} \neq \boldsymbol{0}$. Rewriting $\dot{V}$ as the sum

$$\dot{V} = \sum_i \dot{V}_i = \sum_i s_i\left(-(\widehat{m_i} - m_i)\left(\ddot{\boldsymbol{q}}_r + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}\right)_i - (\widehat{c_i} - c_i)\dot{q}_i - \left(\widehat{g}_i - g_i\right) - k_i\,\mathrm{sign}(s_i)\right) \tag{53}$$

it is clear that $\dot{V}$ will be negative if $\dot{V}_i$ is negative for every i. Suppose that the maximum error for the values in the $\boldsymbol{M}$, $\boldsymbol{C}$, and $\boldsymbol{g}$ matrices are $\overline{m}$, $\overline{c}$, and $\overline{g}$, respectively. Then, if s_i is positive, the maximum possible value of $\dot{V}_i$ is

$$|s_i|\left(\overline{m}\left|\ddot{\boldsymbol{q}}_r + k_d\dot{\boldsymbol{e}} + k_p\boldsymbol{e}\right|_i + \overline{c}\dot{q}_i + \overline{g} - k_i\right). \tag{54}$$

On the other hand, if s_i is negative, the maximum possible value of $\dot{V}_i$ is

$$-|s_i|\left(-\overline{m}\left|\ddot{\boldsymbol{q}}_r+k_d\dot{\boldsymbol{e}}+k_p\boldsymbol{e}\right|_i-\overline{c}\dot{q}_i-\overline{g}+k_i\right). \tag{55}$$

In both cases, $\dot{V}_i$ is guaranteed to be negative as long as

$$k_i>\overline{m}\left|\ddot{\boldsymbol{q}}_r+k_d\dot{\boldsymbol{e}}+k_p\boldsymbol{e}\right|_i+\overline{c}\dot{q}_i+\overline{g}. \tag{56}$$

Then, let k_i be selected as

$$k_i=\overline{m}\left|\ddot{\boldsymbol{q}}_r+k_d\dot{\boldsymbol{e}}+k_p\boldsymbol{e}\right|_i+\overline{c}\dot{q}_i+\overline{g}+k_0, \tag{57}$$

where k_0 is some positive number. This will guarantee that $\dot{V}$ will always be negative, which means that the system will converge to $\boldsymbol{s}=\mathbf{0}$.

Figure 55 demonstrates the effectiveness of the sliding-mode controller in providing robust control. For comparison, the same conditions as in Figure 53 were used.

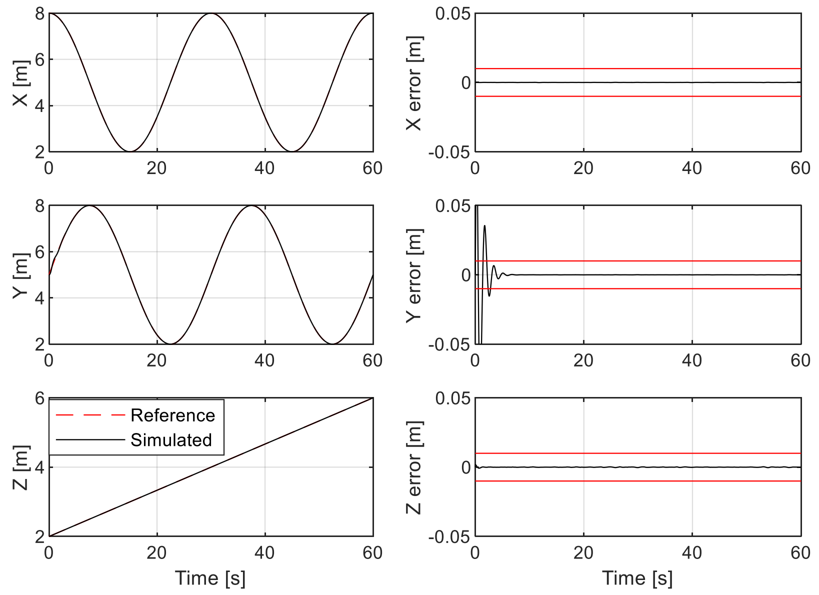


Figure 55. Simulation results using the sliding-mode controller, with model errors and an external force disturbance. The errors remain well within the 1-cm position-error bounds.

That is, model uncertainties and an external force disturbance were included. The control law (44) was implemented with the same k_p and k_d as used in Figure 53, and the $\boldsymbol{K}$ was calculated according to (57), using $k_0 = 5$. Now, the position errors converge to zero and remain within the error bounds.

Even the effects of the external force disturbance are rejected by this new controller, as evidenced by the lack of oscillations in Figure 55. To further explore this result, the simulation was repeated with twice the magnitude of the disturbance. That is, a 10-N sinusoidal force was imposed. With this disturbance, the system response is as shown in Figure 56. Now, the effects of the disturbance are more pronounced in the *x* and *y* directions. Still, the errors remain nearly within the error bounds. As in the feedback-

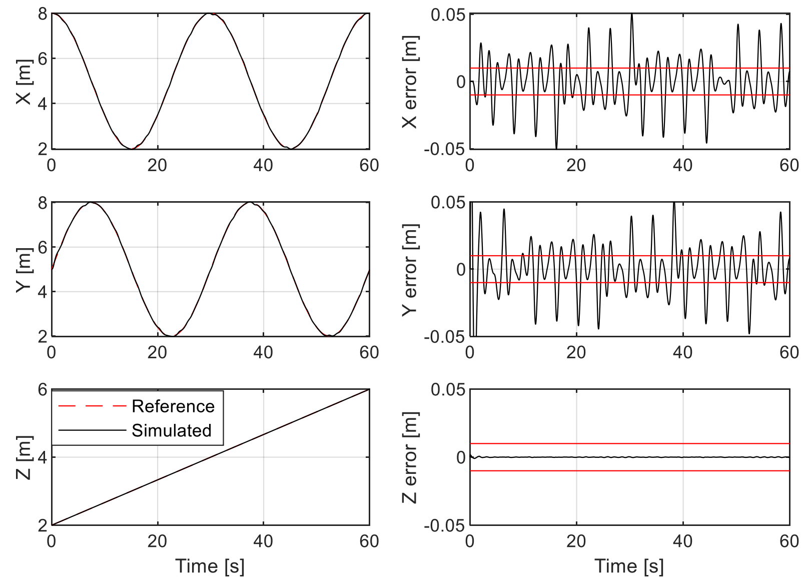


Figure 56. Simulation results using the sliding-mode controller, with model errors and an external force disturbance. The magnitude of the external disturbance was increased to 10-N, which is twice the magnitude used in Figure 55.

linearization case, the errors can be reduced to by increasing the k_p and k_d controller gains. Interestingly, the *z*-direction error remains small regardless of the increased disturbance.

The results in this section support the assertion that that the sliding-mode controller is the better option for tension-based control of the system, especially when there are model uncertainties and disturbances acting directly on the end effector.

3.2.4.3. Tension Optimization

In feedback-linearization and sliding-mode control, the control laws effectively calculate the net force required to move the end effector in the desired way. However, in a CSR with more than three cables the same net force can be created using different cable tensions, owing to the fact that the system is over-constrained with respect to 3-D positioning. This gives an opportunity for the cable tensions to be optimized to achieve a certain objective. Section 3.1.4 presented some possibilities and rational for different objectives. These options are now tested in simulation.

Once again, the test system is the one presented in Figure 51, and the end effector is tasked to follow a helical path. Suppose the feedback-linearization control law is used, and the CSR system exhibits the behavior shown in Figure 52. The Cartesian forces that produce this behavior are shown in Figure 57. As expected, the *z*-direction force remains near the weight of the end effector, and the *x*-*y*-direction forces vary periodically, with the same amplitude, with a zero mean. These behaviors are due to the circular trajectory and the slow upward movement.

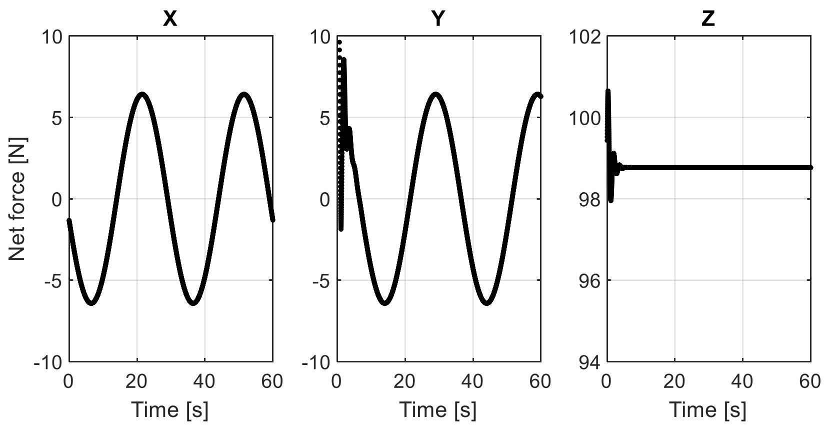


Figure 57. Cartesian forces for the simulation plotted in Figure 52.

Now, some different optimization methods to calculate the cable tensions are tested. All of these methods produce the same net forces shown in Figure 57.

The first method is (15), which is the least-squares method with a nonnegative tension constraint. This method produces the cable tensions shown in Figure 58. In this figure, the cables are labeled *a* through *c*. Cable *a* corresponds to the cable that originates from the anchor point above the origin. The remaining letters are assigned to the anchor points in the counter-clockwise order, when the CSR is viewed from above. The cable tensions are always positive, which is favorable because this prevents cable slack. The tensions curves are also relatively smooth throughout the movement. This is important because any sudden changes in cable tensions could produce unwanted cable vibrations or swinging of the mass. For these reasons, this is one of the preferred methods for tension optimization.

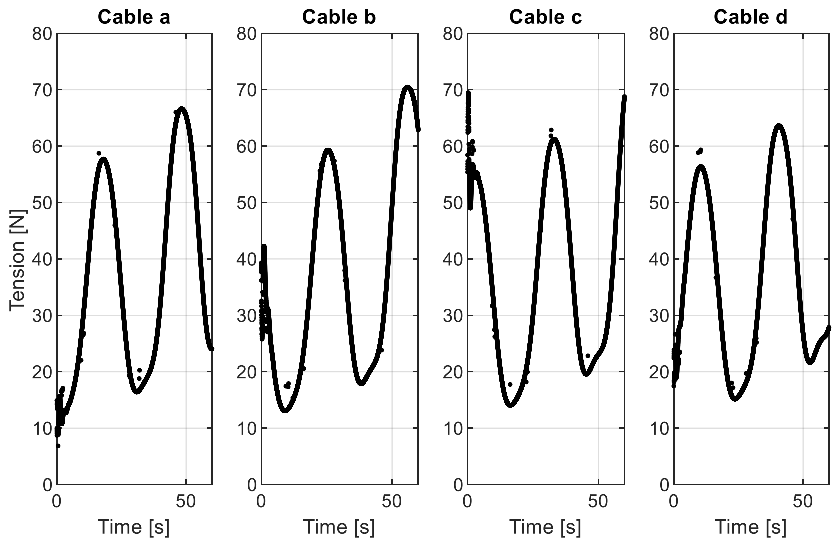


Figure 58. Cable tensions required to produce the net forces in Figure 57, as calculated using the least squares optimization method (15).

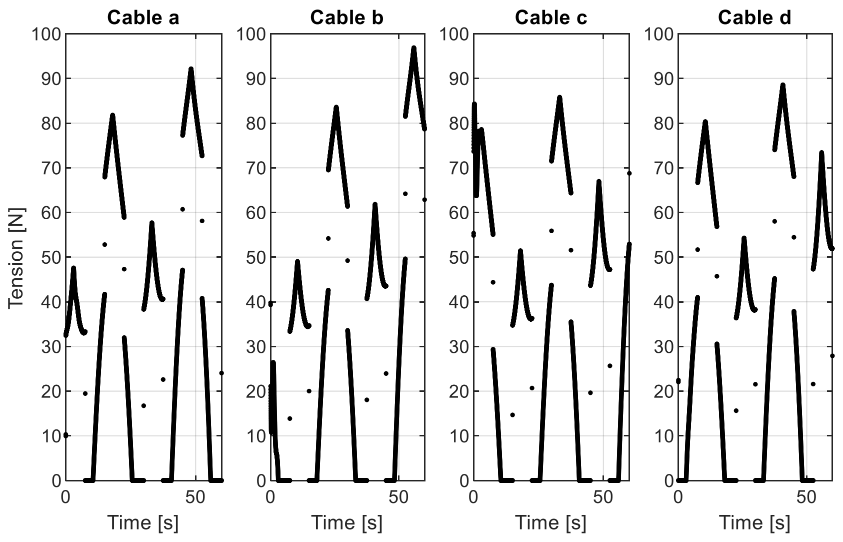


Figure 59. Cable tensions required to produce the net forces in Figure 57, as calculated using the minimum-tension sum optimization method (16).

The second tension-optimization method, (16), aims to minimize the sum of cable tensions. This method produces the cable tensions shown in Figure 59. Immediately, one can see that there are many discontinuities in the curves. For some intervals, the cable tensions go to zero or suddenly spike. Therefore, these cable tensions are not practical because they would be difficult to create in a real system. Also, the cables with zero tension could easily transition to a slack state that is undesired. Finally, the cable tensions reach substantially higher values than those in Figure 58. For these reasons, this optimization method is not preferred.

The third optimization method, (17), minimizes the sum of squared tensions. The resulting cable tensions are shown in Figure 60. These results are nearly identical to those in Figure 58. In fact, it is known that the least-squares method produces the solution with the smallest 2-norm of the cable tensions. This norm is the square root of the objective function used in (17). Thus, the plotted solution is the same. The difference in the two

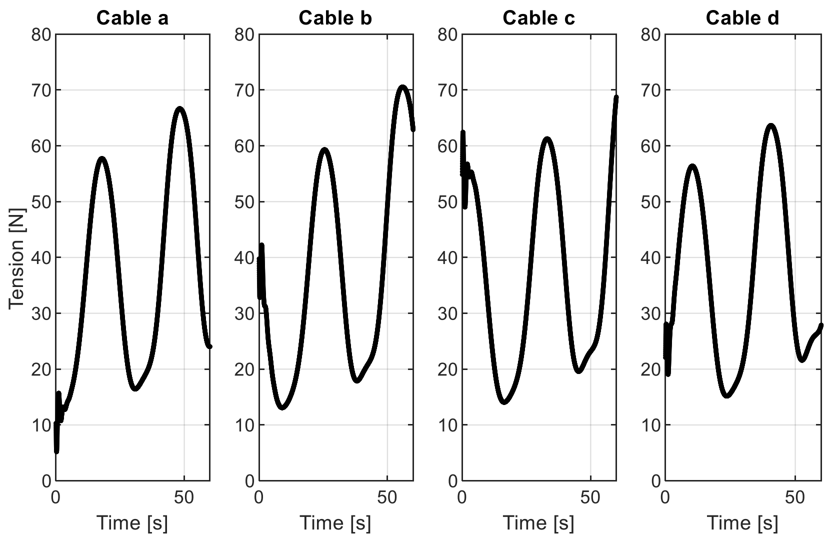


Figure 60. Cable tensions required to produce the net forces in Figure 57, as calculated using the minimum-squared-tension sum optimization method (17).

optimization methods is in how the problem is posed. The least-squares method does not assume that there is a solution to the force-tension equation (11), but the present optimization problem has the equation as a constraint. In this case, the results are identical, save for some occasional noise-like behaviors in Figure 58. The slight differences may be due to the algorithm or tolerance differences of the numerical solvers used to perform the optimizations. A relevant note is that the least-squares optimization method took about 7 times more time to solve in MATLAB than the present method. In this sense, (17) is the most preferred method.

The three optimization methods can also be compared according to their objective-function values. This comparison is done in Table 1, where two different metrics are calculated for each method. The metrics are calculated from the plotted results in this section, and their values represent the objective-functions being minimized in the optimization processes. The first metric is a time integral of the cable tensions. This is a measure of the tension sums over time. The second metric is a time integral of the squared tensions. This is a measure of the squared-tension sums over time. As expected, the second method produces the smallest tension integral among the three methods though it is not much smaller. The third method produces the smallest squared-tension integral. The first method gives nearly the same metrics as the third, supporting the explanation that the two

Table 1. Comparison of metrics for the three tension-optimization methods.

Optimization method	**Time integral of tensions**	**Time integral of squared tensions**
Least squares with nonnegative constraint, (15)	8796.88	392,591.33
Minimize sum of tensions, (16)	8741.86	529,997.92
Minimize sum of squared tensions, (17)	8796.79	392,481.62

methods give essentially the same solution. Based on these results, alone, either the first or third methods are preferred, and the second method does not provide a useful advantage. Recall that the electrical power required to maintain a tension is proportional to the squared tension. In this sense, the largest advantage is provided by the third method.

3.2.5. Cable-Length Control

Another form of control that does not depend on direct knowledge of the cable tensions is cable-length control. In a generic robotic system, this is equivalent to servo control of the joint variables. Since the robot kinematics is known, position control of the end effector is reduced to the control of the individual cable lengths. Specifically, (3) is used to calculate the required cable lengths for a given robot position. Any motion trajectory can be discretized into a sequence of positions.

The cable-length control is accomplished in two steps. First, a servo loop is designed. This loop measures one cable length and commands the associated actuator to change the length to the desired value. The same loop is applied, separately, to every cable in the CSR. Second, a kinematics loop is designed. This loop converts the desired robot trajectory into the required cable lengths. This calculation can include data from another sensor to correct for position errors in real-time. The kinematics and servo loops are arranged in a dual-loop structure in Section 3.2.5.2.

3.2.5.1. Servo Control

Servo control is the most basic element of the CSR cable-length control. The purpose of servo control is to ensure that the cables track some reference lengths. One servo loop is applied to each cable in the system. Therefore, each servo loop is independent and has no information about the other cables. The main performance goals of the servos are to control the cable lengths quickly, accurately, and in a variety of conditions.

In Section 3.1.7, it was proposed that DC motors and spools be used to wind the excess cable lengths. Then, each cable length can be controlled by rotating the respective spool. In Section 3.1.6, the rotary encoder was presented as one way to measure changes in the cable length. The use of rotary encoders is so common that it is possible to purchase a servo motor as a complete package. This package integrates the DC motor, rotary encoder, and controller. Then, the servo requires no further design other than installing it to the desired application. To select the servo motor, one would be interested in knowing the power requirements, signal types, and response characteristics. To give an idea of these details, a simple servo motor will now be designed.

Figure 61 depicts a simple model of the DC servo system. The DC motor is powered by an input voltage on the left and produces a torque on the spool. Meanwhile, the spool also experiences the load force, T_L, from the mass. Though not shown here, the spool is connected to the mass by a rigid, massless cable. Therefore, the same T_L force acts upward on the mass and is counteracted by the weight of the mass. The vertical position of the mass is labeled z, and the angular position of the motor is θ.

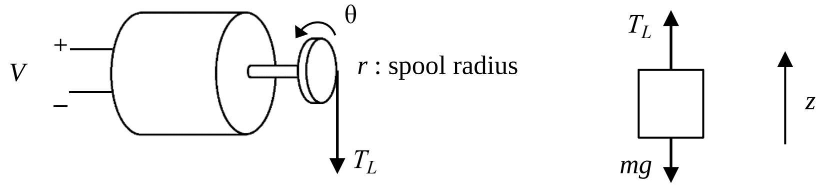


Figure 61. Simple model of a DC motor with an external mass load.

The simple model presented here represents one of the servo motors of the CSR system. In the CSR, the cables run upward from the spool and pass through some pulleys. Then, the cables descend to the suspended mass. All of the cables in the system are attached to the same mass. Therefore, one difference between the model in Figure 61 and the actual CSR is that the effective load for each cable varies in the actual system. The actual load on each cable depends on many factors, such as the position of the CSR and the acceleration of the mass. Another difference is that the mass in the CSR moves in 3D, whereas the simple model only accounts for a mass movement in the vertical direction. To justify the servo design, the simple model will be required to provide control for a variety of mass loads. If the performance of the servo is satisfactory for a variety of loads in the simple model, then it can be expected to also perform well in the CSR system, where the effective load on the DC motor will also vary.

The modeling for the DC motor begins with the mechanical and electrical governing equations, which are

$$K_t i = J\frac{d^2\theta}{dt^2} + B\frac{d\theta}{dt} + rT_L \tag{58}$$

and

$$V = L\frac{di}{dt} + R_m i + K_e\frac{d\theta}{dt}. \tag{59}$$

The explanations and some representative values for the model variables are listed in Table 2. The same values are later used in simulations of the servo motor control and also coincide with the experimental hardware in Chapter 4.

Table 2. DC motor variables, descriptions, and representative values.

Symbol	Description	Representative value
i	Motor current	1 A
V	Motor voltage	1 V
K_t	Motor torque constant	0.0202 N·m/A
J	Rotor inertia	1.24×10^{-6} kg·m^2
B	Motor viscous damping	1×10^{-6} N·s/rad
r	Spool radius	1 cm
L	Motor inductance	0.86 mH
R_m	Motor resistance	8 Ω
K_e	Motor electrical constant	0.0202 V·s/rad

Now, the governing equation for the mass movement is

$$m\ddot{z} = T_L - mg. \tag{60}$$

To couple the motor and mass equations, the kinematic relationship between the motor rotation and the mass movement is used. That is,

$$\dot{z} = r\dot{\theta}. \tag{61}$$

where r is the radius of the motor spool.

After taking the Laplace transform of (58−61) and performing some algebra,

$$\begin{aligned} Z(s) = {} & V(s)\frac{K_t r}{(LJ+Lmr^2)s^3+(JR_m+BL+mr^2R_m)s^2+(BR_m+K_tK_e)s} \\ & - G(s)\frac{mr^2(Ls+R_m)}{(LJ+Lmr^2)s^3+(JR_m+BL+mr^2R_m)s^2+(BR_m+K_tK_e)s}. \end{aligned} \tag{62}$$

The output of this system $Z(s)$ is the vertical position of the mass, and the two inputs are the gravity acceleration $G(s)$ and the motor voltage $V(s)$. Of course, gravity acceleration is a constant, so $G(s)$ is eventually replaced by a step input. The $V(s)$ input, which is the motor voltage, will be the output of the servo controller. This controller calculates the motor voltage based on the position error of the mass. Thus, the complete servo system

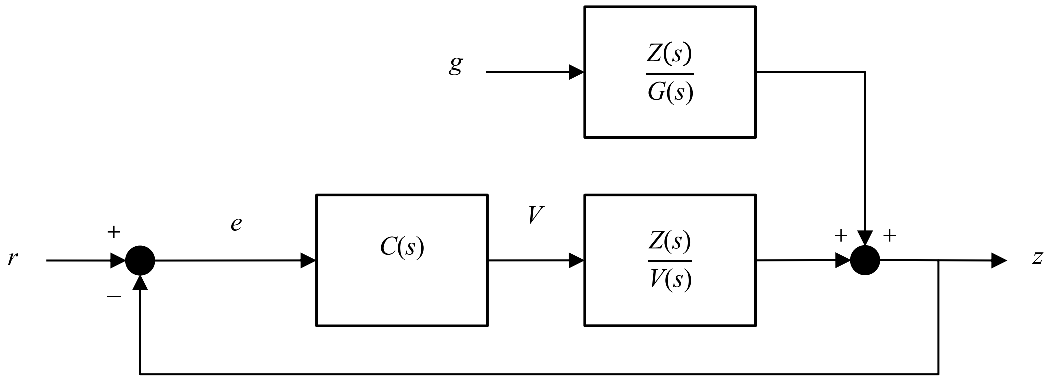


Figure 62. Block diagram of the servo control loop.

can be represented as the block diagram shown in Figure 62. In this closed-loop system, the reference mass position is the input and the actual position is the output. The position is fed back and compared with the reference, creating the error input, *e*, for the controller, $C(s)$.

The remaining task is to design the controller such that the system in Figure 62 is stable and provides a good reference-tracking performance. Suppose that the common PID controller is used. The transfer function for this controller is

$$C(s) = k_P + \frac{k_I}{s} + k_D s, \tag{63}$$

where k_P, k_I, and k_D are the controller gains. However, this form of the controller is not proper, meaning that the degree of the numerator is greater than that of the denominator. If the controller is implemented digitally, with the sampling period T_s, the transfer function can be modified to accurately represent the effects of digital implementation of the derivative term [66]. The modified form is

$$C(s) = k_P + \frac{k_I}{s} + \frac{k_D s}{1 + \frac{T_s}{2} s}, \tag{64}$$

which is a proper system. With this form of the controller, the loop transfer function of the servo system is

$$L(s) = \frac{\left(k_P + \frac{k_I}{s} + \frac{k_D s}{1 + \frac{T_s}{2} s}\right) K_t r}{(LJ + Lmr^2)s^3 + (JR_m + BL + mr^2 R_m)s^2 + (BR_m + K_t K_e)s}. \tag{65}$$

Rewriting (65) as a fraction of polynomials would reveal that the loop transfer function has two poles at the origin and relative degree of three. Therefore, the Bode plot of (65) should exhibit the DC gain and phase of infinity and −180°, respectively. On the other hand, the high-frequency gain and phase of 0 and −270° is expected. Therefore, the system crosses over the unity gain and −180° phase at some frequencies. These crossover points give the sufficient stability margins of the closed-loop system.

As one example, the gains of $k_P = 500$, $k_I = 500$, and $k_D = 50$ were selected. With a sampling period of $T_s = 10$ ms, the Bode plot of the loop transfer function is as shown in Figure 63. The low- and high-frequency results are as expected, and there are ample gain and phase margins, meaning that the system is stable with sufficient robustness. Furthermore, the large low-frequency gain and small high-frequency gain mean that this closed-loop system will be effective for tracking low-frequency reference signals while suppressing high-frequency disturbances.

Now that a controller has been selected, the closed-loop response of the servo loop is tested in simulation. The model parameters from Table 2 are used, and a variety of masses and reference inputs are tested to determine if the servo loop performs well in

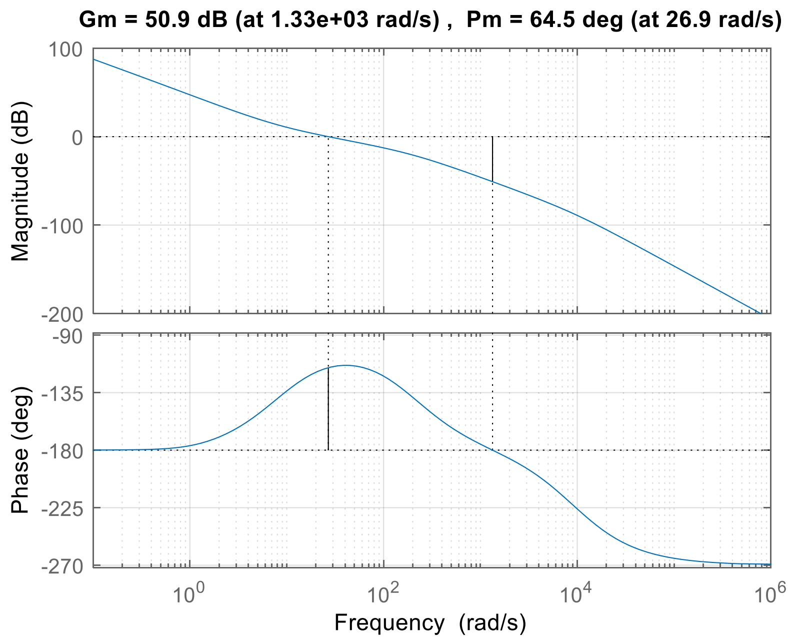


Figure 63. Bode plot for the loop transfer function of Figure 62. The PID controller in (64) was used.

various conditions.

In the first simulation, the end-effector mass is 0.5 kg, and the reference command is a ramp input with the slope of 0.1 m/s. This simulation is plotted in Figure 64. Initially, the mass position goes negative. This is because the mass starts from rest and it takes some time for the motor to increase its torque and counteract the weight of the mass. Nevertheless, the maximum error is relatively small at just over 3 cm. The error soon begins to decrease and becomes very small after about 2 s. The zero asymptotic error is due to the integral term in the PID controller. Without the integral term, the ramp input would produce a steady-state error. The motor voltage and cable tensions exhibit an initial spike in reaction to the sudden mass acceleration. However, these quantities soon reach a steady-state value to produce the constant upward velocity for the mass. The steady-state cable tension is slightly larger than the mass weight to overcome the motor damping.

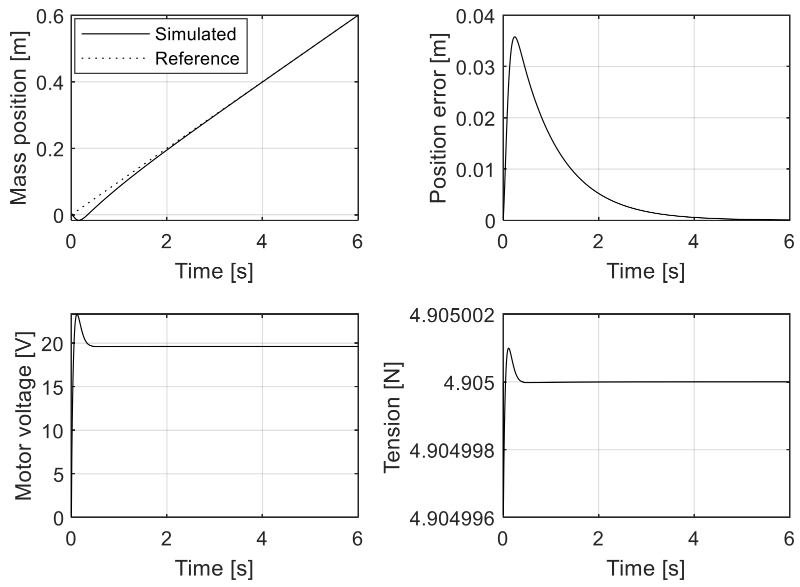


Figure 64. Ramp response for the servo system with a 0.5-kg mass.

The second simulation has the same ramp input as Figure 64, but the mass is decreased from 0.5 kg to 0.1 kg. This drastic change in mass represents the servo system having a much lighter load than expected. If the controller was not robust, this kind of change could cause instability. The new simulation, shown in Figure 65, indicates that the response is not only stable but improved with the smaller mass. The error plot looks almost identical to that in Figure 64, but the scale of the error is ten times smaller. The smaller error is reflected in the position plot, where the mass follows the reference closely from the start. The motor voltage and cable tensions are also much smaller than in Figure 64, owing to the smaller weight of the mass. Plots like these can be used to determine the suitability of the servo system for a given task. For example, the motor voltage plot should stay within the power supply limits, and the cable tensions should stay within a safe range.

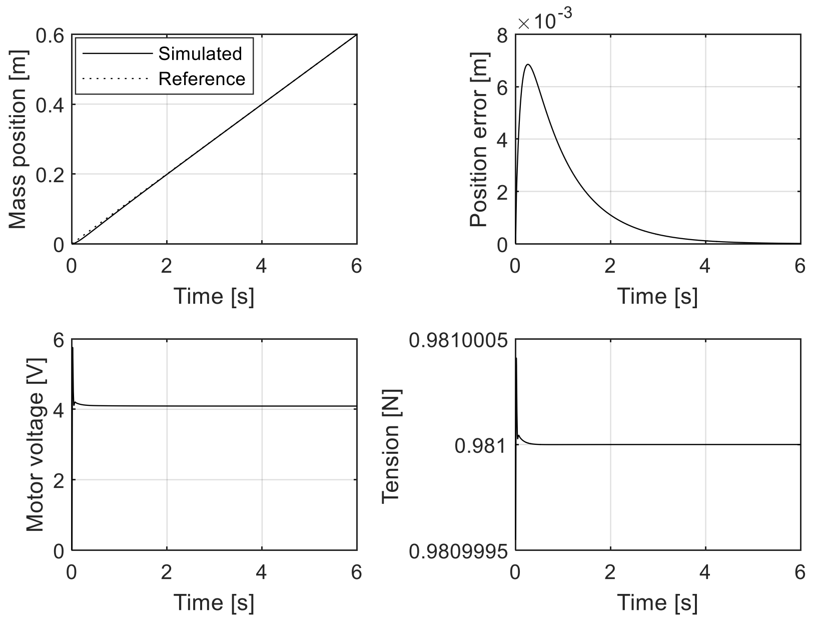


Figure 65. Ramp response for the servo system with a 0.1-kg mass.

The third simulation is again performed with the 1-kg mass. This time, the reference input is a sine function with a 1-m amplitude and 0.25-Hz frequency. This simulation is plotted in Figure 66. The reference curve is indistinguishable from thesimulated response, so only the response is plotted in the position plot. The tracking error remains within a 1-cm bound, but it does not go to zero asymptotically. This is because the sine input is an infinite-order polynomial that is difficult to track. However, as long as the error bound is small enough for the intended application, the tracking performance is acceptable. In steady state, the error oscillates about zero at the same frequency as the reference input. Unsurprisingly, the motor voltage and the cable tensions also vary with the same frequency. The motor voltage and cable tension do not have a zero mean because these must always be positive in order to support the suspended mass.

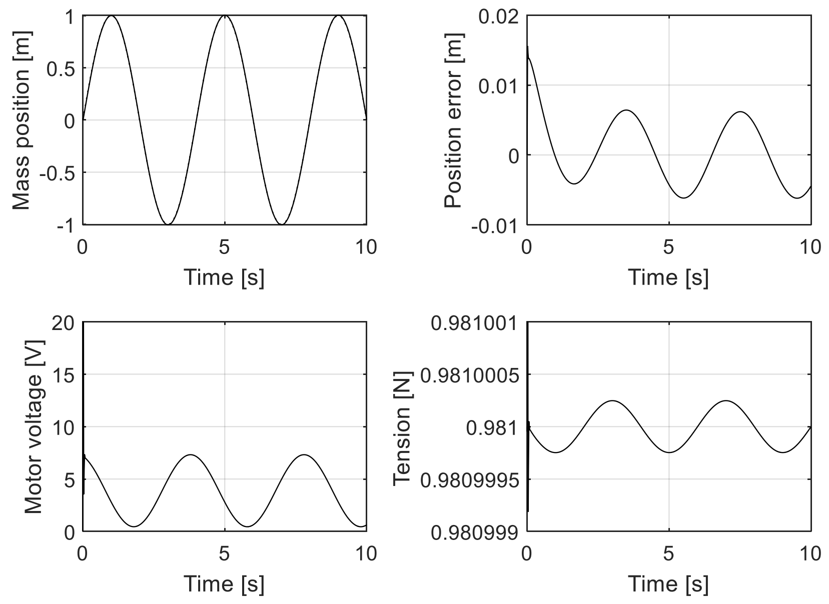


Figure 66. Sine tracking for the servo system with a 0.1-kg mass. The reference function has the amplitude of 1-m and the frequency of 0.25 Hz.

Finally, the sine reference input is tested with the 0.1-kg mass and an increased frequency of 0.5 Hz. This simulation is plotted in Figure 67. Once again, the reference signal is omitted because it is indistinguishable from the response of the system. In this simulation, the reference input produces movement speeds twice as fast as the 0.25-Hz reference. As a result, the error bound increases. Otherwise, the characteristics of the response are the same as seen in Figure 66. This simulation confirms that, in general, faster movement speeds will result in a larger tracking error for the servo system.

Whether a pre-made servo motor is used or the servo loop is newly designed, it is convenient to approximate the cable-length servo loop using a simplified model, such as

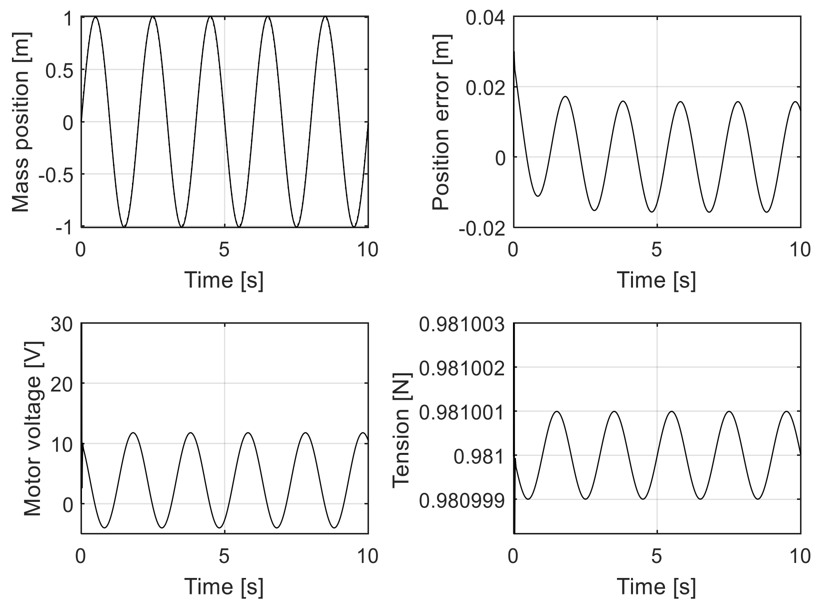


Figure 67. Sine tracking for the servo system with a 0.1-kg mass. The reference function has the amplitude of 1-m and the frequency of 0.5 Hz.

the second-order differential equation

$$\omega_n^2 l_r = \frac{d^2 l}{dt^2} + 2\zeta\omega_n \frac{dl}{dt} + \omega_n^2 l, \tag{66}$$

where l_r is the reference cable length and l is the actual cable length. The natural frequency, ω_n, and damping factor, ζ, are selected so that the response of the model is an acceptable representation of the system. If the servo response is sufficiently fast and accurate, a model like (66) is an acceptable simplification.

The simulations in Section 3.2.5.2 use model (66) for the cable lengths. This is done to add realistic time delays and dynamics to the changing cable lengths. Since the CSR movements are slow, the tracking errors are also small. For the experiment presented in Chapters 4−5, the PID controller developed in the present section is used to implement servo loops in real time, and experiments are performed to verify that the cable-length tracking errors are small.

3.2.5.2. Dual-Loop Control

With the selection of the servo loops that control the cable lengths, the remaining task is to design an outer loop for the CSR system. Ultimately, the outer loop provides the reference cable lengths to the inner-loop servos. In an ideal case, the outer loop can calculate the reference cable lengths simply based on the desired position of the end effector and the robot kinematics. However, disturbances to the kinematics or the end effector produce errors. Therefore, an external sensor can be used to continuously measure the true position of the end effector, and this data can be used to adjust the reference cable lengths in real time. Some examples of sensors that can be used for this purpose were discussed in Section 3.1.6.

Once again, a four-cable configuration like that in Figure 22 or 51 is considered. The dual-loop control structure for this configuration is shown in Figure 68. There are four servo loops, one for each cable, that control the cable lengths. The true cable lengths determine the position of the end effector according to the forward kinematics of the system. Furthermore, the position is affected by some disturbances. The outer-loop controller provides the reference cable lengths to the servo loops based, primarily, on the reference position. The true position can also be used to adjust the reference lengths, but this requires the use of an external sensor.

A critical element of the dual-loop structure is the outer-loop controller. One design for this controller is now presented. Recall that the ideal cable lengths for a given

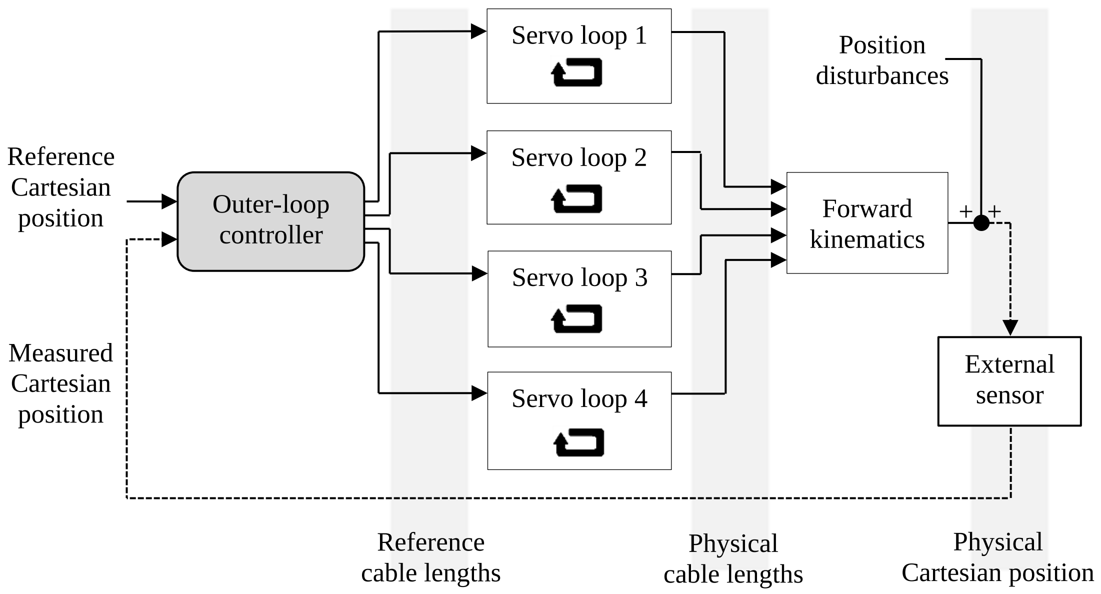


Figure 68. One example of a dual-loop control structure for a four-cable CSR. Each cable-length is individually controlled by a servo loop. The outer-loop provides the reference cable lengths to the servo loops. The reference lengths are based on the reference and true position of the end effector. The dashed lines indicate the flow of information enabled by the external position sensor.

end-effector position are calculated using (3). This equation is also referred to as the inverse kinematics of the system. For cable *i* the ideal reference length is

$$l_{ir} = \sqrt{(x_i - x_r)^2 + \left(y_i - y_r\right)^2 + (z_i - z_r)^2}, \tag{67}$$

where (x_r, y_r, z_r) is the reference position and (x_i, y_i, z_i) is the location of the anchor point for that cable. This calculation can be repeated, but using the measured position of the end effector, (x_m, y_m, z_m), instead of the reference position. This allows for the definition of a new cable length, called the measured cable length, that is based on the data collected by the external position sensor. That is,

$$l_{im} = \sqrt{(x_i - x_m)^2 + (y_i - y_m)^2 + (z_i - z_m)^2}. \tag{68}$$

To clarify, there are three different cable lengths defined in the system. The first length is the true, physical cable length that is controlled by the servo loop. The second length is the ideal reference cable length given by (67). The third length is the measured cable length given by (68). This quantity can be seen as the predicted cable length based on the measured end-effector position and the ideal CSR configuration.

If the position of the end effector is not measured (68) cannot be calculated and the reference cable length can be calculated simply as (67). However, if the position measurements are available, then the measured cable length can be compared against the reference cable length to evaluate the real-time performance of the system. If there is no position error for the end effector, the complete system is working as intended and l_{ir} and l_{im} will coincide. If there is a position error, then l_{ir} and l_{im} will differ. This difference is called the cable-length error and is defined as

$$l_{ie} = l_{ir} - l_{im}. \tag{69}$$

It is proposed that the reference cable lengths provided to the servo loops be adjusted using l_{ie}. For example, the adjusted reference cable length can be defined as

$$l'_{ir} = l_{ir} + k_I \int l_{ie}\, dt. \tag{70}$$

Here, the cable-length error is integrated and multiplied by an integral gain, k_I. Thus, (70) can be seen as an integral controller with the feed-forward term, l_{ir}. A block-diagram representation of the controller is shown in Figure 69.

From (69), it is evident that anything that causes a change in the measured position away from the reference position will also cause a change in the reference cable lengths,

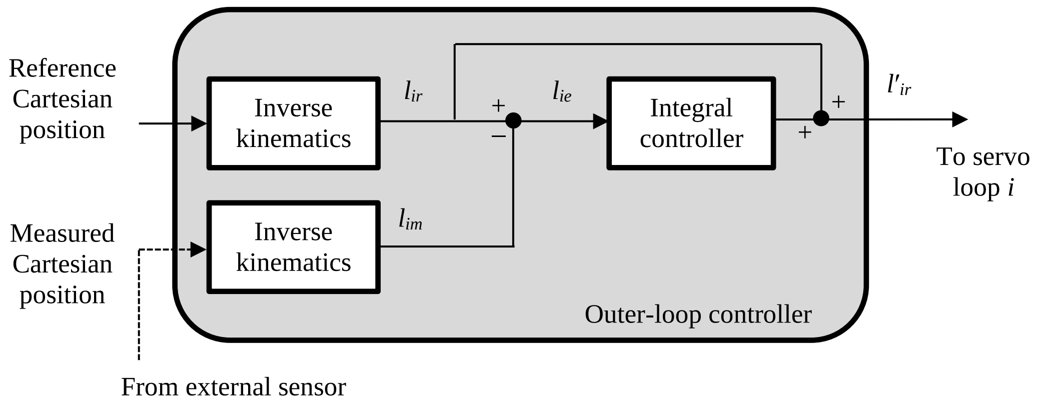
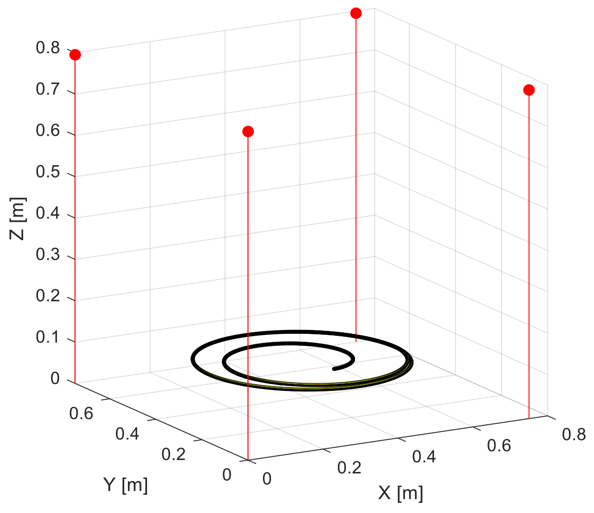


Figure 69. Outer-loop controller with an integral and feed-forward component. The same control method is applied to every cable in the CSR.

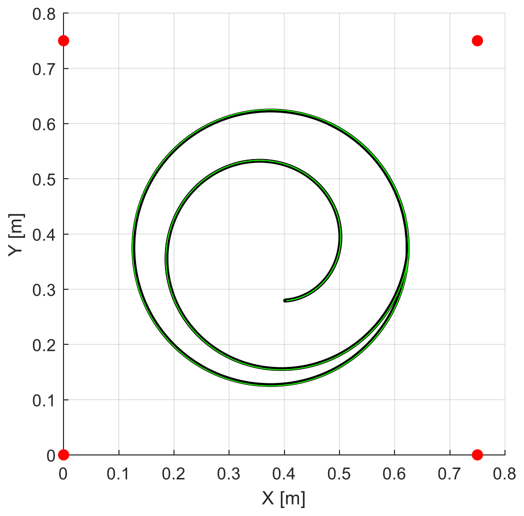
causing the closed-loop CSR system to react in some way. Therefore, the proposed controller effectively converts the measured-position error into the cable space. Some of the possible sources for the measured-position error are: errors in the true cable lengths, errors in the anchor-point placement, and position-sensor errors, and dynamic force disturbances. These errors will now be simulated to test whether the proposed outer-loop controller is stable and effective.

The subject of simulation is a four-cable CSR with the side lengths of 0.75 m and the height of 0.795 m. These dimensions were selected to match the experiment of Chapter 4. The mass of the end effector is 1.0 kg, and the system's mechanical damping was selected as 2 N·s/m. This value was estimated by repeated simulations and comparison to the real experiment. The cable tensions were modeled using (31), with the same material properties that were used to create Figure 49. Finally, (66) was used to model the cable-length servo loops, with ω_n = 50 rad/s and ζ = 0.9. These were selected by comparing the simulation results to the experimental observations.

First, the ideal performance of the system is simulated. In this case, there are no disturbances to the system, and the reference-cable lengths are calculated using (67). The resulting motion trajectory is plotted in Figure 70. The reference trajectory is a spiral path at the consant height of 1 m, and the CSR follows it almost exactly, such that the reference and actual trajectories coincide in the 3D plot. Figure 71 provides more details on the position of the mass throughout the movement in this simulation. Again, the reference and true positions of the mass coincide almost exactly. The difference between the reference and actual positions in the *z* direction is noticeable only because of the small scale on the vertical axis. The error plots show that the mass remains within a few millimeters of the reference trajectory in all three coordinates. The *x*- and *y*-direction errors are periodic with zero mean and about the same frequency as the spiral movement. The *z*-direction error



(a)



(b)

Figure 70. Simulation result for a four-cable CSR with cable-length control. This represents the ideal scenario, when there are no errors in the system configuration or cable lengths. (a) Side and (b) top views.

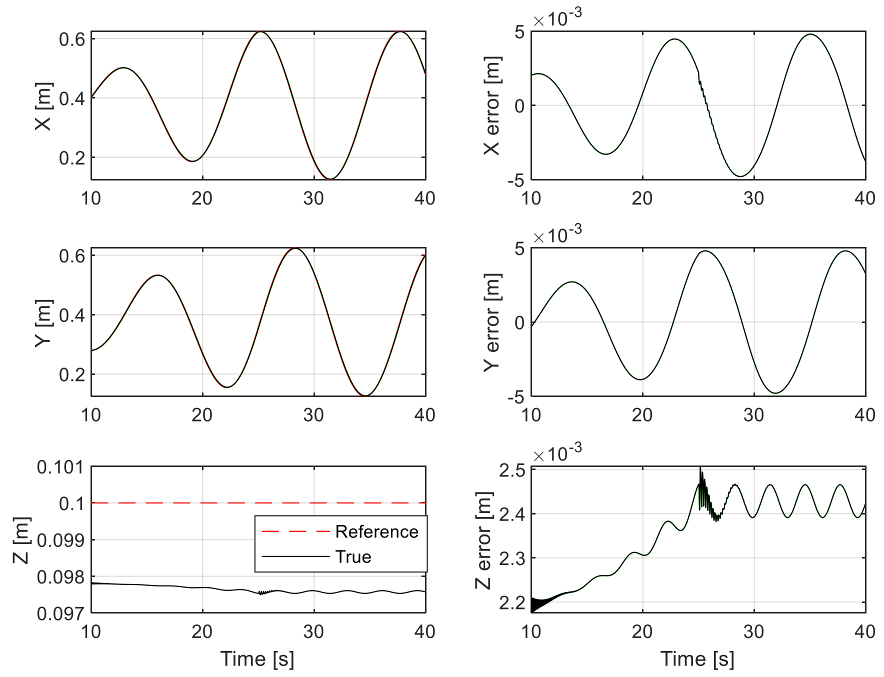


Figure 71. Cartesian position and error of the four-cable CSR. This is from the ideal simulation plotted in Figure 70.

does not have a zero mean but also varies periodically, albeit at a higher frequency and smaller amplitude. This behavior, along with the *x*-*y* error, is a result of the cable stretch not being accounted for in the control law. The result is that the mass hangs slightly lower than the desired height and lags in the *x*-*y* motion. This error is acceptable if it remains within a predetermined error bound. The slightly-chaotic behavior in the error plots around 25 s is insignificant and due to the circular motion suddenly changing to a constant radius.

Some of the error in the mass position is also due to errors in the cable lengths. Figure 66 suggested that a simple servo loop will exhibit a small error for periodic reference inputs. The same behavior is seen in the cable lengths of the ideal simulation (Figure 72). Here, the cables are labeled using alphabet letters, with cable L_a being the

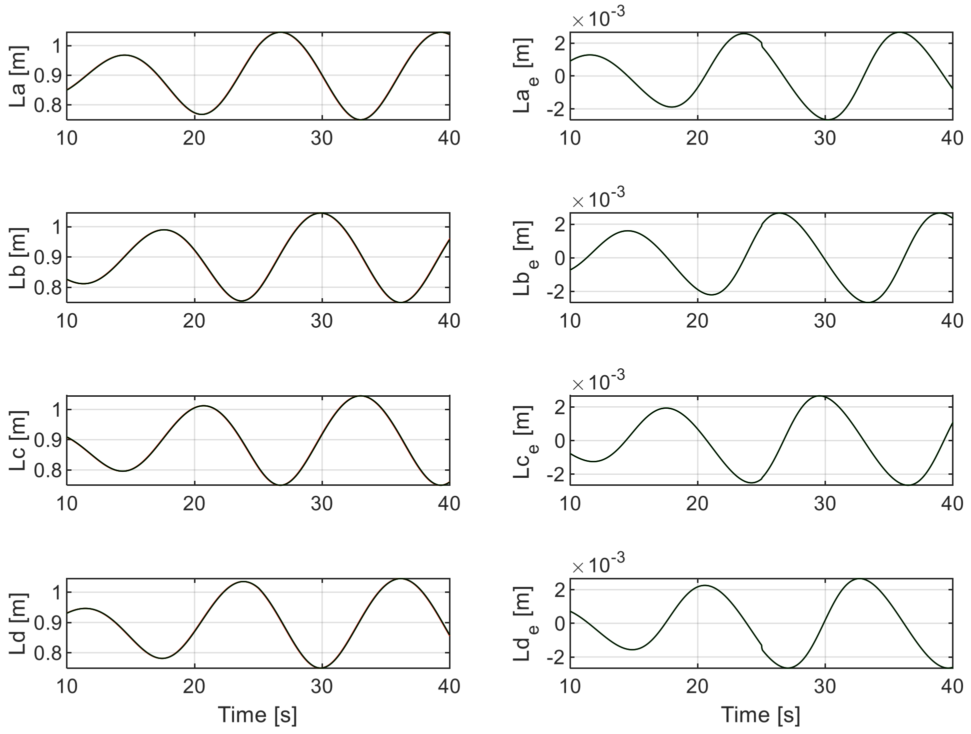
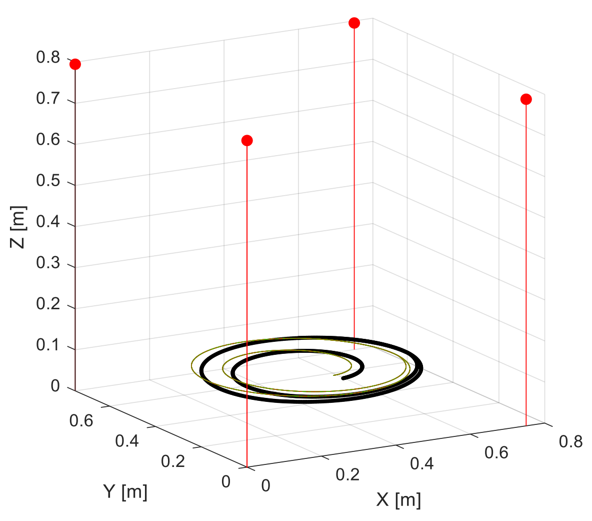


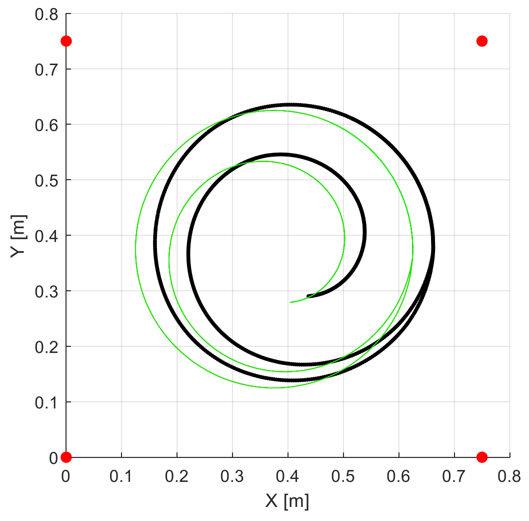
Figure 72. Cable lengths and errors for a four-cable CSR. This is for the ideal simulation plotted in Figure 70.

cable anchored above the origin and the subsequent cables labeled in the counter-clockwise direction when the CSR is viewed from above.

The first source of error that is investigated is cable-length errors. To do this, the true cable lengths of the system are disturbed by constant amounts. Furthermore, it is assumed that the servo loops contain no knowledge of the length errors. This kind of error can occur in a real-life application if the servo loops are not properly calibrated or if the effective cable lengths change unexpectedly during the CSR operation. The effects of this kind of error are demonstrated in Figure 73. Again, the cable lengths are controlled using (67), but the actual lengths produced are incorrect due to the length errors. The result is a



(a)



(b)

Figure 73. Simulation of a four-cable CSR with cable-length control and cable-length errors. The bold line is the simulated trajectory, and the thin line is the reference trajectory. (a) Side and (b) top views.

noticeable discrepancy between the mass position and the reference trajectory. The plot of cable lengths for this simulation, shown in Figure 74, clearly show that the true and encoder-measured cable lengths are separated by a constant amount, owing to the constant cable-length errors. Recall that the encoders are the feedback sensors used to measure the true cable lengths in the servo loops. If these encoders are not setup accurately, or if there are any missed encoder counts, there will be a difference between the true and measured cable lengths. As a result, there will be a tracking error for the cable lengths like that shown in Figure 74. The length errors used in this simulation were −3 cm for L_a, +1 cm for L_c, and −2 cm for L_d. These quantities are evident in the cable-length error plots. The only cable without an error is L_b, which has a response similar to the ideal case in Figure 72.

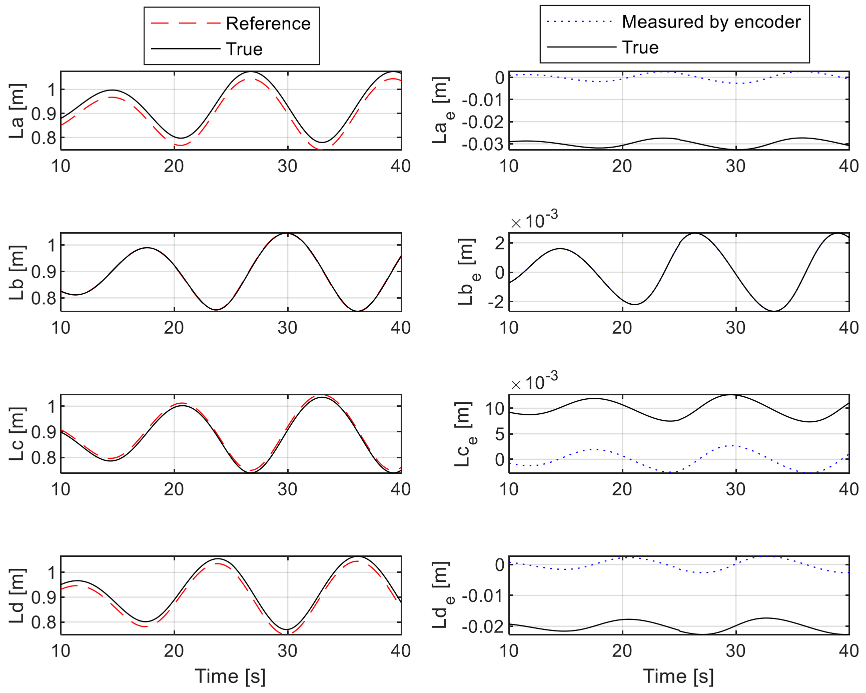


Figure 74. Cable lengths for the four-cable CSR simulation from Figure 73.

Now, the same simulation with cable-length errors is repeated, this time using the closed-loop feedback control law (70). The k_I value used in this control law was 2. The result of this simulation is plotted in Figure 75, and the time plot of the mass position is shown in Figure 76. The 3D plot is nearly indistinguishable from the ideal case plotted in Figure 70, demonstrating that the control law is effective in correcting for the cable-length errors. The time plot of the mass position confirms that the position errors in the Cartesian coordinates is reduced to a few millimeters. In fact, the position error is reduced even compared to the ideal case plotted in Figure 71. Again, the chaotic movement at 25 s is due to the radius of the movement suddenly becoming a constant.

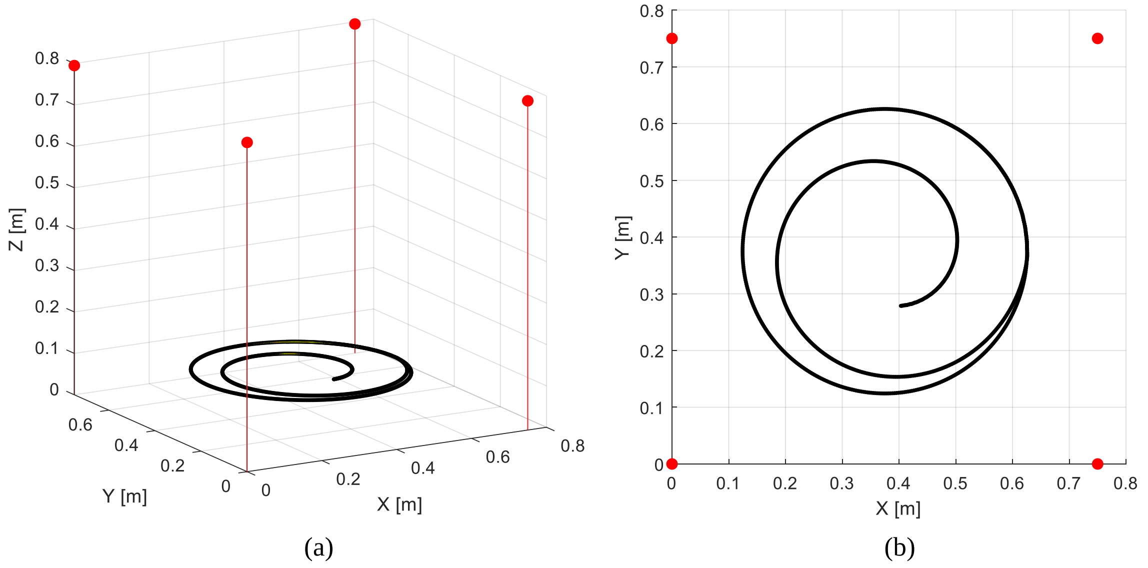


Figure 75. Simulation of a four-cable CSR with cable-length control and cable-length errors. The closed-loop control law (70) was used to correct for the cable-length errors. (a) Side and (b) top views.

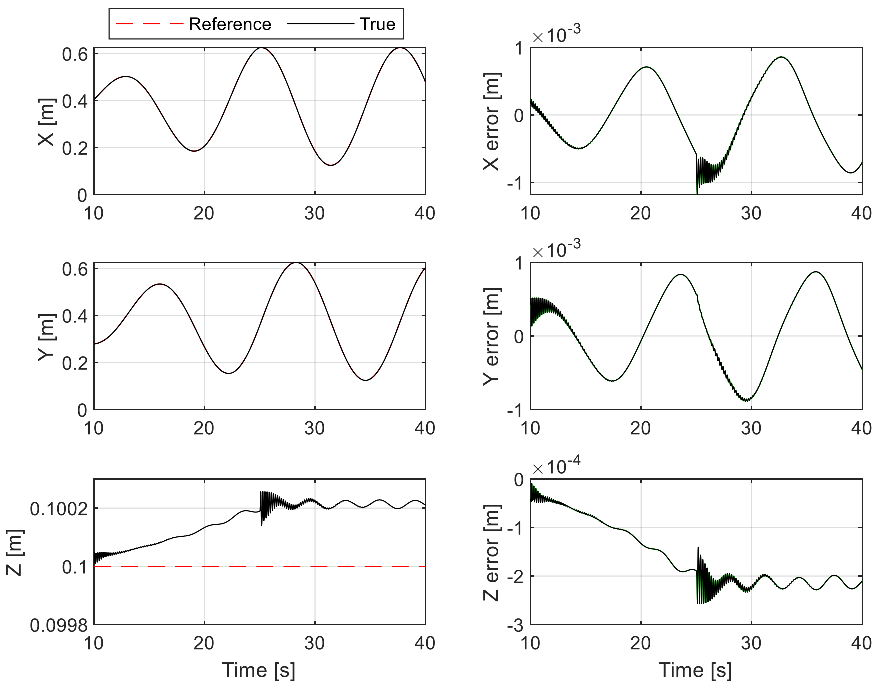


Figure 76. Cartesian position and errors for the simulation plotted in Figure 75.

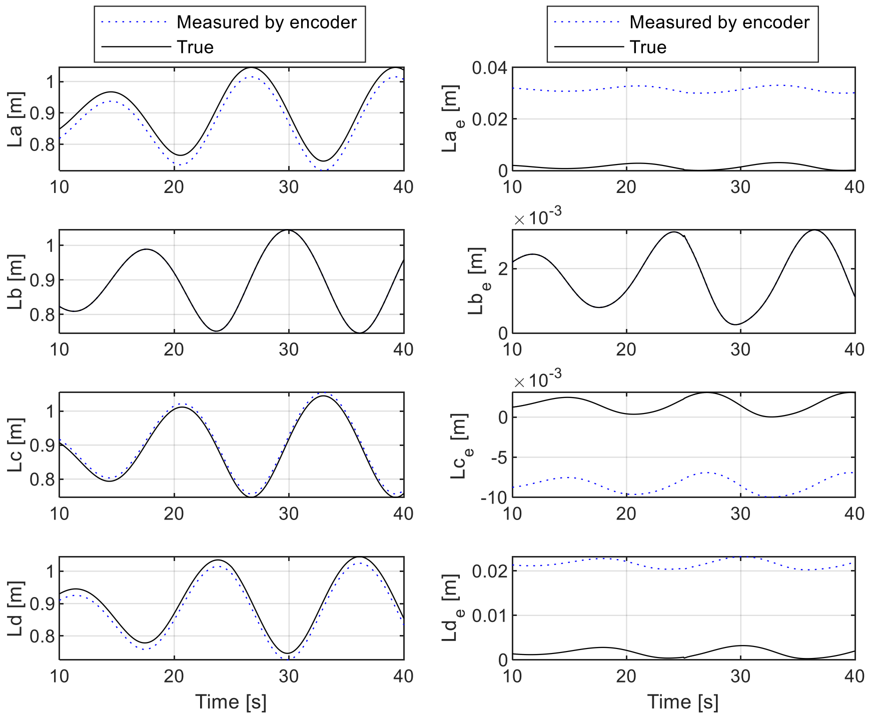


Figure 77. Cable lengths for the simulation plotted in Figure 75.

The cable lengths for the same simulation are shown in Figure 77. The true cable lengths are now near zero owing to the feedback control law. Compare this to Figure 74, where the encoder-measured lengths were near zero but the true cable lengths had a large error. Interestingly, the true cable-length errors are not zero mean but are slightly positive. This means that the true cable lengths are slightly shorter than the ideal lengths given by (67). Thus, the control law helps correct for the cable stretch and produces a more accurate position for the mass.

The second source of error that was investigated was cable anchor-point errors. These errors result from inaccurate placement of the cable pulleys, and they alter the system kinematics. More specifically, an error in the anchor-point placement alters the expected cable lengths and directions. To test this scenario, a three-dimensional position error was randomly generated for all four cable anchor points. The anchor-point errors were $(-0.047, 0.006, 0.038)$ m, $(0.017, -0.031, -0.013)$ m, $(-0.004, 0.048, -0.034)$ m, and $(0.036, 0.015, -0.012)$ m for cables a, b, c, and d, respectively. Then, the CSR response was simulated using the control law (67). Again, this is the control law without the position feedback, where the reference cable lengths are calculated using the ideal anchor-point coordinates. The result of this simulation is plotted in Figure 78. The two views demonstrate the drastic effect that the anchor-point errors have on the reference tracking. Internally, the CSR system has no information of the anchor-point errors. In fact, a plot of the cable lengths in this simulation is identical to the ideal case in Figure 72, meaning that the reference lengths are faithfully tracked by the servo loops. The tracking errors are due to the fact that the true CSR configuration is not as expected.

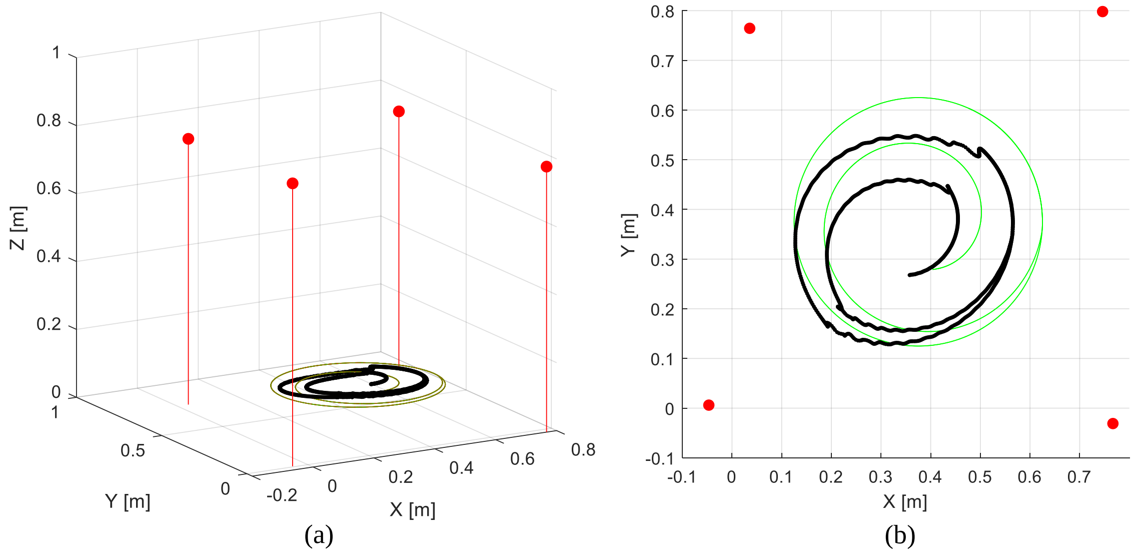


Figure 78. Simulation of a four-cable CSR with cable-length control and anchor point-position errors. The control law (67) without position feedback was used. (a) Side and (b) top views. The bold line is the actual trajectory, and the thin line is the reference.

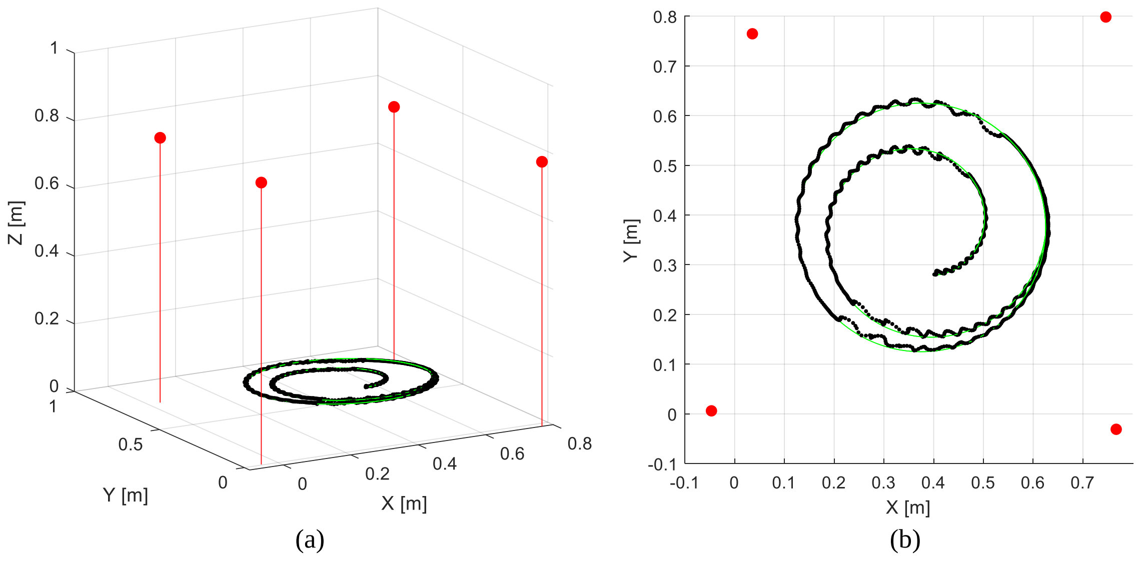


Figure 79. Simulation of a four-cable CSR with cable-length control and anchor point-position errors. The closed-loop control law (70) was used, as opposed to the simple control law used in Figure 78. (a) Side and (b) top view.

Now, the simulation is repeated with the same conditions but with the position-feedback control law. The result is shown in Figure 79. Compared to Figure 78, the tracking error is much reduced. The position is also plotted versus time in Figure 80, where it is apparent that the mass follows the reference trajectory to within a couple centimeters. However, there are some noticeable oscillations and peaks in the error plots. The same oscillations are evident in Figure 79. Regardless, the true trajectory is much closer to the reference than the case without feedback control. Furthermore, the anchor-point position errors in this simulation represent a worst-case scenario in some sense. All of the anchor points have a three-dimensional position error, and the error magnitudes are significant. In a real situation, one would hope to minimize such errors from the start. Nevertheless, the position-feedback control law is an effective method for reducing the position error.

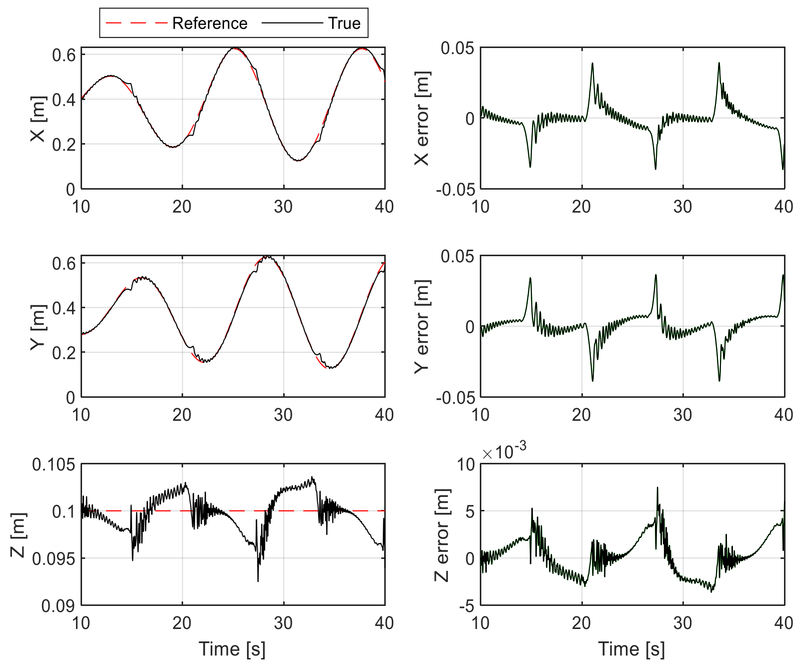


Figure 80. Position and position error for the simulation in Figure 79.

The third source of error investigated is position-sensor errors. Suppose that a camera or laser system is used to measure the position of the end effector for the purpose of feedback control. If this measurement has a systematic error, from improper calibration or otherwise, it is useful to know how the system will respond even if there are no other errors in the system. To test this scenario, the position measurement is injected with constant errors. For instance, suppose that $(x_m, y_m, z_m) = (x, y, z) + (0.05, 0.02, -0.03)$ m. There are no others errors in the system. That is, the robot configuration and cable lengths are known exactly. This scenario was already presented in Figure 70, but without the position feedback component. With the position feedback and position-measurement error, the new control law will produce a change in the reference cable lengths even if the

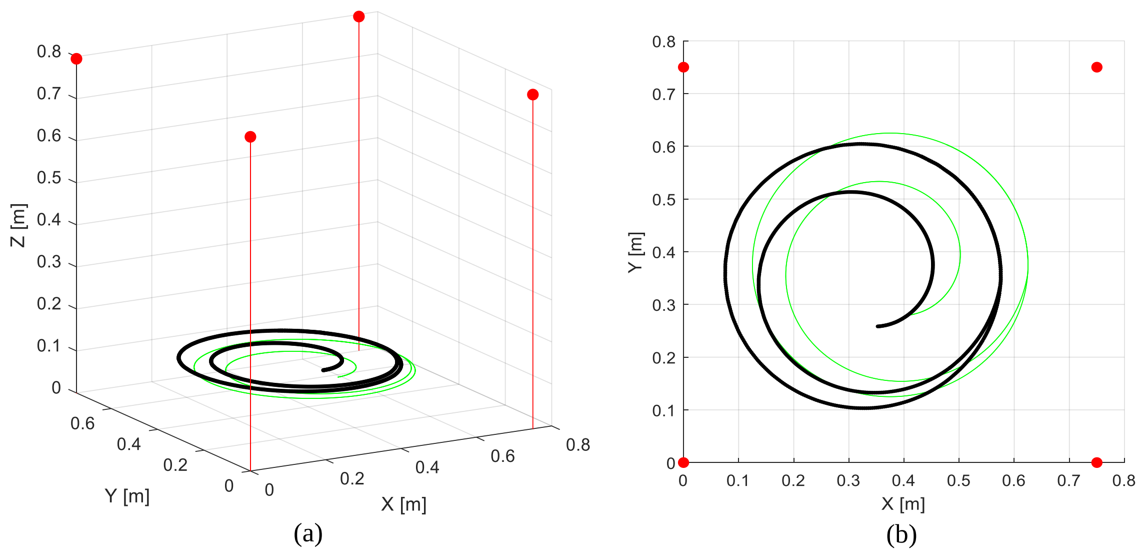
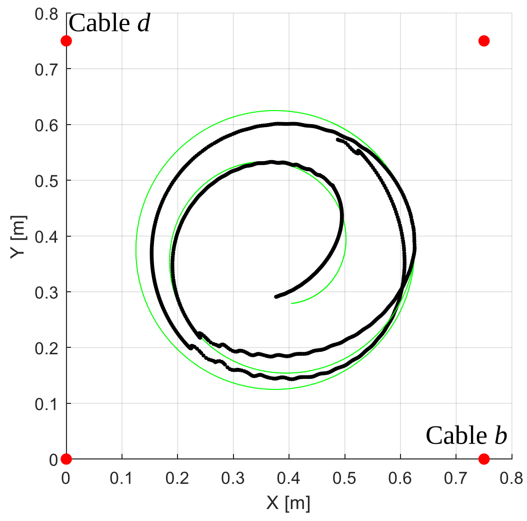
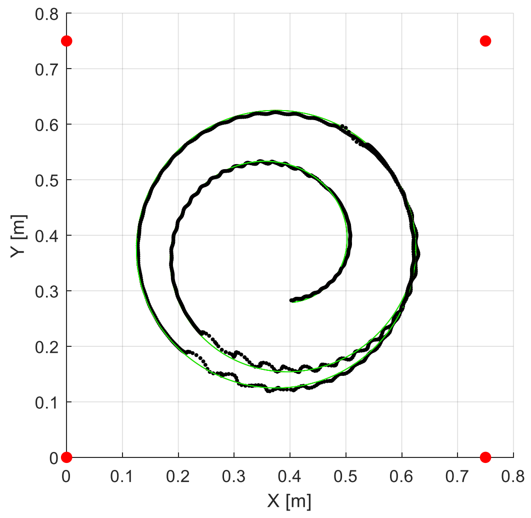


Figure 81. Simulation of a four-cable CSR with cable-length control and position-measurement errors. The closed-loop control law (70) was used. (a) Side view and (b) top views.

ideal trajectory closely follows the reference. The new response of the CSR is shown in Figure 81. Overall, the position-measurement error causes a constant offset in the position of the mass. The amount and direction of the offset is exactly the negative of the position-measurement error. In practical terms, the control law will cause the system to follow the reference trajectory as seen by the position-measurement sensor, even if this measurement has systematic errors. In the simulated example, there are no other noticeable distortions or oscillations in the trajectory, other than the shift in position. Therefore, a position-sensor error like this one could go unnoticed in a practical application. One way to detect the systematic error in the measurement is to operate the system with and without the position-feedback term and study the difference in the responses, keeping in mind that other factors also affect the trajectory errors.



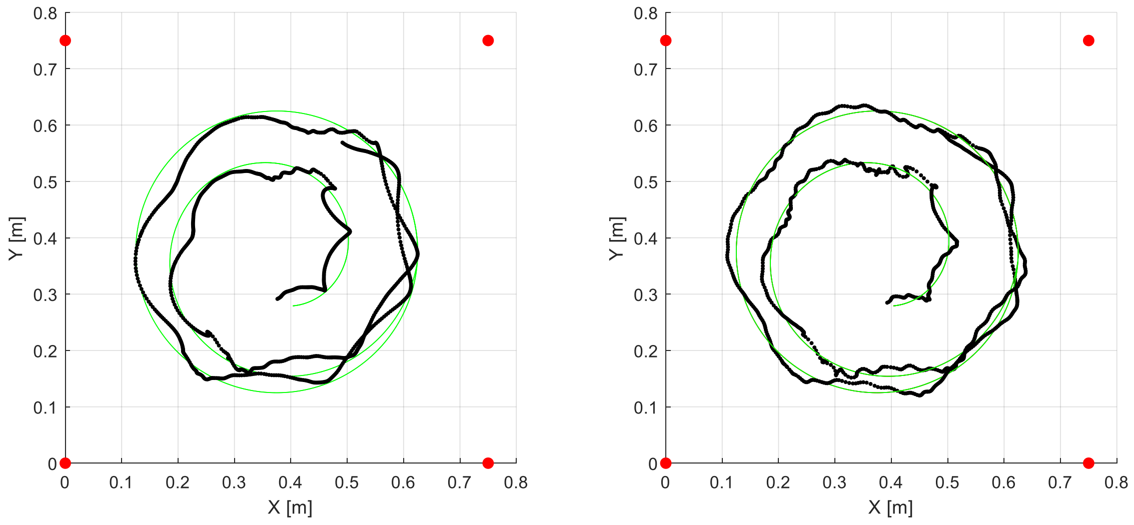
(a)



(b)

Figure 82. Top view of a four-cable CSR simulation with cable-length control. Two of the cables are subject to a 0.1-Hz periodic disturbance to their lengths. Responses (a) without and (b) with position feedback.

The final source of error investigated is dynamic force disturbances. Such disturbances could have many causes, including contact with foreign objects or environmental factors, like wind. Another possible cause is disturbances to the CSR cables themselves. To simulate this, two of the cable lengths in the four-cable configuration are disturbed using sine functions. Cables *b* and *d* are disturbed with the amplitude of 5 cm as the CSR tracks the reference trajectory. First, a low-frequency disturbance of 0.1 Hz is tested. The result of this disturbance, with and without the position feedback, is plotted in Figure 82. Without the position feedback, the mass trajectory is drastically affected, with as significant departure from the reference trajectory. With the position feedback, the trajectory closely follows the reference, with some oscillations like those in Figure 79.



(a) (b)

Figure 83. Top view of a four-cable CSR simulation with cable-length control. Two of the cables are subject to a 0.5-Hz periodic disturbance to their lengths. Responses (a) without and (b) with position feedback.

Thus, the position-feedback control law is effective in correcting the overall trajectory error, but there are still some small, high-frequency errors.

The simulation with dynamic cable-length disturbances is repeated, this time with a 0.5-Hz disturbance applied to cables *b* and *d*. The disturbance amplitude is the same as before. The result of this simulation is plotted in Figure 83. Again, the disturbances cause a significant error in the trajectory tracking without the feedback control law. However, the simulation with the feedback control is only slightly improved, as there is still a noticeable tracking error and some small oscillations. This suggests that the position-feedback control law is not as effective in reducing errors from high-frequency disturbances. This result is not surprising, since the mechanism in the control law, (70), for correcting position errors is an error integral term that has some lag in responding. One might also expect that increasing the integral gain, k_I can produce some oscillations in the response of the system. This explains, in part, the oscillations observed in Figures 79 and 82. With a large-enough k_I, the system becomes unstable. The k_I value used in these simulations was 2, and this value was selected after repeated trials. Values larger than 5 caused instabilities in the system especially when there were significant tracking errors. For any system with low-frequency disturbances, a small k_i value is recommended.

3.2.6. Anchor-Point Position Estimation

One of the possible causes of error in a CSR system is the misplacement of the cable anchor points. Figure 78 demonstrated the drastic effect that this error can have on the system trajectory. This is because the reference lengths used in cable-length control are primarliy based on accurate knowledge of the cable anchor-point locations. In this section, some novel methods to estimate the anchor point locations are investigated.

The anchor-point estimation methods are based on the assumption that the basic form of the robot kinematics is known, but that some of the parameters are only approximately known. Then, experimental data can be used to estimate the uncertain parameters. The methods begins with the equation for the ideal cable length (3). The equation is reproduced here for convenience.

$$l_i = \sqrt{(x_i - x)^2 + \left(y_i - y\right)^2 + (z_i - z)^2} \tag{71}$$

Recall that (x_i, y_i, z_i) is the ideal location of the cable anchor point and (x, y, z) is the location of the end effector. Then, (71) gives the Euclidean distance between these two points. The actual cable length will be slightly longer due to sagging or stretch.

Suppose that the anchor-point position for cable i is displaced by some small amount, given by the vector $[\delta x, \delta y, \delta z]^T$. Then, the new ideal cable length is

$$l'_i = \sqrt{(x_i + \delta x - x)^2 + (y_i + \delta y - y)^2 + (z_i + \delta z - z)^2}. \tag{72}$$

This equation can be expanded and rearranged as

$$l'^2_i - l_i^2 = 2(x_i - x)\delta x + 2(y_i - y)\delta y + 2(z_i - z)\delta z + \delta x^2 + \delta y^2 + \delta z^2, \tag{73}$$

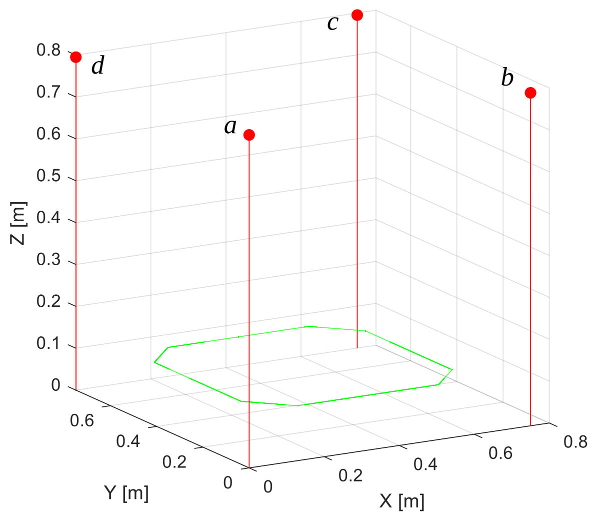
where (71) was substituted.

Now, suppose that an experiment is run using the cable-length control of the CSR. The data available from the experiment are the measured mass position and cable lengths at various times. These measurements can be substituted in (73) to produce the equation

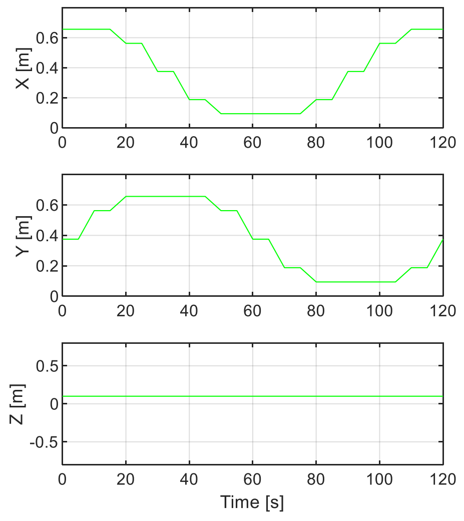
$$l'^2_{im} - l_{im}^2 = 2(x_i - x_m)\delta x + 2(y_i - y_m)\delta y + 2(z_i - z_m)\delta z + \delta x^2 + \delta y^2 + \delta z^2, \tag{74}$$

where the subscript m indicates the measured values. With these measurements, every quantity in (74) except the δ terms is known.

The immediate goal is to estimate the unknown terms using a series of measurements. To ensure that (74) is a valid relationship for the data, special care is taken to ensure that the cables are taut during the experiment. This could be an issue if there are redundant cables in the CSR, since they can exhibit sagging. One way to address this issue is to use a special trajectory where the cables are guaranteed to be in tension. For example, the trajectory pictured in Figure 84 produces a series of SEPs where at least one of the cables is guaranteed to be in tension. Then, the data collected during these equilibrium points are known to approximately satisfy (74) for the cable in tension. For instance, from 10 to 15 s cable c must be in tension, so the data during this period will satisfy (74) for



(a)



(b)

Figure 84. A special reference trajectory that is used to ensure that (74) is a valid relationship for the experimental data. At different vertices in this trajectory, certain cables are guaranteed to be in tension, even if there are anchor-point placement errors.

The first method assumes that (74) is a linear equation with four unknown terms.

this cable. Once data sets are collected that satisfy the equation for all cables, one of the following methods can be used to estimate the (δx, δy, δz) for each cable anchor point.

The four unknowns are δx, δy, δz, and $\delta x^2+\delta y^2+\delta z^2$. The last term is not independent of the first three terms. However, treating the fourth term as an independent constant allows for a straightforward solution. From four distinct time instances, four independent (74) equations are created. This system of equations will be linear with respect to the four unknowns and can be solved using basic linear algebra.

The second method recognizes that there are only three unknowns in (74). To calculate the unknowns, three independent equations are created from three distinct time instances. Then, the following optimization problem is posed

$$\min \left\| \begin{matrix} F(\delta x, \delta y, \delta z)_{t=t_1} - \left(l'^{\,2}_{im} - l^{2}_{im}\right)_{t=t_1} \\ F(\delta x, \delta y, \delta z)_{t=t_2} - \left(l'^{\,2}_{im} - l^{2}_{im}\right)_{t=t_2} \\ F(\delta x, \delta y, \delta z)_{t=t_3} - \left(l'^{\,2}_{im} - l^{2}_{im}\right)_{t=t_3} \end{matrix} \right\|_2 \tag{75}$$
$$\begin{gathered} s.\ t. \\ lb \le \delta x \le ub \\ lb \le \delta y \le ub \\ lb \le \delta z \le ub. \end{gathered}$$

where $F(\delta x, \delta y, \delta z)$ is the nonlinear function on the right side of (74). In other words, this optimization problem finds the values of δx, δy, and δz that best satisfy (74) for all three distinct time instances. If the equation is exactly satisfied, the objective value will be zero. However, due to system imperfections or measurement errors, the minimum objective value may only be close to zero. The optimization problem also allows for specifying the lower and upper bounds of the δ values. These bounds, *lb* and *ub* in (75), are selected based on the knowledge of the physical CSR.

A third method is developed using a geometric approach. This method is similar to a location-triangulation problem, as illustrated in Figure 85. The illustration shows one circle drawn around the mass at different time instances. The radius of the circle is equal to the measured cable length at that instance. Ideally, the circles will intersect at the true anchor point location. In 3D, the circles are spheres. If there is measurement noise in the mass position or cable lengths, the spheres will not intersect perfectly at the anchor point. Then, several additional measurements can be used to improve the estimate of the point.

Mathematically, this method can be expressed as the task of finding the point in space that is closest to all of the sphere boundaries. Then, this problem can be posed as

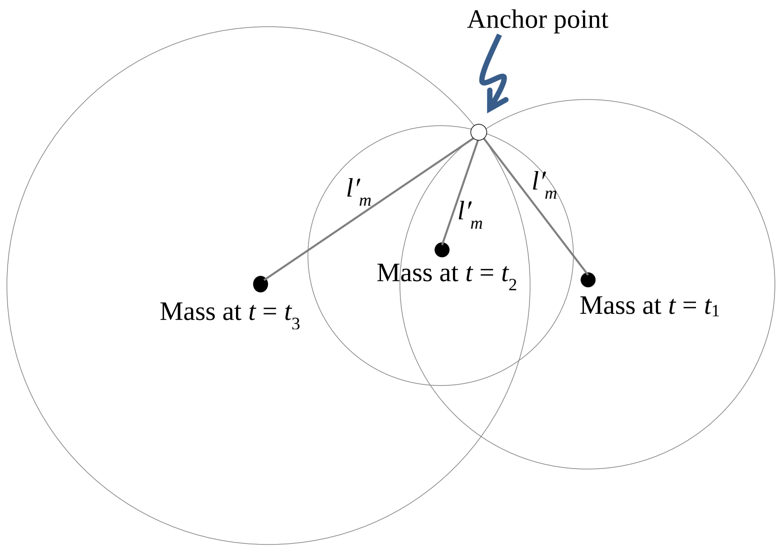
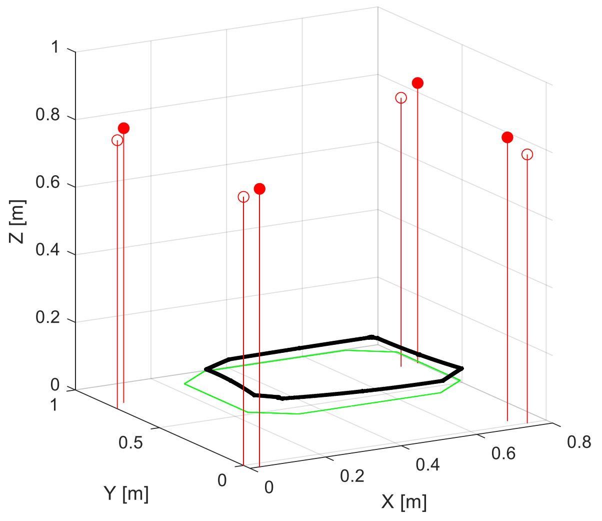


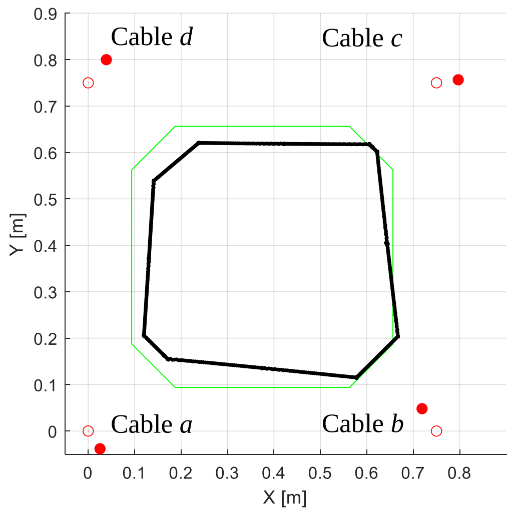
Figure 85. An illustration of how the cable anchor point can be estimated from successive measurements of the mass position and cable lengths.

$$\min \sum_{t_n = t_1}^{t_3} \left| \sqrt{(x_i + \delta x - x_m)^2 + \left(y_i + \delta y - y_m\right)^2 + (z_i + \delta z - z_m)^2} - l'_{im} \right|_{t = t_n}$$
$$\text{s. t.} \qquad (76)$$
$$lb \leq \delta x \leq ub$$
$$lb \leq \delta y \leq ub$$
$$lb \leq \delta z \leq ub.$$

The quantity in the sum is the distance between the estimated anchor point and the spherical boundary at time t_n. The term in the square root is the Euclidean distance from the mass position to the estimated anchor point. The l'_{im} term is the radius of the sphere, which is the measured cable length. Then, the absolute value of the difference between these two terms is the distance from the estimated anchor point to the spherical boundary. The objective value in (76) is the sum of this distance for the three time instances, and the goal is to find the anchor-point location that minimizes this sum. At least three time instances are required to identify the anchor point.



(a)



(b)

Figure 86. Simulation result for a four-cable CSR with cable-length control and anchor point displacements. The filled circles are the true anchor points, and the unfilled circles are the ideal anchor points. The bold line is the actual trajectory, and the thin line is the reference. (a) Side and (b) top views.

The three methods described above are now tested in simulation. The same configuration and reference trajectory from Figure 84 is used. However, the anchor points are displaced from their ideal locations by random amounts up to 5 cm in the *x*, *y*, and *z* directions. An initial simulation is run, where the reference cable lengths are calculated using (67). That is, feedback from the position measurement is not used and the anchor points are assumed to be in their ideal locations. The result of this simulation is shown in Figure 86. If the anchor points were in their ideal locations, the mass would closely follow the reference trajectory, similar to Figure 70. Instead, the errors in the anchor points produce large errors in the trajectory. The cables are labeled *a* to *c*, and their anchor points are plotted. The filled circles are the anchor points after the random displacements, and the unfilled circles are the ideal anchor points. Every anchor point is displaced in 3-D, which represents a worst-case-scenario. There are twelve unknown displacements.

Now, the data from the simulation in Figure 86 are used to estimate the anchor-point displacements. Using the first method, where (74) is assumed to be a linear equation with four unknowns, the anchor point displacements are estimated as shown in Figure 87. In this plot, the true anchor-point displacements for each cable are plotted as circles. The estimates, which were calculated using the data from four different time instances, are solid points. To test the effect of measurement noise, random values up to 2 mm were generated and added to the mass-position and cable-length data at the four time instances. The anchor-point displacement calculation was repeated for this case and plotted using an "×" symbol. The estimates without noise are generally close to the true values, with some exceptions. With noise, the estimates are further from the true values. Some of the

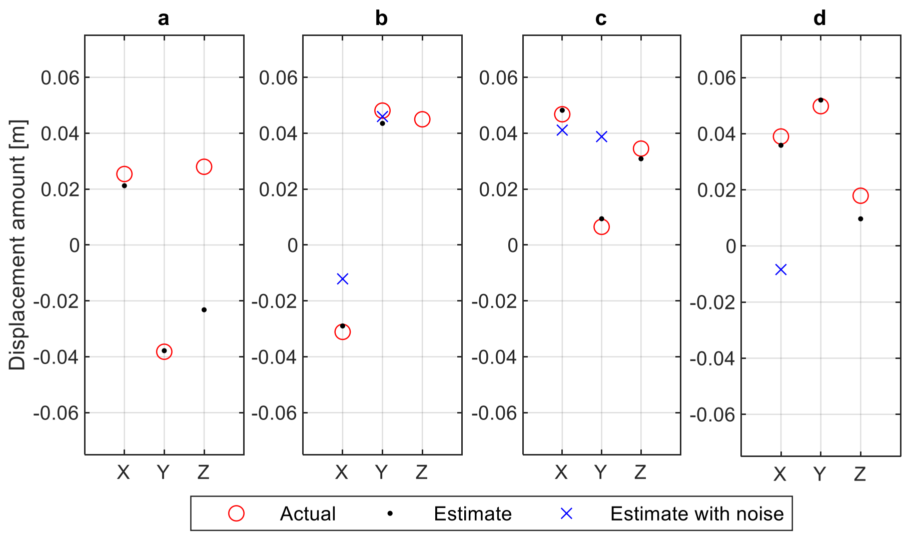


Figure 87. Estimated anchor-point displacement errors using the first method, where (74) is assumed to be linear with four unknowns.

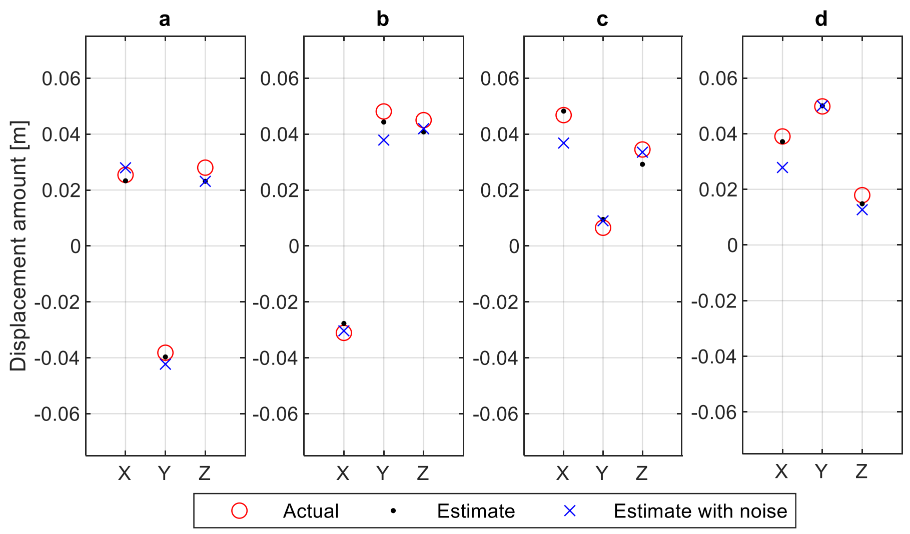


Figure 88. Estimated anchor-point displacement errors using the second method, where the optimization method (75) is used to estimate the three unknowns in (74).

estimated values with noise do not appear within the plot range, meaning that that they are fairly inaccurate. One disadvantage of this method is that the bounds for the estimates are not explicitly set, allowing for estimates that are obviously inaccurate.

The displacement estimates were repeated using the second method, as presented in (75). The results of this calculation are plotted in Figure 88. These estimated values, with and without measurement noise, are closer to the true values than the estimates seen in Figure 87. Based on this, the second method is preferred. The estimates without the noise are more accurate since any measurement noise is not accounted for in (74).

Finally, the results of the third estimation method, (76), are plotted in Figure 89. These results are similar to those in Figure 88. Specifically, the estimates with and without measurement noise are near to the true values. A close observation reveals that methods (75) and (76) are minimizing the norm of the same vector. That vector is

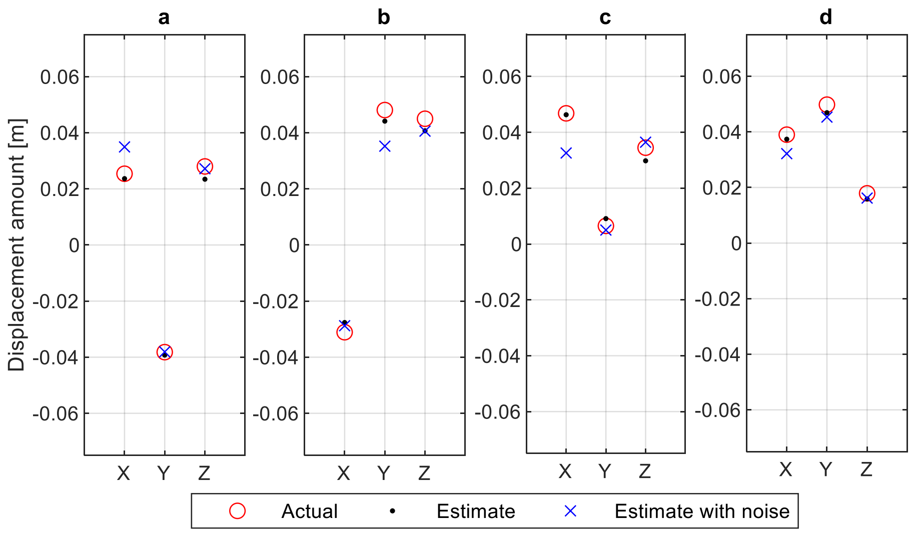
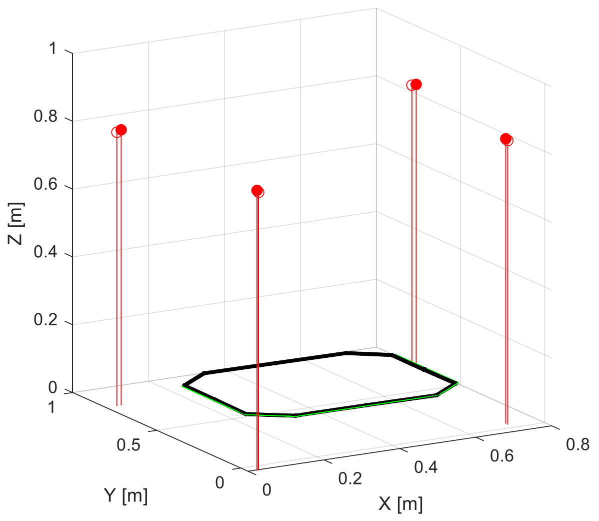


Figure 89. Estimated anchor-point displacement errors using the second method, where the optimization method (76) is used to estimate the three unknowns in (74).

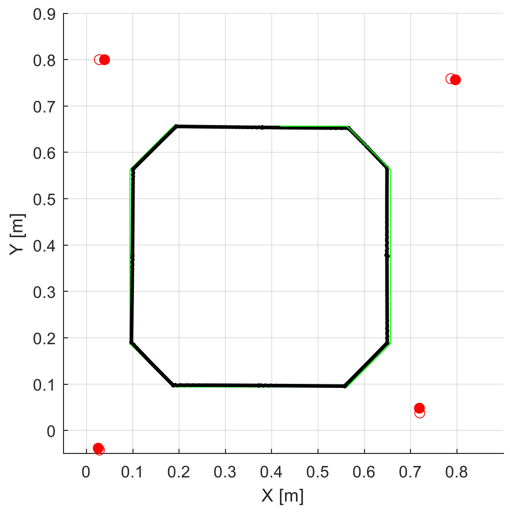
$$\begin{bmatrix} \left(\sqrt{(x_i+\delta x-x_m)^2+\left(y_i+\delta y-y_m\right)^2+(z_i+\delta z-z_m)^2}-l'_{im}\right)_{t=t_1} \\ \left(\sqrt{(x_i+\delta x-x_m)^2+\left(y_i+\delta y-y_m\right)^2+(z_i+\delta z-z_m)^2}-l'_{im}\right)_{t=t_2} \\ \left(\sqrt{(x_i+\delta x-x_m)^2+\left(y_i+\delta y-y_m\right)^2+(z_i+\delta z-z_m)^2}-l'_{im}\right)_{t=t_3} \end{bmatrix}. \qquad (77)$$

The difference is that (75) minimizes the 2-norm of the vector whereas (76) minimizes the 1-norm. Either method could be used, or both could be used and the results compared.

To demonstrate the usefulness of these methods, the CSR simulation was repeated. This time, the estimated anchor-point displacement values from Figure 88, corresponding to the case with measurement noise, were used to calculate the reference cable lengths. In other words, the location of the anchor points was updated based on the estimated results. The resulting simulation is shown in Figure 90. Clearly, the estimated anchor-point



(a)



(b)

Figure 90. Simulation result for a four-cable CSR with cable-length control and estimated anchor point displacements. The filled circles are the true anchor points, and the unfilled circles are the estimated anchor points using method (75). The bold line is the actual trajectory, and the thin line is the reference. (a) Side and (b) top views.

locations, even with the measurement noise, are close enough to the true locations that the resulting trajectory now closely follows the reference trajectory. The results are even better for the estimated values without measurement noise.

In a real application of a CSR, one or more initial calibration trials could be run in order to estimate the true cable anchor points. A special calibration trajectory, such as that in Figure 84, can be used for this purpose.

The anchor-point position estimation methods presented in this section, along with the position-feedback results from Section 3.2.5.2, are expected to be effective strategies for addressing some imperfections in a CSR such as cable-length errors or displacements of the cable anchor points from their idea locations. The same methods are tested in the forthcoming experiments in Chapter 5.

4. EXPERIMENTAL PROTOTYPE

In order to support the analyses from Chapter 3, a laboratory-scale prototype of a four-cable CSR was created and operated. The complete prototype is pictured in Figure 91. The components of this prototype are presented in this chapter. The experimental results are discussed in Chapter 5.

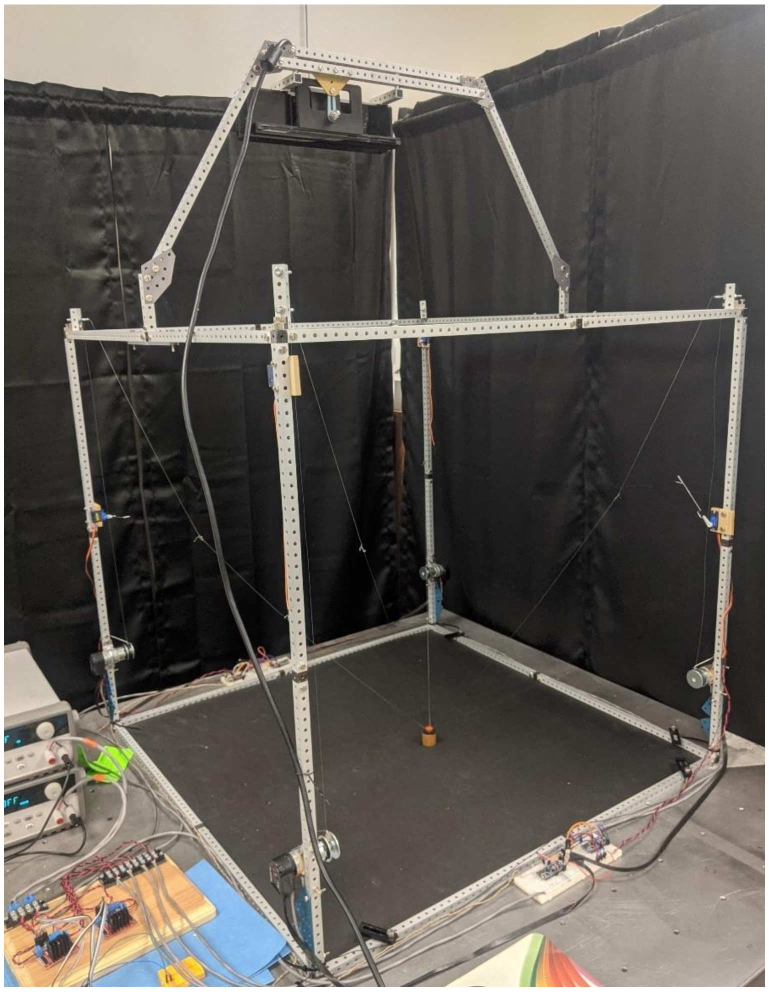


Figure 91. Complete experimental prototype of a four-cable CSR with an external position sensor.

4.1. Robot Frame

The first major component of the prototype is the robot frame that is used to set the robot configuration. In Figure 22, it was suggested that individual towers be used to anchor each cable pulley. Then the cables would run from the motors, up to the pulleys, and back down to the end effector. However, any movement or flexibility of the towers moves the pulley locations and alters the robot configuration. For the experiment, it was important to maintain a fixed position for the cable anchor points. Therefore, the robot frame was constructed using a cubic structure.

To construct the robot frame, aluminum bars and joints, like those pictured in Figure 92, were used. The bars are hollow, with a square cross-section of 0.5 in, and they have 5/32-in diameter holes spaced at 0.5 in along every side. These holes serve as standard mounting points for other components. Some of the components are pictured.

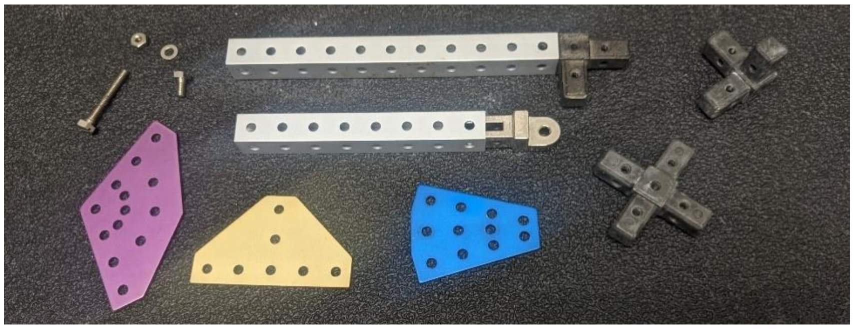


Figure 92. Aluminum bars and links used to construct the robot frame.

Aluminum bars and joints of various lengths were joined to form a rectangular prism, as shown in Figure 93. The prism had the outer dimensions 32×32×30.5 in. The

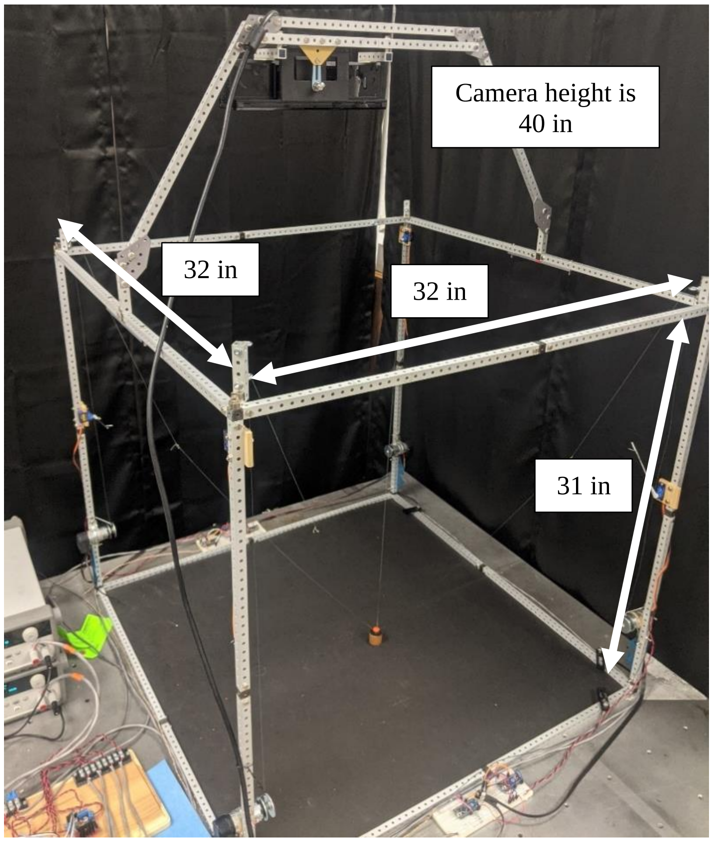


Figure 93. The basic frame used for the CSR prototype experiment. The frame was constructed using aluminum bars of various lengths.

frame also had an arch attached across two of the top bars. This arch served as the mounting point for the camera sensor that is discussed in Section 4.5.

The robot frame was firmly secured to the top of an optical table using mounting screws. Sitting below the frame was a black construction paper that marked the ground level of the CSR system. The black paper was important because it provided a uniform background to contrast with the end-effector.

When assembled, the robot frame was rigid and resistant to movement. This was important because the frame maintained the robot configuration throughout the experiment, and any significant vibrations would be difficult to quantify. The next sections detail several components that were attached to the robot frame.

4.2. Cable System

One of the main purposes of the robot frame is to create the anchor points for the four CSR cables. To provide for this function, four short aluminum bars and small eye hooks were attached to the top corners of the frame, as seen in Figure 94. When the position of the eyehooks is taken into account, the final cable anchor points are at the height of 0.79 m and arranged in a square with side lengths of 0.74 m.

For cables, a braided fishing line was used. The specific line used was the Power Pro Spectra 65-lb line with 0.4-mm dimeter. This line is light, low in friction, and fairly inextensible. To test the cable stretching properties, the test mass of 1 kg was suspended

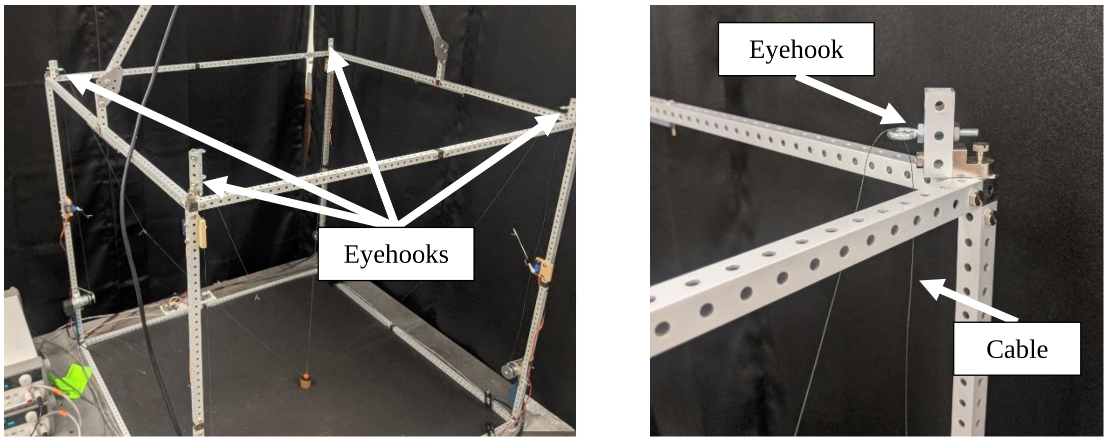


Figure 94. Eye hooks that serve as the mounting points for the CSR cables to pass through. These eye hooks effectively set the cable anchor point locations.

from a 1-m section of the braided line. The stretch amount for this condition was about 1.5 mm or 0.15% of the cable length. Since the 1-kg load is more than the load during the CSR operation, and since the cable lengths were always shorter than 1 m, the stretch of the braided line was assumed to be negligible at all times. Note that even if the stretch amount was 2 mm the equivalent linear stiffness of the 1-m cable would be almost 5 kN/m.

The test mass used to represent the end effector was a 0.5-kg brass dead weight, pictured in Figure 95. Four 0.5-m cables were securely tied to the weight at one end and a small washer at their other end. Then, another cable of about 1 m was tied to each of the washers. These longer lengths of cable were then passed through the eye hooks and attached to the system actuators.

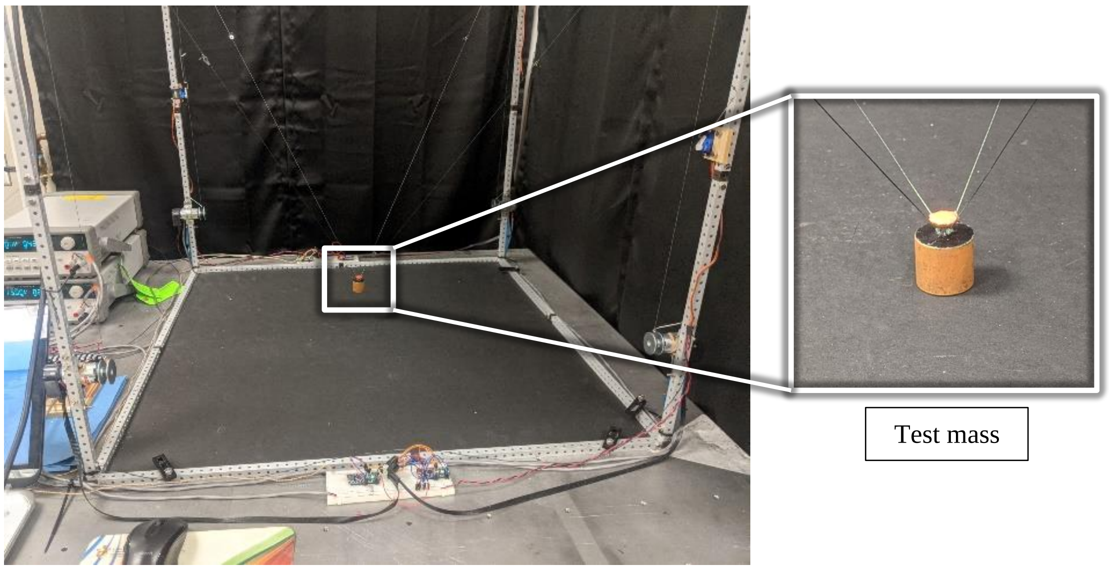


Figure 95. Four cables are tied to the 0.5-kg test mass of the experiment. The cables run through the eye hooks of the robot frame.

The purpose of the small washers was to produce known starting cable lengths for every experiment trial. The washers were large enough that they could not be pulled through the eye hooks, but they were not large enough to add a significant mass to the cables. At the start of each trial, the cables were pulled by the actuators until the washers were stopped by the eye hooks. At that point, the effective cable lengths were known to be 0.5 m, and the experiment could begin.

4.3. DC Motors

After passing through the eyehooks in Figure 94, the cables ran down to the DC motors, pictured in Figure 96. The motor shafts were attached to spools of 1-cm diameter and equipped with high-precision encoders. The motors used were the Maxon A-max 110950, and their electromechanical characteristics are listed in Table 2. The encoders, equipped from the factory, were the HEDS 5540. These quadrature encoders use optical sensors to produce 1024 PPR. Therefore, the encoders can be used to measure cable-length

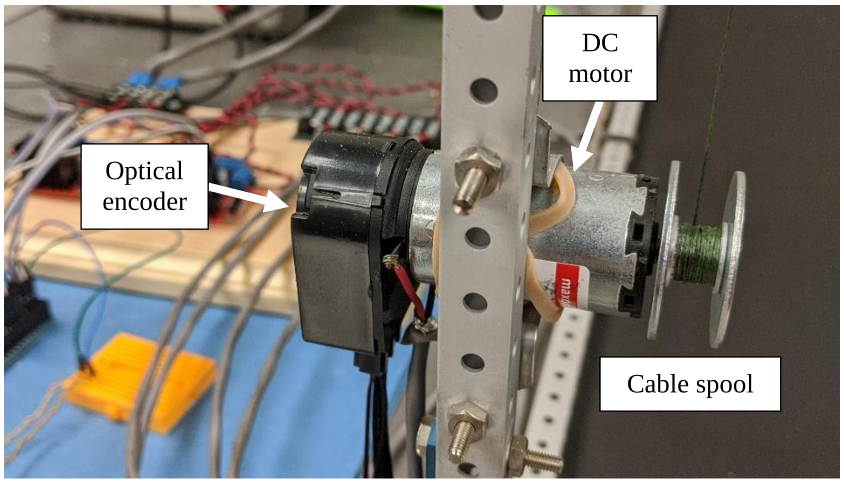


Figure 96. DC motors and cable spools used to wind the excess cable lengths. The DC motors are equipped with high-precision quadrature encoders.

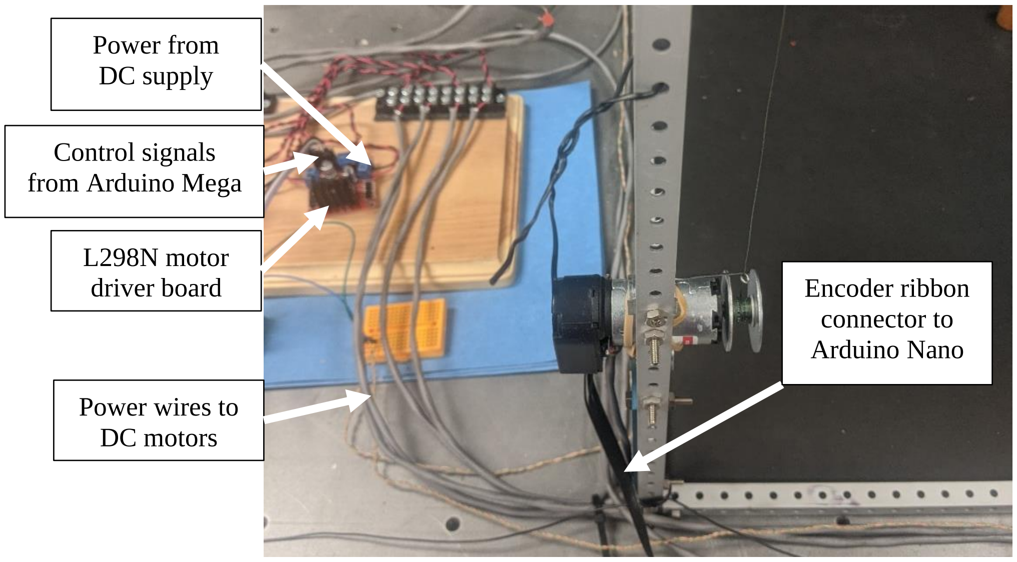

Power from DC supply
Control signals from Arduino Mega
L298N motor driver board
Power wires to DC motors
Encoder ribbon connector to Arduino Nano

Figure 97. Wiring for the DC motors and encoders.

changes as small as 16.5 μm. This allowed for the implementation of precise servo loops for each cable length. Example results of the servo loops are presented in Section 5.1.

The DC motors were powered using an Agilent E3644A DC power supply operating at 15 V, and the encoders were powered using a separate supply at 5 V. The wiring for the encoders and motors is detailed in Figure 97. To control the effective power input to the motors, L298N driver boards were placed between the power supply and the motors. These motor drivers were controlled using an Arduino microcontroller board.

4.4. Microcontrollers

A total of six microcontrollers were used in the experimental prototype. The central microcontroller was the Arduino Mega 2560 board. This board implemented the main program used in the CSR operation. Four additional Arduino Nano boards were used

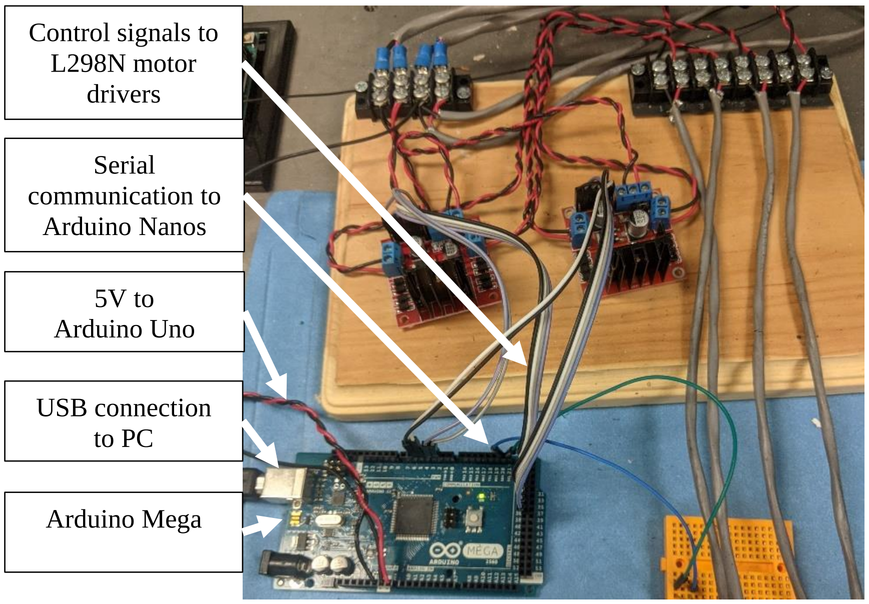

Control signals to L298N motor drivers
Serial communication to Arduino Nanos
5V to Arduino Uno
USB connection to PC
Arduino Mega

Figure 98. Wiring for the Arduino Mega 2560 microcontroller board. This controller ran the main program of the CSR.

to count the encoder pulses. The sixth microcontroller was an Arduino UNO board that was used to create a PWM signal and control some small servo motors. All of the Arduino codes are in Appendix B.

The main CSR program ran on the Arduino Mega 2560 and implemented all of the calculations and control loops included in Figure 68, except for the encoder counting and position-sensor processing. The complete wiring of the Arduino Mega is detailed in Figure 98, and all of the functions fulfilled by this board are listed in Table 3.

The four Arduino Nano boards were used to keep the count of the encoder pulses, and then, communicate the updated counts to the Arduino Mega upon request. The communication was accomplished using the i2C serial-communication protocol. One Arduino Nano was used for each encoder. The wiring for one of these Arduinos is shown

Table 3. List of functions fulfilled by the Arduino Mega 2560 microcontroller boards.

Function	Description	External Connections
Timing	Keep track of time in the program. Enforce time delays to achieve sampling periods.	
Reference trajectory	Calculate the reference trajectory in the Cartesian coordinates using some mathematical equations.	
Reference cable lengths	Convert the reference trajectory to reference cable lengths using the robot kinematic equations.	
Measured cable lengths	Periodically update the measured cable lengths based on the encoder counts.	i2C communication with the four Arduino Nanos
Camera-measured position	Read and interpret the camera data regarding the measured position of the test mass.	Serial communication with the PC via the USB connection
Position-feedback	Implement the outer-loop controller including the camera-measured position to update the reference cable lengths.	
Servo controller	Implement the PID law that is used to control the cable lengths.	
Motor PWM	Generate the command signals for the motor drivers.	Digital signal to the L298N motor driver boards
Debugging	Display some diagnostic data upon request. Provide a convenient structure for enabling/disabling various functions in the code.	USB communication with the PC

in Figure 99, and the wiring for the other Arduinos is the same.

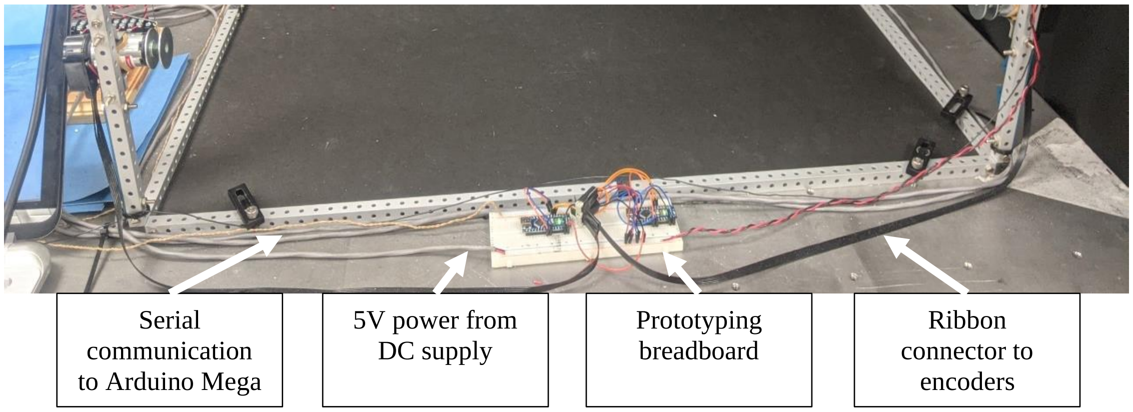

Serial communication to Arduino Mega
5V power from DC supply
Prototyping breadboard
Ribbon connector to encoders

Figure 99. The wiring for the Arduino Nano microcontroller boards. These boards are used to count the encoder pules. Only two Arduino Nanos are shown here, but the wiring is the same for all four Arduino Nanos.

Initially, the encoder-counting task was attempted using a single Arduino board, but the counts were inaccurate due to missed encoder pulses. It is likely that the digital signals from the encoders interfered with each other, such that one Arduino could not accurately process all of the counts. Therefore, a dedicated Arduino Nano was used for each encoder. Furthermore, only the rising edge from one channel of each encoder was used for detecting the encoder pulses. This means that the maximum resolution of the encoders was not utilized. The final sensing resolution of 66 μm per encoder count was acceptable and, ultimately, allowed for precise control of the cable lengths.

The last microcontroller that was used in the experiment was an Arduino Uno board. This board is smaller than the Arduino Mega and has a slower clock speed.

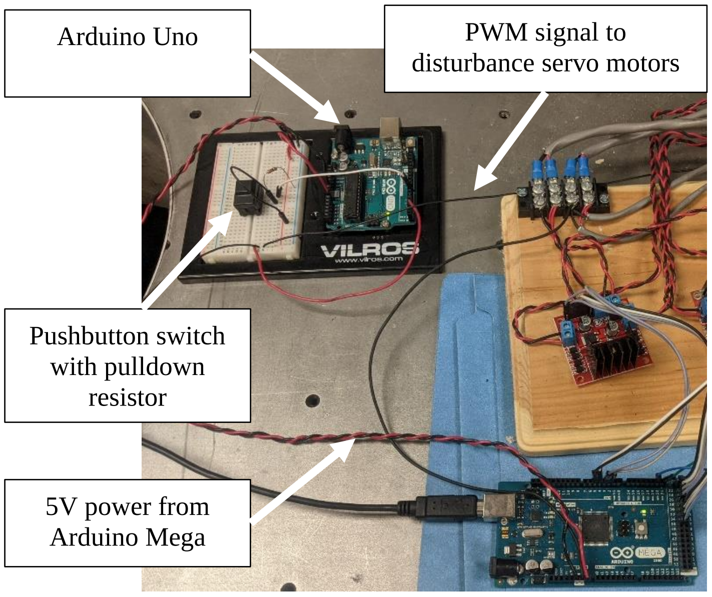
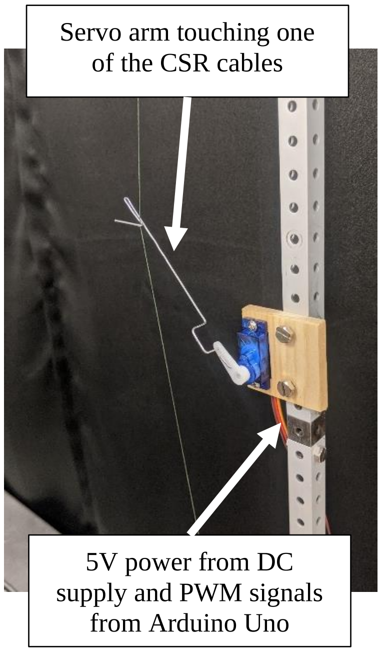

Arduino Uno
PWM signal to disturbance servo motors
Pushbutton switch with pulldown resistor
5V power from Arduino Mega

Servo arm touching one of the CSR cables
5V power from DC supply and PWM signals from Arduino Uno

Figure 100. The wiring of the Arduino Uno microcontroller board. This board was used to control the movement of two small servo motors in some of the experiment trials.

However, the board was only used to generate a PWM signal that swept between a minimum and maximum duty cycle. The PWM signal controlled two small servo motors that disturbed the cables in one set of experiment trials. The wiring for this microcontroller is detailed in Figure 100. The only external wiring for this microcontroller was a single pushbutton that was used to initiate the sweeping PWM signal.

4.5. Kinect Camera

To experimentally support the position-feedback control law (70) it was necessary to include some independent sensor that could measure the position of the test mass. For this task, a Kinect v2 camera was used. This sensor was created by Microsoft for use with a popular gaming system. The sensor contains a color camera, infrared camera, and microphone. The placement of the Kinect sensor in the prototype is show in in Figure 101.

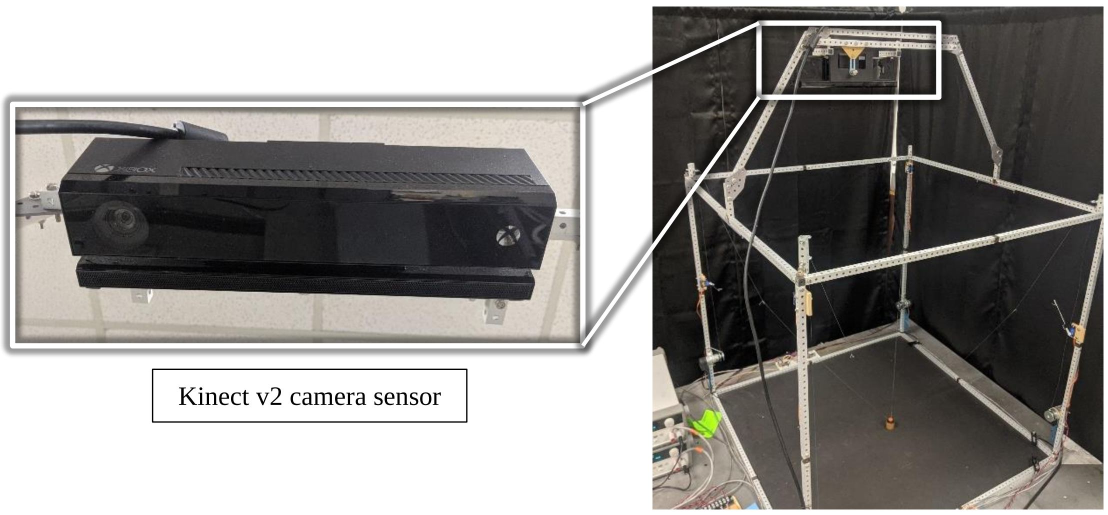


Figure 101. The Kinect v2 camera, a camera and infrared-sensor package, was designed for use with a popular gaming system. This sensor was used in the experimental prototype to measure the position of the test mass.

Microsoft provides the cable adapters and drivers necessary for using the Kinect v2 sensor with a PC instead of the gaming system. Furthermore, there are some open-source libraries that facilitate the use of the Kinect functions. In this experiment, the PyKinect2 Python library by Vlad Kolesnikov was used to operate the Kinect sensor via a Python script. The full script is provided in Appendix B.

The color and infrared images from the Kinect sensor are used to measure the 3-D position of the test mass in the CSR. This is accomplished by repeatedly collecting the image frames from the Kinect sensor and, then, employing some computer-vision functions that are available through the open-source Python library OpenCV. The full algorithm is described in Table 4.

Table 4. Algorithm used to measure the 3-D position of the mass using the Kinect sensor.

For every color (or depth) frame captured by the Kinect sensor, where the frame data is stored in one or more 2-D arrays of the appropriate dimension.

Step 1. Crop the frame according to a pre-determined crop area. This is to cut off the area of the frame that is outside of the CSR work area. Called it the cropped frame.
Step 2. In the cropped frame, extract a small focus area. Call this the focus area. Only the focus area will be further processed. The focus area is known to contain the mass because the area is based on the processing results from the last frame. For the first frame, or for any time that the mass position is lost, the focus area is expanded to the entire cropped frame.
Step 3. Filter the focus area by HSV value (for color frames) or by depth values (for depth frames). After filtering, the frame becomes a binary image. That is, the areas of the image that passed the filter are filled with a value of one and all other areas are filled with a value of zero. Call it the filtered image.
Step 4. Apply a polygon-finding algorithm on the filtered image. Keep only the largest polygon that matches the shape of the mass. In this dissertation, the matching polygon is a polygon with more than four sides. If necessary, the polygon should be a minimum size, in pixels. The minimum size can be determined experimentally. Once the desired polygon is found, fill the polygon area with ones. Make all other areas of the polygon have a value of zero.

Step 5. Using a moment-calculation algorithm, calculate the center of area for the polygon found in Step 4. Note the pixel position of the polygon in the filtered image.
Step 6. Calculate the pixel position of the polygon center with respect to the cropped frame from Step 2.
Step 7. Using the position found in Step 6, calculate the true-world coordinates of the mass. This can be done by comparing the pixel position to a pre-determined set of calibration data.
Step 8. Save the current pixel position of the mass. Return to Step 1 to process the next frame.

One of the major requirements for the computer-vision algorithm to work properly is for the test mass to be easily distinguished from other objects in the images. For this reason, a black construction paper was placed on the floor of the robot, as seen in Figure 93, and a bright tape was placed on the top of the test mass, as seen in Figure 95. Furthermore, the space above the test mass was always kept unobstructed. The cables were thin enough that they did not interfere with the view of the camera.

4.6. Personal Computer

The computer-vision algorithm presented in Section 4.5 required significant data processing and compatibility with the Kinect v2 drivers. Therefore, a PC was used along with the Kinect sensor. The PC, pictured in Figure 102, was the Satellite S55t-B from Toshiba with the Windows 10 operating system. This model had the Intel Core i7-4710HQ processor operating at 2.50 GHz. Any comparable PC could be used to obtain similar results.

The algorithm from Table 4 was implemented in a Python script and ran on the PC at the maximum speed. The rate of capture for the images was about 30 frames per second (FPS). This maximum speed was governed by the Kinect hardware. At the end of the

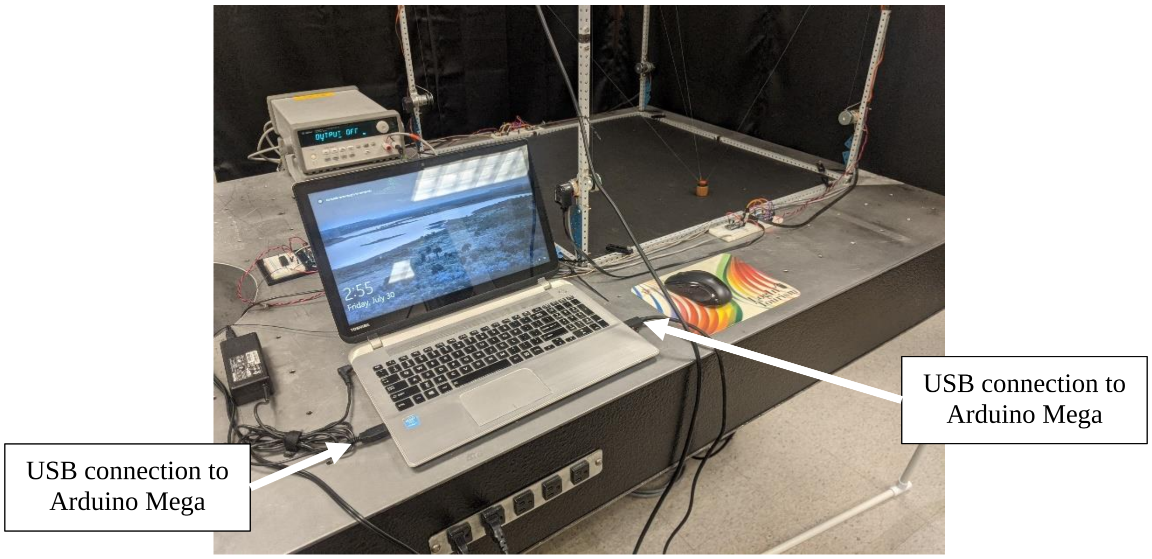

USB connection to Arduino Mega
USB connection to Arduino Mega

Figure 102. The laptop PC used for real-time communication with the Kinect, running the computer-vision algorithm, and data communication to the Arduino Mega.

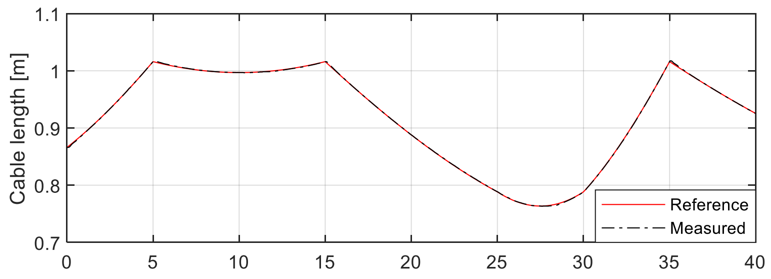
algorithm, the position data calculated in the script was communicated to the Arduino Mega via the USB connection.

5. EXPERIMENTAL RESULTS

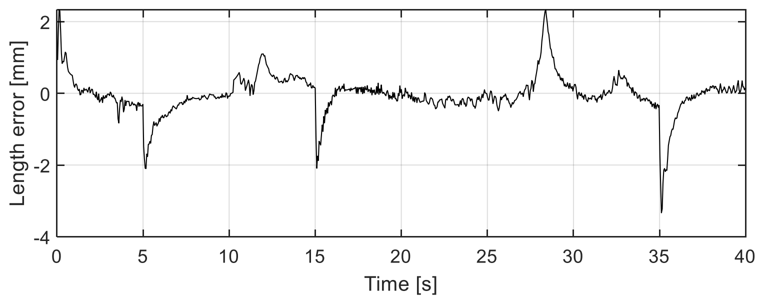
A series of experiments were performed to support the simulation results from Chapter 3. In particular, the cable-length control with position feedback was tested, and some errors were manually introduced into the experimental CSR system to study the effects on the mass trajectory. Furthermore, the anchor-point position estimation method from Section 3.2.6 was demonstrated. The tension-based methods, such as the cable-tension control and the system stiffness calculations, were not included in this set of experiments due to the lack of force sensors in the experimental setup.

5.1. DC Motor Servo Loops

The cable-length control in the experiment is dependent on the adequate performance of the DC-motor servo loops, so the first experimental results demonstrate the reference-tracking performance with respect to the cable lengths. Recall that the cable lengths are controlled by turning the DC motors that, in turn, wind and unwind the cables. The reference lengths are calculated according to the desired movement of the CSR and should be followed closely by the lengths measured with the motor encoders. The same PID controller that was presented in Section 3.2.5.1 was implemented in the Arduino code with a time period of 15 ms. The gains were $k_P = 10000$, $k_I = 20000$, and $k_D = 300$ were selected via experimentation. When tuning the controller gains, a fast response with smooth motor movements was sought. Furthermore, the response was tested using various test masses to ensure that the servo response was stable for high or low cable tensions, like in the simulation of Section 3.2.5.1. Figure 103 shows the length of one cable during one



(a)



(b)

Figure 103. Experiment results of the DC motor servo loops. The reference cable lengths are calculate based on the desired movement of the CSR, and the actual cable lengths are measured using the motor encoders. (a) Cable length and (b) length error.

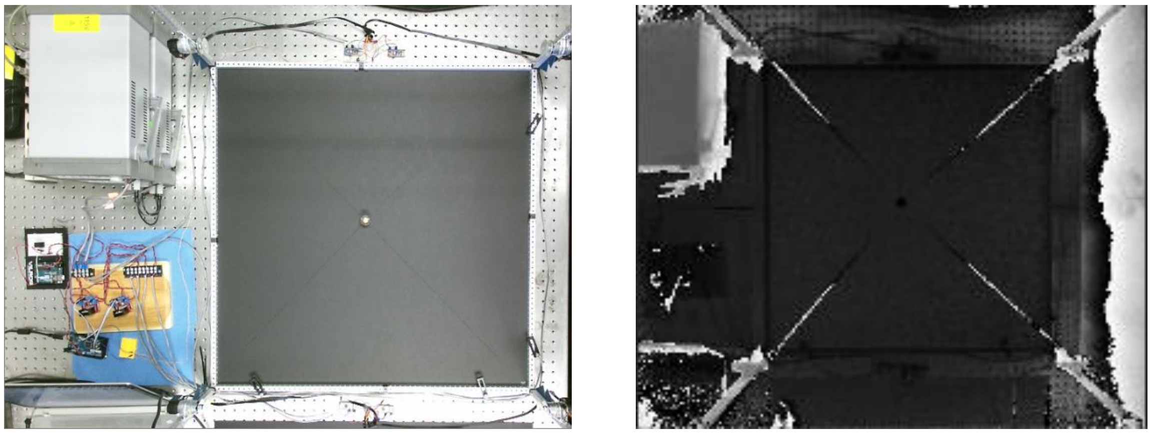
of the experiment trials. In this trial, the CSR mass was moved in a square trajectory and at a constant height. This movement requires the cable lengths to vary by about 0.2 m. The measured cable length during this movement closely tracks the reference length, such that the two curves cannot be distinguished in Figure 103(a). The length error is plotted in Figure 103(b) using a millimeter scale. The maximum length error in the trial was about 3 mm, and the error typically stayed well within 1 mm. The error increased when there were sudden changes in the reference cable lengths. After these increases, it takes about two seconds for the error to approach zero again.

With this DC servo loop performance, the cable lengths can be accurately controlled within a few millimeters throughout a trial. Considering the high stiffness of the cables and the possibility of an inexact spool radius, especially due to the layering of the cables as the spool winds up, the four-cable system can be expected to position the mass with a precision of several millimeters, but not better than that. Furthermore, redundant cables could exhibit a small slack at certain times of the experiment due to their lengths being slightly longer than required. This slack should be small and not detrimental to the performance since the light cables do not provide a significant tension when sagging.

5.2. Camera Position Tracking

The second component to be tested was the camera system. The camera provides the 3-D position feedback that enables the CSR to react to positioning errors in real time. As such, it is important to ascertain how precise and accurate the camera can be.

Figure 104 depicts some of the original frames captured by the camera in the CSR system. The image on the left is from the color sensor, and the image on the right is from the depth sensor. The depth gray scales do not have any practical meaning, but the data in

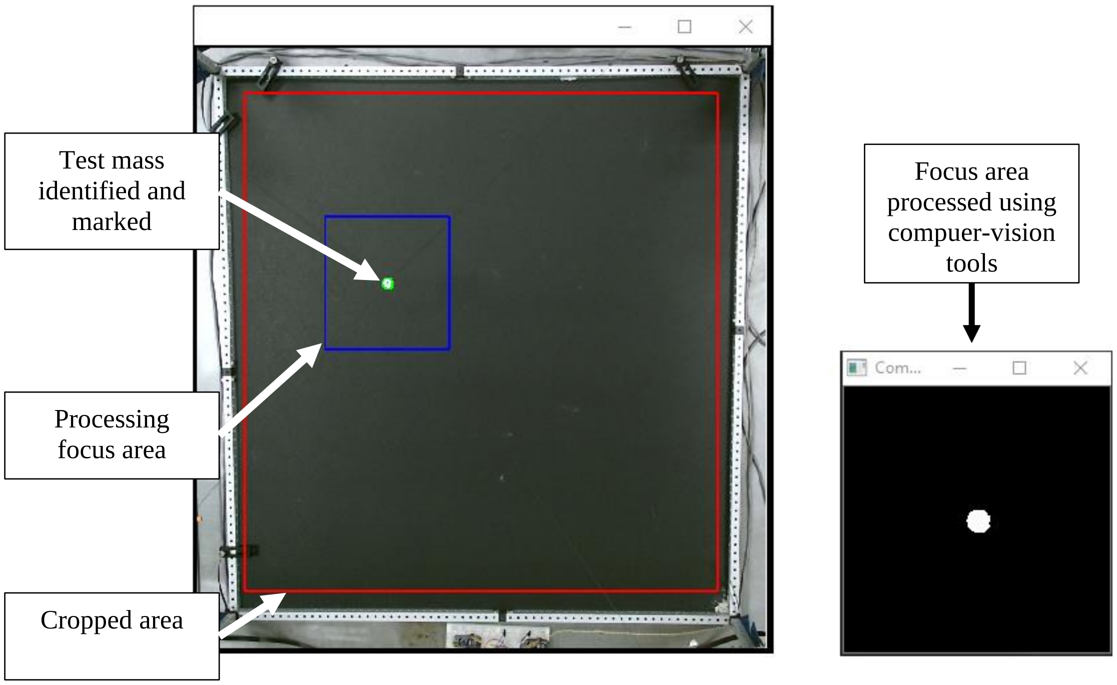


(a) (b)

Figure 104. Original images from the camera sensor. (a) A color image with 1080 × 1920 px resolution. (b) A depth image with 424 × 512 px resolution.

the corresponding image matrix has the information about the distance from the sensor to the objects in the image. Centered in the frame is the ground of the CSR, and the cable-suspended mass is visible in bright orange. Around the ground are some of the instruments of the experiment. Nothing ever obstructs the view between the camera and the end effector. At the start of every computer-vision-program loop, the camera produces such images in matrix form to the experiment PC.

Figure 105 gives a visual representation of the computer-vision algorithm. The top-left image is the small focus area that was clipped from the color image in the first steps of the algorithm. This focus area was then filtered by color values, such that only the



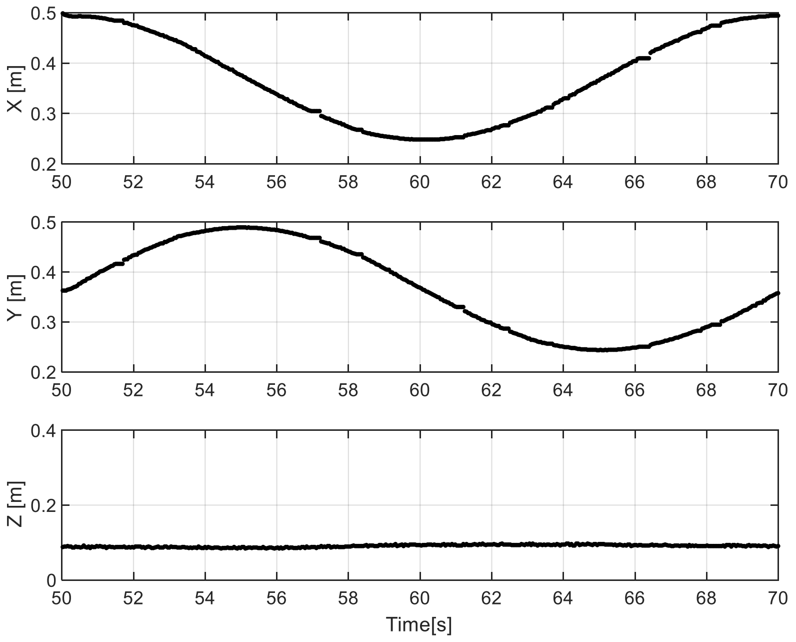
Test mass identified and marked
Processing focus area
Cropped area
Focus area processed using compuer-vision tools
Com...

Figure 105. Visual results of the computer vision image-processing algorithm as applied to the color image. The same steps are applied, separately, to the depth image. First, a working area is cropped from the full image. Then, a focus area containing the last-known location of the mass is processed using computer-vision tools. Finally, the position of the test mass is calculated and marked within the original color image.

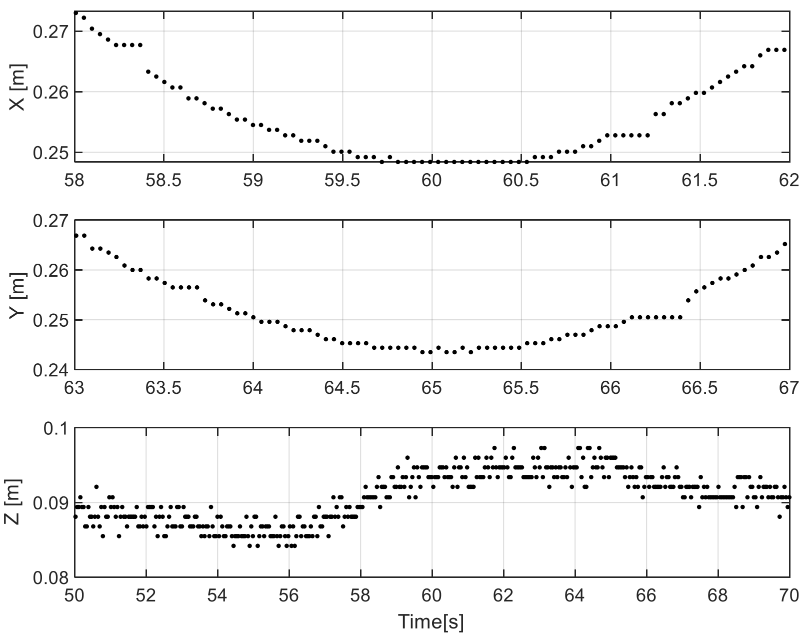
mass remained in white after the filtering. The program then calculated the location of the mass based on the white shape. The calculated position is highlighted in the processed color image and the position data was saved as a pixel location. Figure 105(b) is another small focus area that was clipped from the depth frame. Again, the mass was identified within the focus area, and the mass location was found. In a live video feed of the processed images, one can observe the focus windows and mass position update at 30 FPS.

The pixel values calculated by the computer-vision algorithm must ultimately be converted to the world-frame coordinates. In order to do this, a coordinate frame was established on the CSR ground. Using a ruler, a square grid of nine equally-spaced points were marked on the ground. The spacing between the points was 20 cm, and the middle point was centered with respect to the CSR frame. These nine points served as reference positions for calibration of the camera data. Upon placing the test mass on each of these points on the ground, the corresponding pixel values given by the computer-vision algorithm were recorded. Then, this data set was used to implement a linear-interpolation function in the camera program. The same procedure was repeated for the depth-image data, using reference heights between 5 and 15 cm. The range of heights was intentionally kept small to avoid any distortion effects from the camera lens. Preliminary experimental results showed that the linear interpolation would not give accurate results for the entire operating space of the system. Therefore, the decision was made to run all trials within the 5 to 15-cm-height range.

The final output of the camera algorithm is the 3-D position of the test mass in meters. A sample of the data is shown in Figure 106, where the test mass was moved in



(a)



(b)

Figure 106. An example of the data output from the camera sensor and computer-vision algorithm used to measure the real-time position of the test mass in the CSR experiment. (a) The position of the test mass and (b) details of the position for various time ranges.

a circle trajectory at a constant height of 10 cm. As expected, the *x* and *y* positions follow sinusoidal trends, and the *z* maintains a nearly constant height. The detail view in Figure 106(b) suggests that the *x-y* measurement resolution, defined as the smallest change in the data that can reasonably be detected from a visual analysis of the plot, is about 1 mm and the *z* resolution is about 2 to 3 mm.

Another important detail about the data is the elapsed time between consecutive points. This determines the effective sampling rate of the camera and algorithm as a whole. Figure 107 shows the time between every two consecutive image frames in a sequence of 400 frames. The time period holds steady at about 45 ms, save for a few occasional points. Despite the maximum framerate of the camera being 30 FPS, the extra time required to process the image frames and to save the data adds about 10 ms of processing time per frame. Recall that the time period for the DC motor servo loops was 15 ms. To account for the slower sampling period of the camera system, the camera data were sampled every third loop of the servo program. This dual-sampling-period scheme produced smooth control of the cable lengths and utilized the maximum sampling rate of the camera.

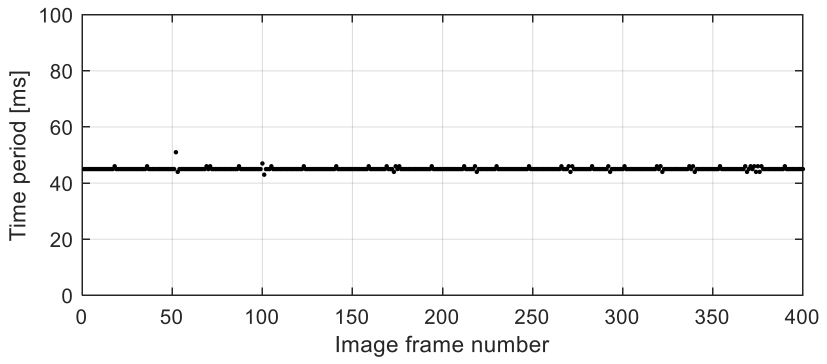


Figure 107. Typical sampling period for the camera for an experimental trial.

5.3. Nominal Performance

With the experimental CSR assembled, the DC motor servo loops tested, and the camera sensor implemented, the cable-length control of the system could begin. The first test of the complete system explored the nominal performance. This means that the operating conditions of the robot were as perfect as possible, without any deliberate cable-length errors, cable anchor-point errors, or external disturbances. Also, this test was carried out without incorporating the position feedback from the camera sensor.

The trajectory for the nominal-performance test included a square and circle shape at a constant height. This trajectory tests the system's ability to follow the straight and curved paths typically seen in AM applications. The result of the nominal-performance experiment is plotted in Figure 108. The solid line is the reference path and the dots are

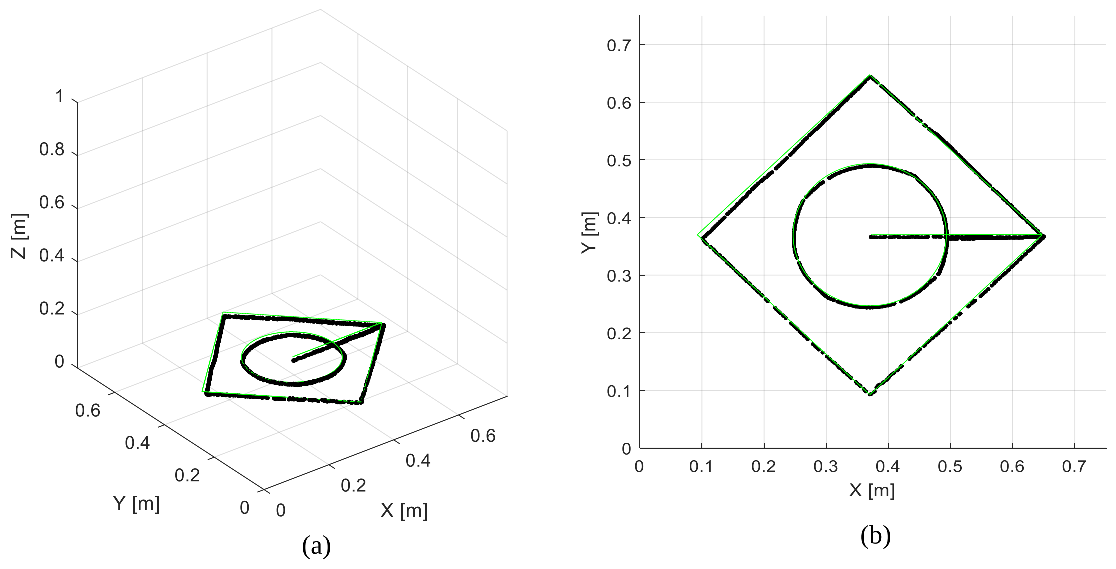


Figure 108. Nominal performance test of the experimental CSR system, without any deliberate system errors or external disturbances. The solid line is the reference trajectory and the solid dots are the position data recorded by the camera system. (a) Side and (b) top views.

the position data captured by the camera. Two views are shown to demonstrate that the test mass closely follows the reference line. This result is comparable to that in Figure 70.

The position data of the nominal test is plotted again in Figure 109. This plot shows the speed of the movement and highlights the small variations in the position of the test mass. The trajectory begins at 10 s, after the test mass has settled at its starting position. From 10 to 15 s, the mass moves to the right-most corner of the square shape. Then, the mass travels at a constant speed along the top two lines of the square, from 15 to 35 s. The bottom two lines of the square trajectory are traversed twice as fast, from 35 to 45 s. From 45 to 50 s, the mass moves to the start of the circle path. The final movement is the circle path at a constant linear speed, from 50 to 70 s. The square path appears as straight lines in the *x*-*y* plots, and the circle path appears as sinusoidal lines. The *z* plot shows that the

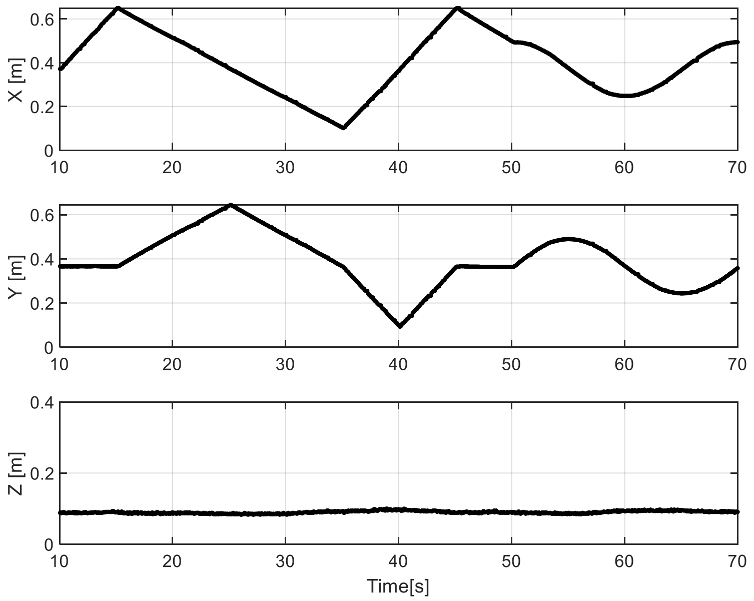


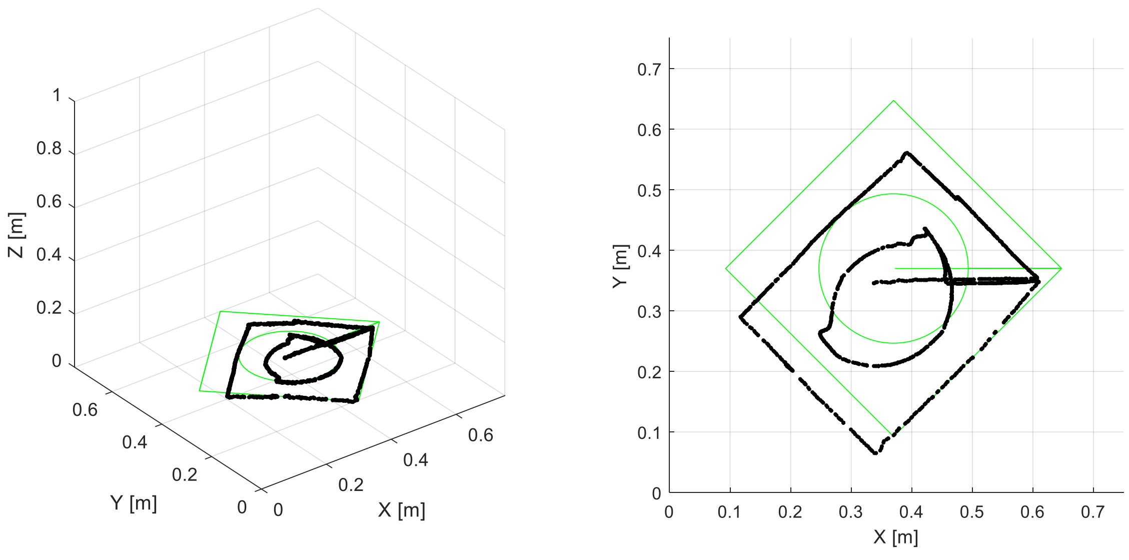
Figure 109. Position data for the nominal performance experiment plotted in Figure 108.

test mass maintains a constant height at 10 cm almost perfectly.

The nominal-test results represent a best-case scenario for the CSR. The test was performed after careful assembly of the experiment prototype in a controlled environment, and there were no significant disturbances in the laboratory environment. Even so, a small tracking error can be observed in the results, highlighting the system's sensitivity to small imperfections. As previously discussed, these imperfections can occur in many locations of the robot, like at the cable anchor locations, at the cable spools, or along the cable lengths. If the same robot configuration were implemented at a large scale, where the environment is not controlled, system imperfections could be even more prevalent.

5.4. Cable-Length Error

The first type of error that was explored in the experiment was inaccurate cable lengths. To do this, the same test trajectory as in Figure 108 was used, but errors were intentionally added to cable-length measurements in the Arduino code. The four cable lengths, which were measured by the DC-motor encoders, were altered by +5 cm, +2 cm, 0 cm, and −3 cm. These quantities were chosen arbitrarily but small enough so that the test mass did not touch the ground during the trial. The largest of these errors is about 7% of the CSR-frame width. One of the length errors was negative, meaning that the altered cable length is shorter than the measured length. Two of the other cable-length measurements were increased, and one cable length was unaltered. When a cable length is manually altered in this way, without the true cable length changing in the physical system, the servo controller operates with inaccurate values. As a result, the cable length will have a nearly constant tracking error, even though the servo controller performs its function adequately.



(a) (b)

Figure 110. Experiment results with cable-length errors and no real-time position feedback. (a) Side and (b) top views.

The experimental result with cable-length errors is shown in Figure 110. Here, the position measurements from the camera were not used in the feedback control to demonstrate the full effect of the uncorrected cable-length errors. The trajectory of the test mass is seriously affected, and the tackinging error is large. If this system were being used for AM, the resulting workpiece would surely be deformed and not able to serve its intended function.

In the next experiment, the feedback control law, (70), was activated. The control law incorporates the position error as measured by the camera. The integral gain used in this experiment was $k_I = 3.0$. This value was selected after repeated experimentation. Values ranging from $k_I = 1.0$ to $k_I = 16.0$ were tested, and the general observation was that a larger value produced a smaller error but was more likely to cause unstable vibrations. Furthermore, k_I values greater than about 5.0 marginally decreased the positioning error.

The experiment with the position-feedback control is plotted in Figure 111, using the same views as in Figure 110 for comparison. The clear effect of the real-time position feedback is to reduce the tacking error, such that the trajectory of the test mass resembles the nominal-performance experiment plotted in Figure 108. The only exceptions are some momentary deviations along the circle part of the motion and the initial error at the start of the motion. The error at the start of the motion is corrected after the integral term of the controller has had time to accumulate the persistent error and begins to noticeably affect the control action. The deviations along the circle path occur as one or more of the cables transitions through a state of momentary slack. As mentioned previously, cable slack is

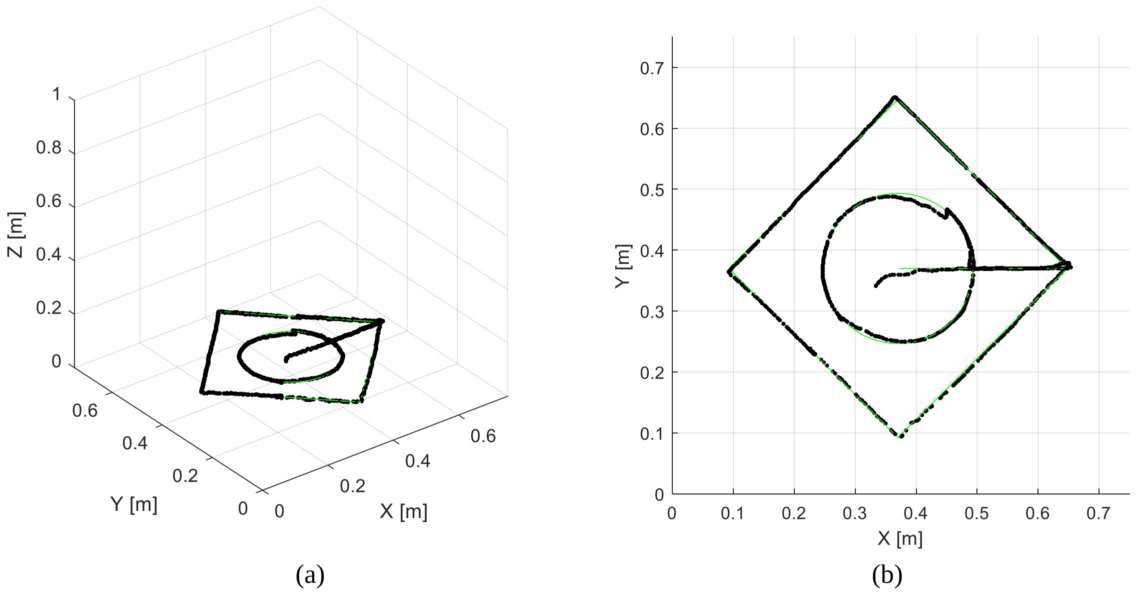


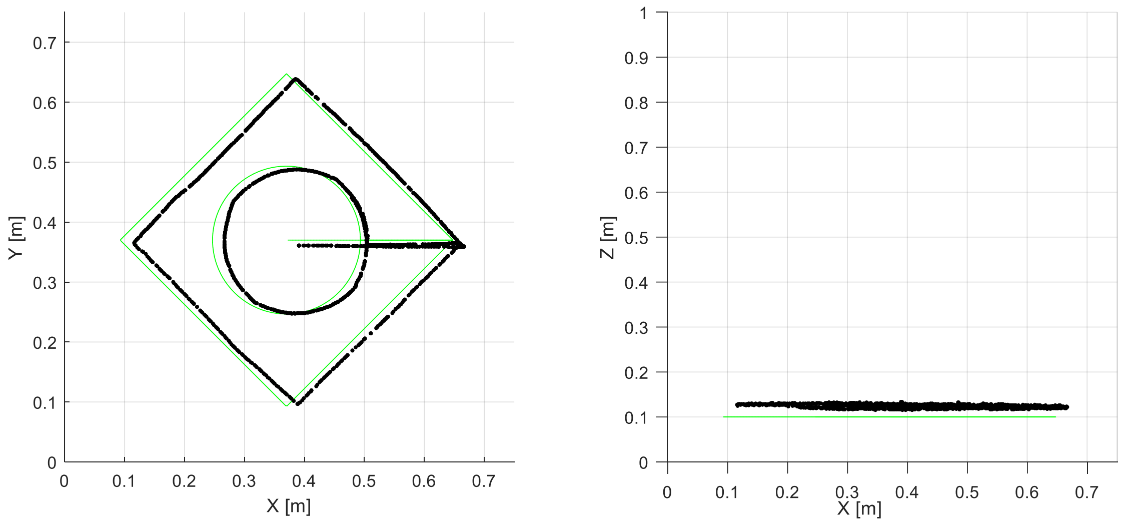
Figure 111. Experiment results with cable-length errors and including real-time position feedback. (a) Side and (b) top views.

one of the prevalent issues in a CSR with redundant cables. Nevertheless, the position error of the trajectory remains within a small error bound of 15 mm.

5.5. Camera Measurement Error

Until now, both the real-time position feedback and the plotted results of the mass movement were made possible by the use of the camera as an independent sensor. This begs the question, what if the camera data is inaccurate? Aside from the camera calibration used to process the image frames, how can one be confident that the true position of the mass indeed tracks the reference trajectory in an experiment like the one shown in Figure 111? This question can, in part, be addressed by studying the effect of camera-measurement errors on the closed-loop performance of the CSR.

Figure 112 shows the experimental result when the camera data is intentionally



(a) (b)

Figure 112. Experimental results with camera measurement errors and no real-time position feedback. (a) Top view showing the *x*-*y* position of the test mass as measured by the camera and (b) side view showing the measured height.

altered by a constant amount. The measured *x*, *y*, *z* values that were measured by the camera were changed by (2, −1, 3) cm. In this experiment, the real-time position feedback was not active. As a result, the true trajectory of the mass was something like that shown in Figure 108. However, due to the errors inserted in the camera data, the trajectory recorded by the camera appears shifted exactly by the error amounts. That is, the trajectory is shifted by (2, −1, 3) cm when compared to the reference. Without knowing about the error in the camera data, one might conclude that the mass did not track the reference. Thus, the camera should be well-calibrated before being used as an independent sensor.

Next, the experiment was repeated but, this time, activating the real-time position feedback using the same flawed camera data. The result of this experiment is plotted in Figure 113. This time, the mas closely followed the reference trajectory, giving a result

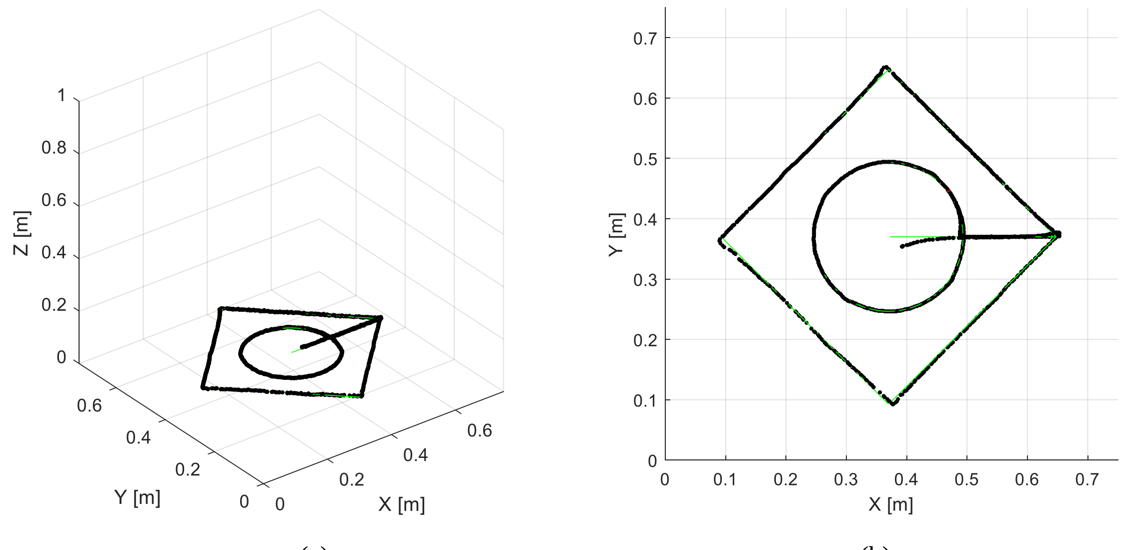


Figure 113. Experimental results with camera measurement errors and including real-time position feedback.

nearly indistinguishable from the nominal experiment of Figure 108. This means that, even though the camera has some errors in measuring the position of the mass, the effect of the controller is to drive the mass towards the reference trajectory. Furthermore, the reference trajectory followed by the mass is the trajectory as seen by the camera, even though the camera has an error in its view of the real world. This experimental result agrees exactly with the simulation result presented in Figure 81.

Returning to the question at hand, regarding the importance of the camera accuracy in the CSR system, the result in this experiment suggests that one can be confident in the CSR’s ability to track the reference position as seen by the camera. Then, one major focus in designing the CSR should be ensuring that an accurate, independent position sensor is incorporated into the system. If the position sensor is properly calibrated to the real-world, then the accurate positioning of the mass, as seen by the camera, can be trusted to reflect the real-world performance. On the other hand, any error accepted in the position-sensor data will affect the positioning of the mass in a predictable way if that sensor’s data is used for real-time feedback control of the type considered in this dissertation. Finally, the plotted results presented herein are understood to be only as accurate as the camera data, and it is claimed that the true system performance can be improved by simply incorporating a more accurate position sensor.

5.6. Dynamic Cable Disturbance

Another interesting question is how the CSR performance is affected by dynamic disturbances. A dynamic disturbance is one that varies with time, as opposed to being constant. In general, dynamic disturbances are more difficult to reject. One such disturbance can come from external forces on the CSR cables. These forces would be transmitted to the end effector and could induce vibrations. One way to address this is to directly measure the forces acting on the mass, as discussed in Section 3.1.6.

To test dynamic cable disturbances in the cable-length control scheme, servomotors with short arms were attached near the anchor points of cables *b* and *d* (Figure 100). These servomotors were programmed to move in a sweeping motion, such that the arms pushed the two cables periodically and changed the cable lengths by a few centimeters. Although the servo motion was at a constant speed, the placement of the servo arms was not precise. The resulting disturbances were irregular and difficult to measure.

The first experiment tested a slow movement of the disturbance servos. The servos moved at a speed of 0.6 rad/s. The result of this experiment is plotted in Figure 114, using

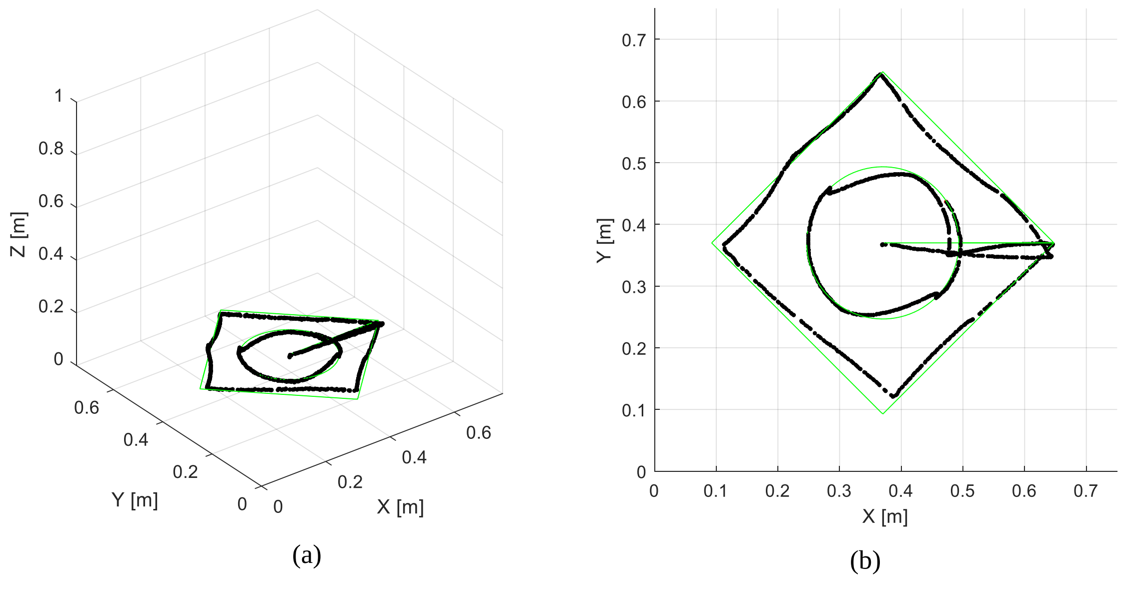


Figure 114. Experiment results with slow dynamic cable disturbances and no real-time position feedback. (a) Side and (b) top views.

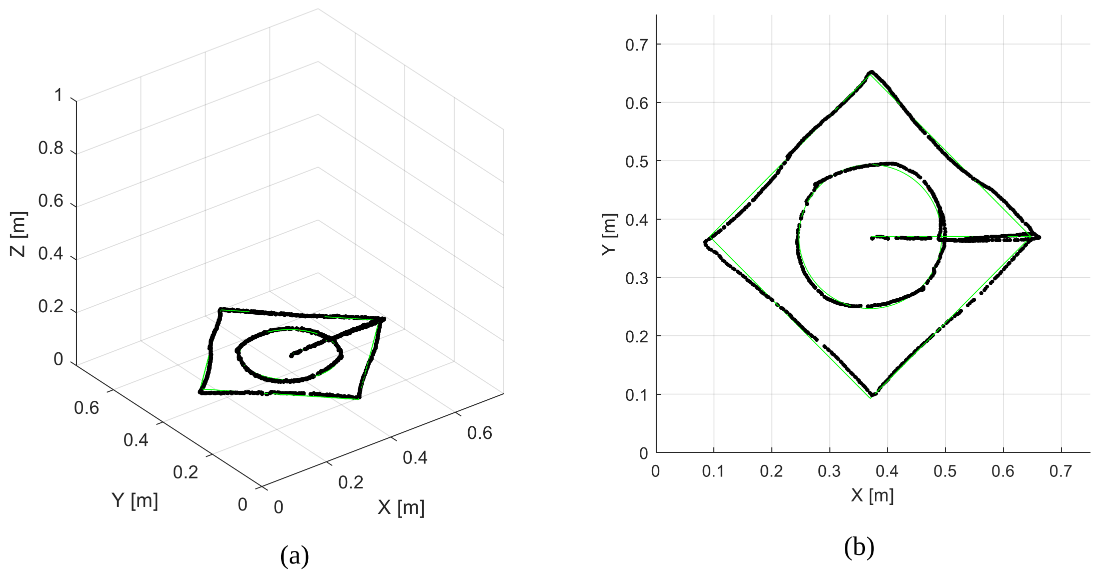


Figure 115. Experiment results with slow dynamic cable disturbances and including real-time position feedback. (a) Side and (b) top views.

the same test trajectory as in Figure 108. The test mass periodically moved away from the reference trajectory in an irregular way that would certainly result in AM build errors. Although other disturbance magnitudes were not tested, one can predict that the tracking error is commensurate with the change of the cable lengths due to the disturbance.

The experiment was repeated, this time including the real-time position feedback. The result is shown in Figure 115. While there is still some noticeable periodic motion in the trajectory, the magnitude of the tracking error is reduced, such that the mass remains within a small distance of the reference curve. Therefore, the inclusion of the feedback controller is effective in reducing the error due to the slow dynamic cable disturbances although the effects are not eliminated completely. In explaining this result, it is useful to recall that the integral term in the feedback controller, (70), must lag the dynamic

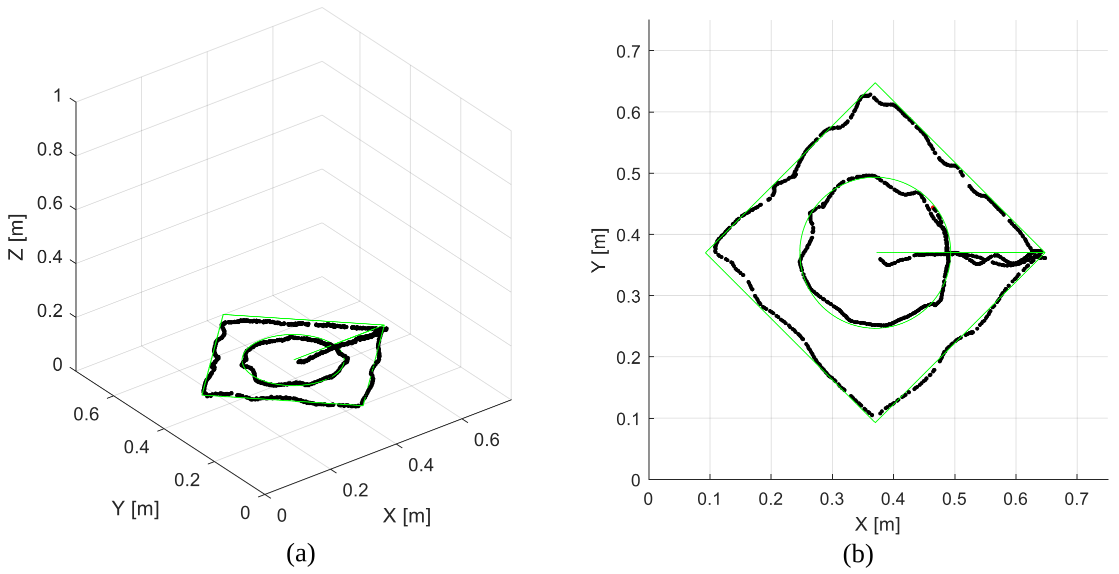


Figure 116. Experiment results with a moderately-fast dynamic cable disturbances and no real-time position feedback. (a) Side and (b) top views.

disturbances, since it takes a finite time for the position-error integral to accumulate. Therefore, the effective bandwidth of the feedback controller must be limited.

To explore the bandwidth of the feedback controller in rejecting dynamic disturbances, another experiment was performed with twice the movement speed of the disturbance servos. The placement of the servo arms was the same as before. The result of this trial is shown in Figure 116. Clearly, the faster cable disturbances result in a more drastic effect on the trajectory of the mass. While the overall movement follows the reference path, there is a chaotic motion that could almost be described as periodic. During this trial, the test mass was observed to swing slightly as the two cable lengths were suddenly moved by the servo arms. During these intermittent swings, some of the CSR

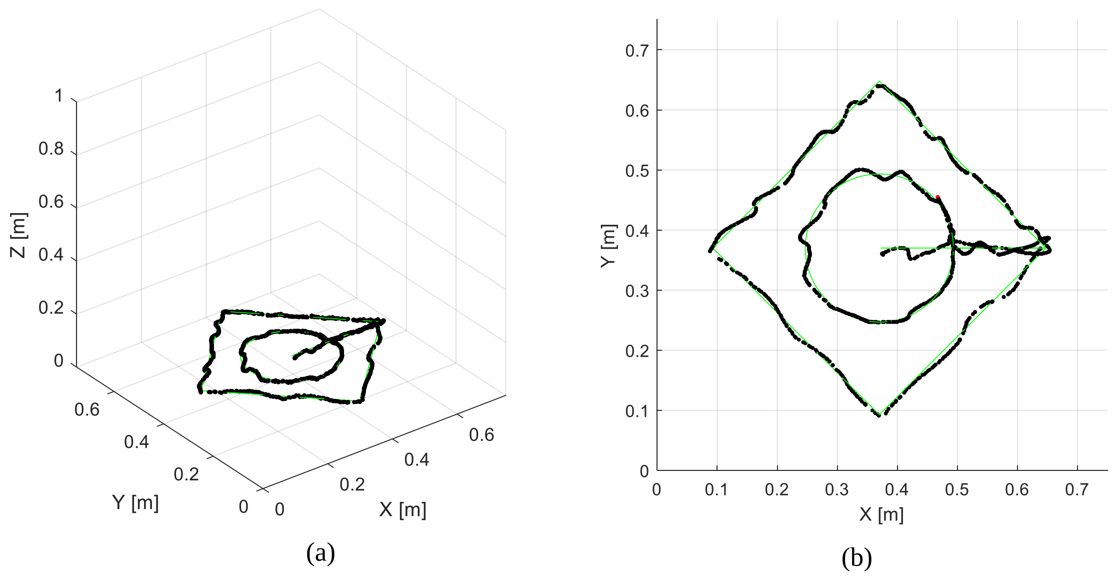


Figure 117. Experimental results with dynamic cable disturbances and including real-time position feedback. (a) Side and (b) top views.

cables appeared to exhibit slack, which is problematic for control, since a slack cable cannot provide a force to the test mass.

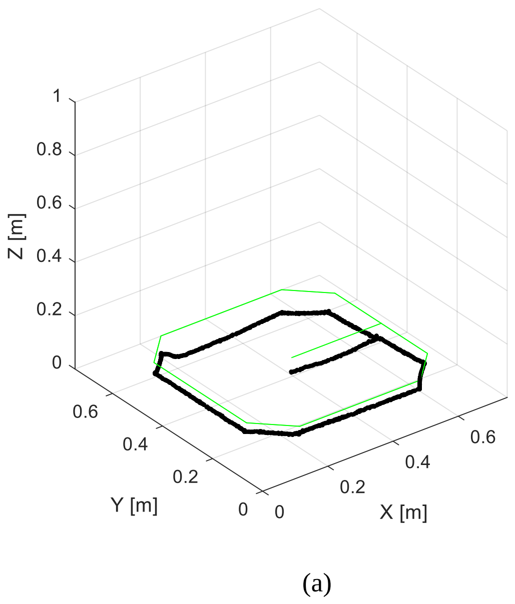
As before, the experiment was repeated with the position feedback control activated to see if the error could be effectively reduced. The result of this trial is plotted in Figure 117. This time, the position-feedback control does not produce a major improvement. At best, the overall trajectory trends closer to the reference curve. However, the chaotic motions are prevalent. This is the same result as in the simulation of Figure 83.

The results in this section supports the claim that the position-feedback effectiveness is somewhat limited to constant or low-frequency disturbances. This is a fundamental reality that warrants future work in search of an improved controller or, alternatively, some effective means for suppressing the chaotic vibrations in the end effector. This latter idea is, in part, why the proposed design of the CSR presented in Figure 22 features a nozzle-stabilization mechanism as part of the end effector. That is, the CSR may be effective in controlling the overall placement of the AM deposition nozzle along a desired trajectory, but it may still be necessary to implement some mechanism near the nozzle may for the fine-positioning task. Otherwise, the CSR may not be able to function properly in an environment with high-frequency external disturbances, and this would reduce the usefulness of the machine in AM applications.

5.7. Anchor-Point Position Error

The final error investigated was incorrect anchor-point positions. To this end, the eyehooks that guided the cables in the experimental CSR were intentionally moved from their nominal positions. Referring to the labels in Figure 84, the cable-*a* eyehook was displaced by (0, 0, 3.75) cm and the cable-*c* eyehook by (−7.0, 1.0, −2.5) cm. The *b* and *d* eyehooks were not altered. These values were chosen arbitrarily but were influenced by the experiment setup. In particular, the eyehooks were moved to different holes in the extruded aluminum pieces that made up the robot frame (Figure 92). The largest of the anchor-point displacements was about 10% of the frame width. It is unlikely that a real implementation of a CSR would have such large errors, so these displacements represent a worst-case scenario.

The effect of the anchor-point errors on the trajectory is shown in Figure 118. The test trajectory was the same as the one used in the simulation of Figure 84. Just as in the simulation, the errors drastically affect the path of the mass. The effect is comparable in magnitude to that of Figure 110, where the cable lengths were altered. To be exact, the



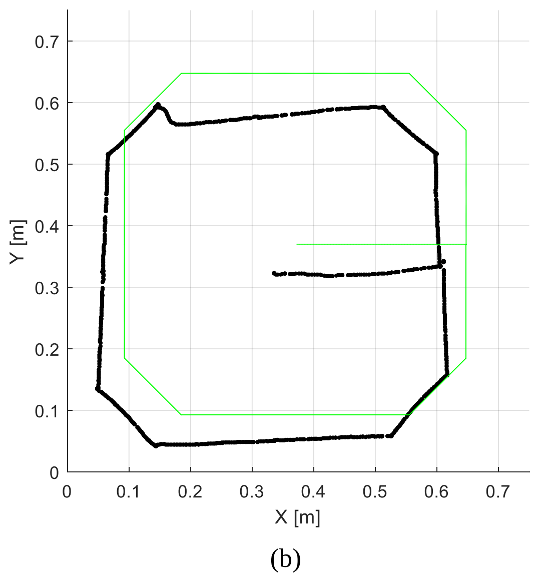
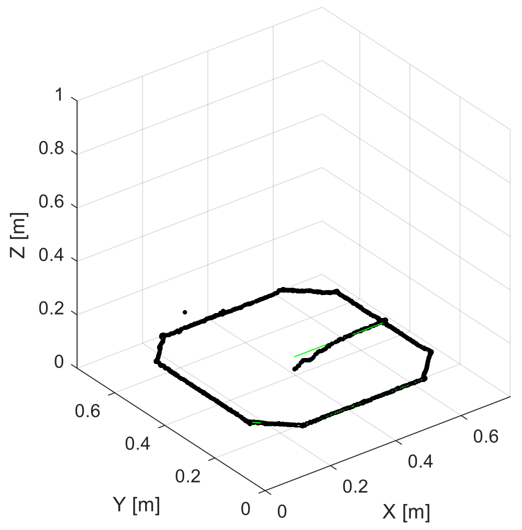


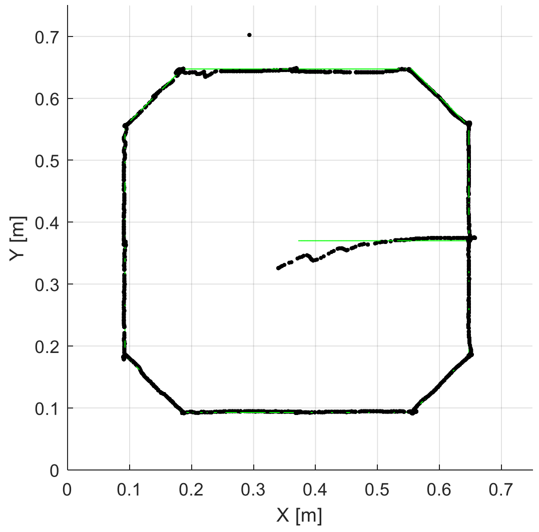
Figure 118. Experimental results with cable anchor-point position errors and no real-time position feedback. (a) Side and (b) top views.

positioning error in both of these experiments is due to an incorrect kinematics resulting from errors in setting up the robot configuration. Ideally, these errors would be prevented by careful setup of the machine. However, the portable CSR system that is proposed in this dissertation would be more susceptible to setup errors from repeated assembly of the machine in new locations.

The experiment was repeated with the use of the position-feedback, and these results are shown in Figure 119. Again, the controller is effective in reducing the tracking error almost completely. This result is somewhat surprising since the controller in no way helps predict the errors in the robot configuration due to the anchor-point displacements. Thus, the CSR system with the closed-loop position feedback is significantly tolerant to errors in the robot configuration, should they be present in the machine.



(a)



(b)

Figure 119. Experimental results with cable anchor-point position errors and including real-time position feedback. (a) Side and (b) top views.

5.8. Anchor-Point Position Estimation

Despite the CSR system being tolerant to some anchor-point-position errors, one may still be interested in gathering information about the true anchor-point positions. After all, once a configuration error is known, the robot kinematic equations can be updated to improve the performance of the CSR even without the use of the position-feedback control. This concept was demonstrated in the simulations of Section 3.2.6. The experimental counterpart of this simulation is now be presented.

The same experiment from Figure 118 was used to test the anchor-point estimation. The data were then processed using the method (75). The result is shown in Figure 120, where the true and estimated anchor-point displacements are compared. Some

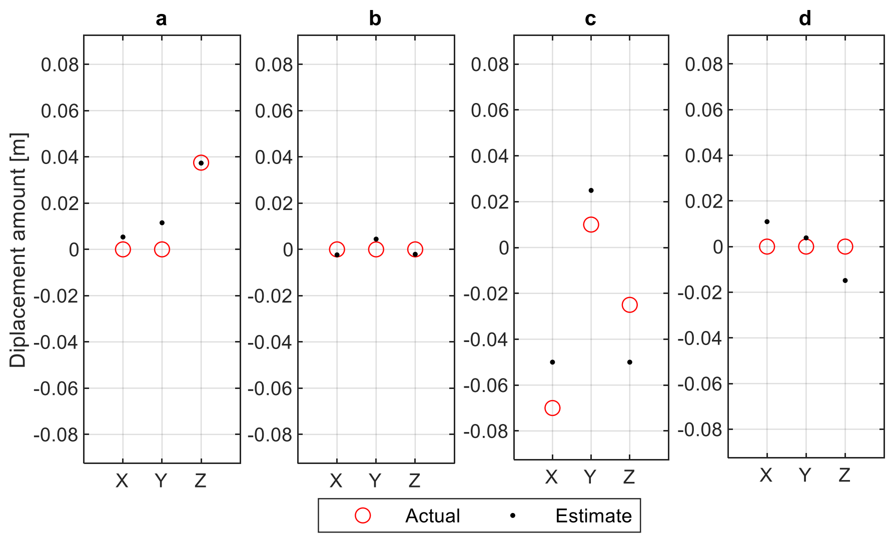


Figure 120. Estimated anchor-point displacement errors using experimental data and estimation method (75).

of the displacements are estimated quite accurately while for others there is some error. The simulations in Section 3.2.6 showed that even approximately correct estimates can significantly improve the CSR performance. Furthermore, the confidence in the estimation results can be increased by repeating the experiment several times. In fact, this estimation method could be incorporated into the setup of the CSR machine in a practical application, such that any constant anchor-point placement errors can be estimated before using the system. Then, the estimates would be used to update the robot kinematic equations to be used thereafter, and any errors in the estimates would be accounted for by using the closed-loop position feedback.

The anchor-point position estimation presented herein is offered as one additional tool for improving the use of the CSR in a large-scale application. Ideally, the setup of such a machine would be accompanied by accurate position sensors that could be used to position the cable anchor points to begin with. The method presented in this section can be substituted or used in conjunction with the usual equipment as a means to reduce operating costs or to increase confidence in the CSR configuration.

5.9. AM Layers

If the CSR is to be used for AM, it is also important to test the vertical motion of the end effector. In particular, one may be interested in experimenting with incremental changes in the mass height. This kind of movement is necessary for depositing material layers in an AM process. The material layers can be described as 2-D paths deposited on level planes. Each plane is slightly higher in the *z* direction than the previous plane.

To test this kind of movement, an experiment was performed where the CSR mass moves in a 2-D trajectory, rises by 1 cm, and then continues along a second 2-D trajectory. The overall path is like the one shown in Figure 108, but with the change in height following after the first two straight-path segments. This complete motion represents the layering process of an AM operation. The result of this test is plotted in Figure 121. The change in height can be observed in the bottom right data points of Figure 121(b), where the reference trajectory goes from 0.095 to 0.105 m. Before and after the vertical movement, there is some noticeable *z*-direction error. In this test, the real-time position feedback from the camera was not used, so the small position errors are the result of the small system imperfections like inexact spool radii, cable stretch, or measurement errors. Still, the overall motion of the mass is close to the desired trajectory. The same experiment

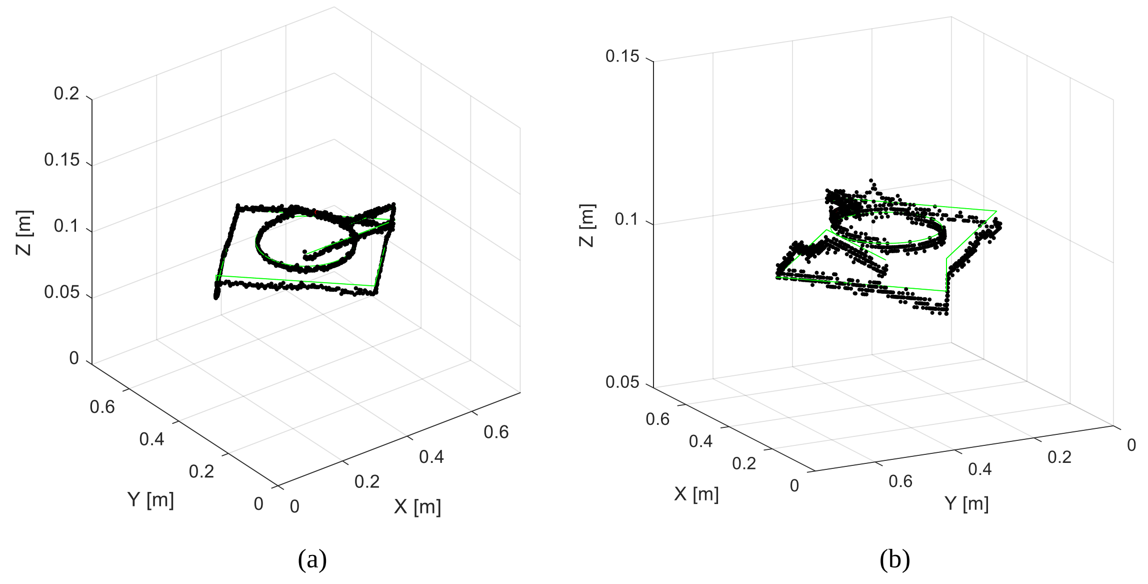


Figure 121. Experiment testing the vertical movement of 10 mm between two planar trajectories, without real-time position feedback. (a) Side view from the front and (b) side view showing the vertical movement.

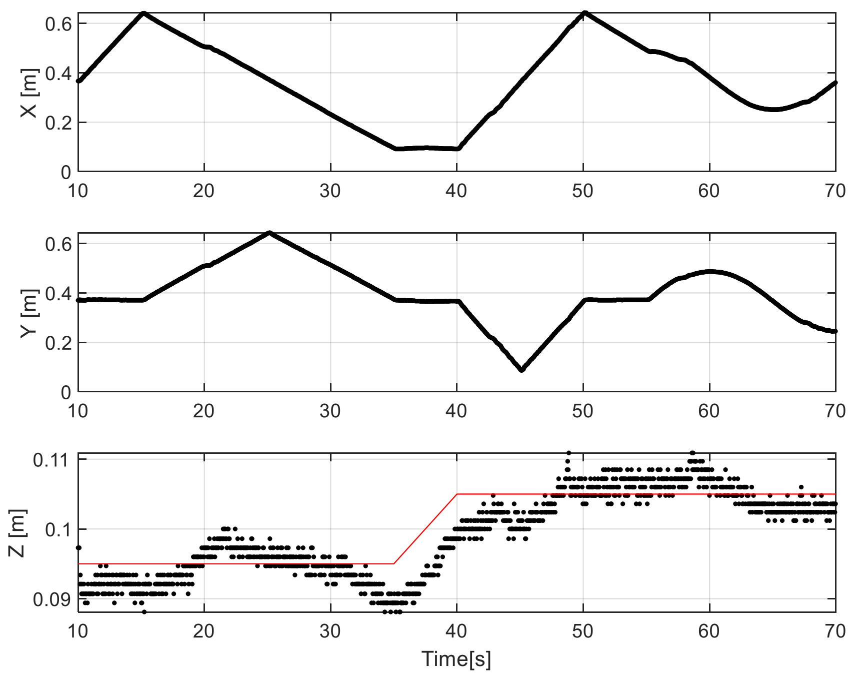
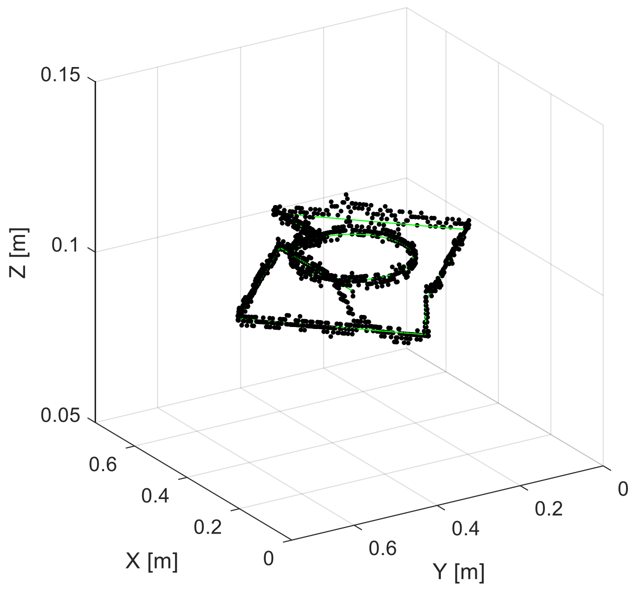


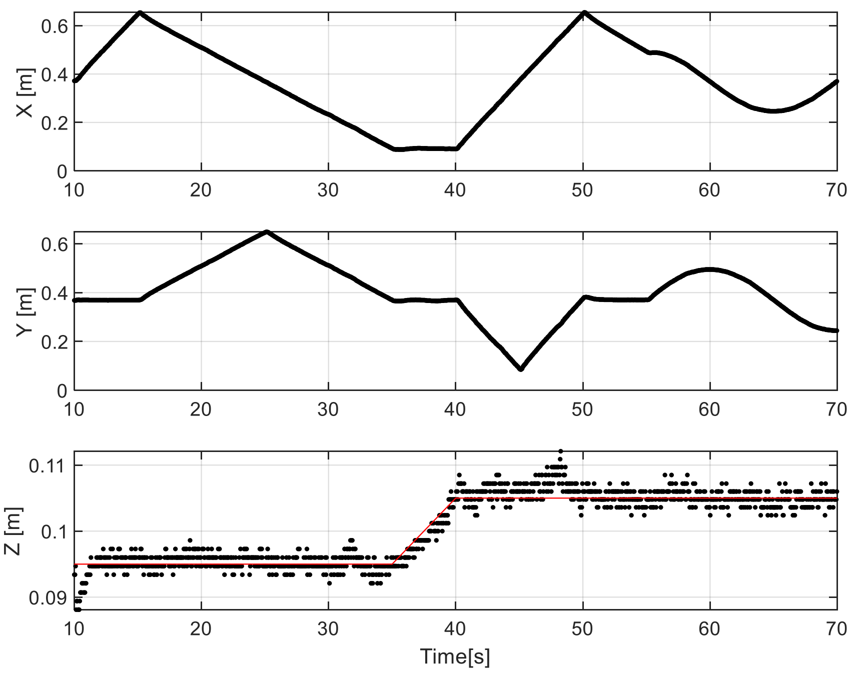
Figure 122. Mass position for the experiment shown in Figure 121. Beginning at 35 s, the reference trajectory was raised by 10 mm in the *z* direction, representing a change in AM-layer height. Real-time position feedback was not used in this test.

data are plotted versus time in Figure 122. Here, the reference height is shown as a solid line in the *z*-direction plot. The vertical motion being studied occurs from 35 to 40 s. During this time span, the mass tracks the trajectory with the same error that is shown in Figure 121.

The experiment was repeated, this time with the use of the real-time feedback control enabled by the camera system. The result of this experiment is shown in Figure 123. Now, the mass tracked the reference trajectory much closer, and the error was reduced. If this CSR were to be used for AM, the position-feedback control would produce a more precise deposition of the material layers in the 2-D path and a more consistent layer height. Without the feedback control, the minimum layer height would increase.



(a)



(b)

Figure 123. Mass position for an experiment with a 10-mm height change at 35 s. The real-time position feedback from the camera was used to reduce the positioning error. Shown here are (a) a 3-D perspective view and (b) a time plot of the *x*-*y*-*z* position. The reference height is shown as a solid line in the *z*-direction plot.

The 10-mm vertical movement was about 1.3% of the experimental CSR heigh and about four times the *z*-measurement resolution observed in Figure 106. To explore the practical limit of the layer-height precision for this CSR, another experiment was run with a height change of 5 mm. This is about 0.6% of the frame height and about two times the *z*-measurement resolution. The result of this trial is shown in Figure 124. This figure can be compared with the 10-mm experiment shown in Figure 121. The change in height is apparent in the bottom right of Figure 124(b), but the persistent error in the *z*-direction is significant enough that the 2-D trajectory does not remain at a consistent height after the vertical movement. In this case, it cannot be said that the CSR is effective in moving the end effector in 2-D paths along the planes with 5-mm precision. This conclusion is supported by the time plot of the mass position, shown in Figure 125. The mass height

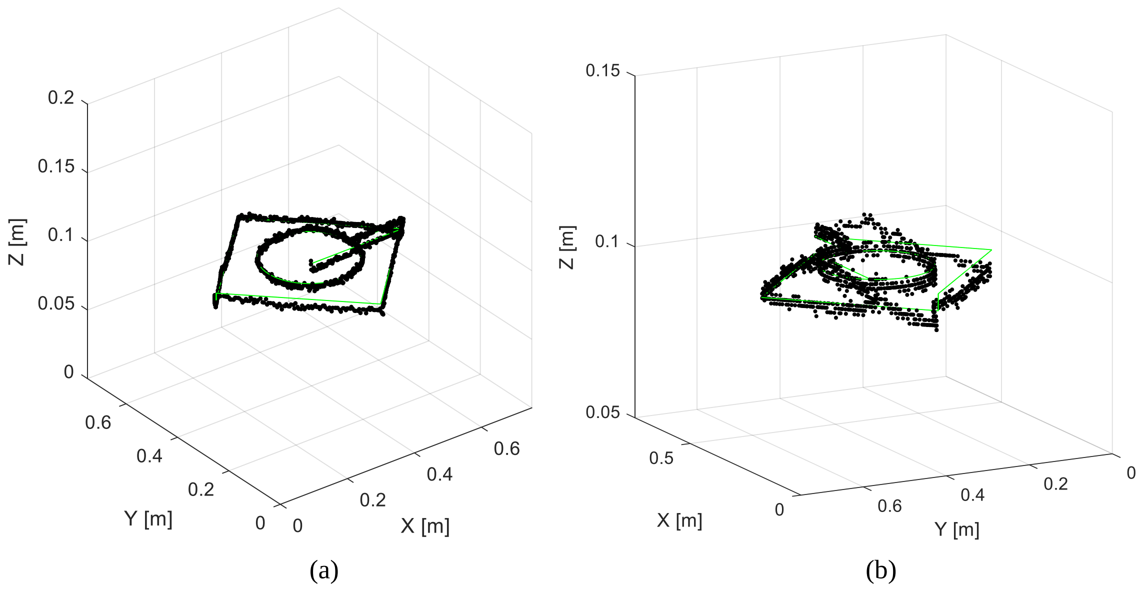


Figure 124. Experiment testing the vertical movement of 5 mm between two planar trajectories without real-time position feedback. (a) Side view from the front, and (b) side view showing the vertical movement.

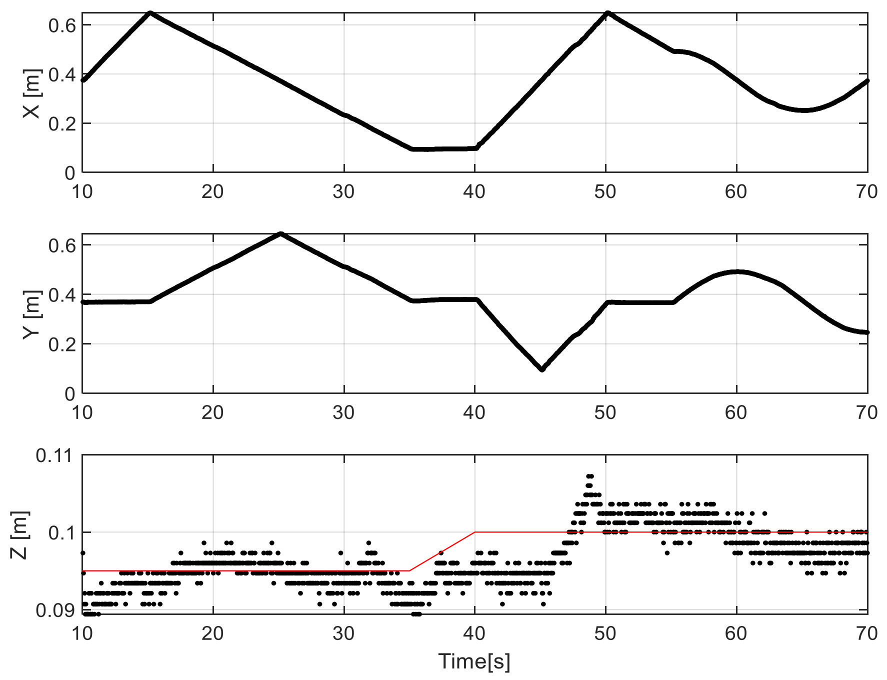
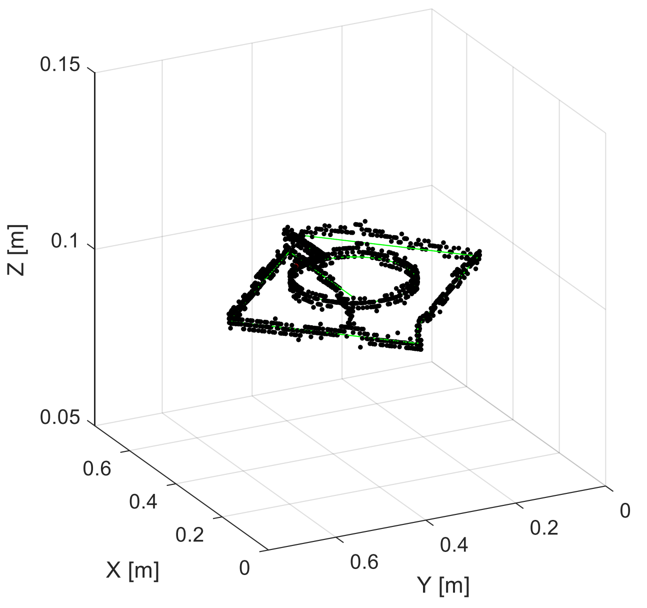


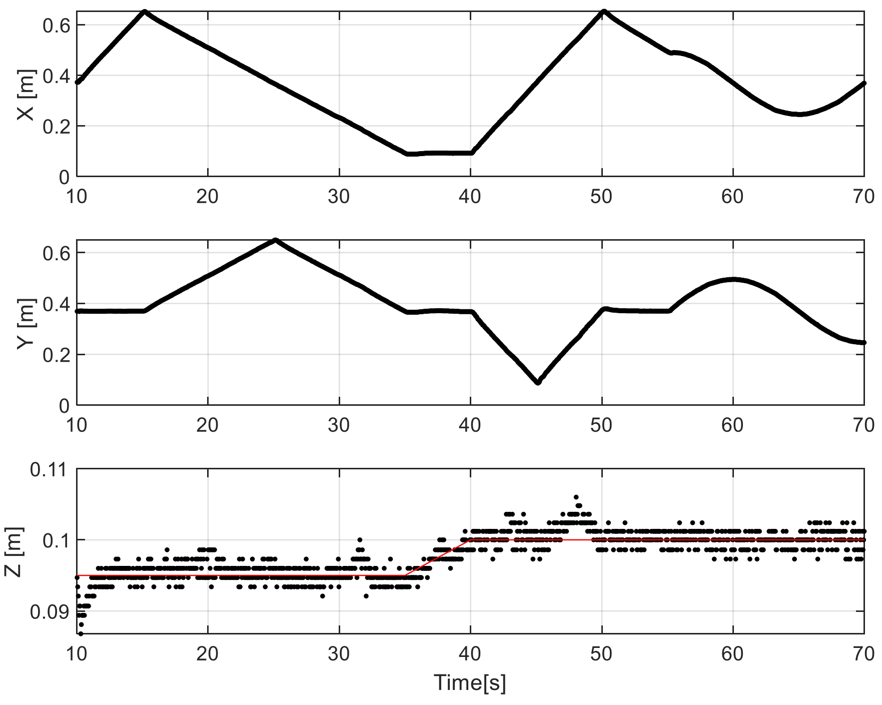
Figure 125. Mass position for the experiment shown in Figure 124. Beginning at 35 s, the reference trajectory was increased by 5 mm in the *z* direction. Real-time position feedback was not used in this test.

varies by several millimeters throughout the trial, such that it is difficult to distinguish between the two heights in the mass trajectory. Overall, it cannot be said that the mass tracks the reference *z* position effectively.

The experiment for a 5-mm vertical movement was repeated with the real-time position feedback. The result is shown in Figure 126. The trajectory of the mass followed the reference much closer this time, such that the heights can be distinguished from each other in the z-direction plot of Figure 126(b). Although there are still small variations in the *z*-position of the mass, the CSR is effective in maintaining a level height throughout the movement. Thus, the precision of the positioning in the *z*-direction appears to be about four times the camera-measurement resolution for this system.



(a)



(b)

Figure 126. Mass position for an experiment with a 5-mm height change at 35 s. The real-time position feedback from the camera was used to reduce the positioning error. Shown here are (a) a 3-D perspective view and (b) a time plot of the *x*-*y*-*z* position. The reference height is shown as a solid line in the *z*-direction plot.

If the goal is to use the CSR for large-scale AM, experiments like the ones presented here would be beneficial in determining the achievable positioning resolution of the system. Furthermore, the inclusion of real-time position feedback can help improve the positioning precision almost to the same level as the measurement resolution, as shown in this section. The real-world results would vary based on some design details, such as the accuracy of the servo motors or the precision of the system sensors. Nevertheless, the same methods that were demonstrated in this dissertation can be used to achieve better results in a practical application.

6. CONCLUSIONS AND FUTURE WORK

This dissertation explored one idea born from the combination of two modern technologies, AM and CSRs. On their own, these technologies enable manufacturing tasks in flexible ways that are suitable for automation and reduced costs. Some of the AM technologies, like FDM, are so simple in working principle that they were already demonstrated at large scales. One prevalent example is the construction of buildings using concrete extrusion. At such scales, it is worth considering novel mechanisms that can provide for accurate positioning of the AM nozzle in a low-cost, flexible manner that is also portable. One novel mechanism is the CSR. The focus of this dissertation was the design, analysis, and control of such a machine.

First, the conceptual design of a four-cable CSR was presented. In this design, the coarse position of the AM nozzle is controlled by the system cables but the fine position and orientation of the nozzle is controlled by some unspecified mechanism. The design of this mechanism is a critical element for the final realization of the AM concept and is suggested for future work. Otherwise, the CSR configuration must be revisited to provide for full orientation control of the nozzle. Another part of the design that was unspecified is the management strategy for the material hose. Since the hose is sure to add weight, tension, and lateral forces to the nozzle mechanism, effective hose management should be studied and eventually tested. Related items for further consideration are the nozzle dimensions and the fluid reaction forces that result from depositing material. Once the nozzle mechanism is designed the system cables can also be selected.

Second, the analyses and experiments presented in this dissertation demonstrate that a four-cable CSR with a square footprint can be used to provide the slow movements typically seen in AM processes. Furthermore, the cable-length control of the system with a dual-loop structure allows for precise positioning of the nozzle as long as the cables are relatively light and inextensible. This idea was supported through simulations and successfully implemented in the laboratory-scale experiment to provide for precise positioning of the robot test mass. Several movement trajectories, including straight and circular paths were successfully tracked in the experiment with errors smaller than 1 cm. Vertical movements between the paths also demonstrated that the CSR can accurately provide AM layer height changes as small as 5 mm. However, all of the motions in this dissertation were realtively slow. In future work, higher accelerations could be tested to analyze the system dynamics more fully and to study the phenomenon of cable slack. Futhermore, additional external forces could be tested to mimic the realistic working conditions of a large-scale CSR. For example, the forces could mimic realistic cable mechanics or the existence of a material hose.

Finally, CSR imperfections like cable-length errors, position-sensor errors, cable anchor-point errors, and some dynamic cable disturbances were effectively addressed using the real-time feedback from an independent position sensor. In every case, except for the fast dynamic cable disturbances, the position feedback greatly reduced the trajectory-tracking errors from several centimeters to less than 1 cm. The positon data from the independent sensor was also used to estimate the true position of the cable anchor points when these were displaced from their ideal locations. Therefore, the indpenedent position meausurement is a useful addition not just for real-time control but also for offline analysis of the CSR. This provides an opportunity to apply analysis tools beyond what was done in this dissertation. For example, artificial intelligence tools like machine learning could be an effective way to address the CSR system imperfections in real time. The work presented here is a significant contribution to the field since it is a marked departure from previous studies, where ideal working conditions for the CSR were the norm.

One possible improvement for experimentation is the further development and testing of the CSR for FDM. This means selecting and implementing a material extruder in place of the test mass. Then, the full operation of the machine could be studied with realistic contact forces from depositing AM layers. There is also an opportunity to test some of the tension-based control methods like those presented in this dissertation. This would require installing tension sensors in the experiment and implementing the sensor data in a closed-loop controller. A secondary position sensor, such as an IMU, could also be a beneficial addition to the experiment since it would corroborate the position data collected by the camera.

Even though various aspects of the CSR were studied, simulated, and tested in this work, there remain many open questions that should be answered before the idea of a CSR for large-scale AM is implemented in full. Nevertheless, the content of this dissertation serves as a foundational work for exploring this idea, and the analysis tools developed here can be useful in future efforts.

REFERENCES

[1] F. Calignano, D. Manfredi, E. P. Ambrosio, S. Biamino, M. Lombardi, E. Atzeni, A. Salmi, P. Minetola, L. Iuliano, and P. Fino, “Overview on additive manufacturing technologies,” *Proc. IEEE*, vol. 105, no. 4, pp. 593−612, Apr. 2017.

[2] J. Gardan, “Additive manufacturing technologies: state of the art and trends,” in *Additive Manufacturing Handbook*, A. B. Badiru, V. V. Valencia, and D. Liu, Eds., Boca Raton, FL, USA: CRC Press, 2017, pp. 155−161.

[3] Y. Ning, Y. S. Wong, J. Y. H. Fuh, and H. T. Loh, “An approach to minimize build errors in direct metal laser sintering,” *IEEE Trans. Autom. Sci. Eng.*, vol. 3, no. 1, pp. 73−80, Jan. 2006.

[4] Q. Wang, J. Li, M. Gouge, A. R. Nassar, P. Michaleris, and E. W. Reutzel, “Reduced-order multivariable modeling and nonlinear control of melt-pool geometry and temperature in directed energy deposition,” in *Proc. 2016 American Control Conference*, Boston, MA, USA, Aug. 2016, pp. 845–851.

[5] The University of Maine. “World’s largest 3D printer.” Advanced Structures and Composites Center. https://composites.umaine.edu/am-equipment (accessed Jul. 15, 2021).

[6] Apis Cor. “Introducing apis cor.” Apis-Cor. https://www.apis-cor.com/news (accessed Jul. 15, 2021).

[7] S. Lim, R. A. Buswell, T. T. Le, S. A. Austin, A. G. F. Gibb, and T. Thorpe, “Developments in construction-scale additive manufacturing processes,” *Automation in Construction*, vol. 21, pp. 262−268, Jan. 2012.

[8] M. Sakin and Y. C. Kiroglu, “3D printing of buildings: construction of the sustainable houses of the future by BIM,” in *Proc. 9th Int. Conf. on Sustainability in Energy and Buildings*, Crete, Greece, Jul. 2017, pp. 702−711.

[9] M. Yossef and A. Chen, “Applicability and limitations of 3D printing for civil structures,” in *Proc. Conf. on Autonomous and Robotic Construction of Infrastructure*, Ames, IA, Jun. 2015, pp. 237−246.

[10] P. Wu, J. Wang, and X. Wang, “A critical review of the use of 3-D printing in the construction industry,” *Automation in Construction*, vol. 68, pp. 21−31, Aug. 2016.

[11] M. Harris, J. Potgieter, K. Arif, and R. Archer, “Large scale 3D printing: feasibility of novel extrusion based process and requisite materials,” in *Proc. 24th Int. Conf. Mechatronics and Machine Vision in Practice,* Auckland, New Zeeland, Nov. 2017, pp. 1−6.

[12] Y. Vijay, N. D. Sanandiya, S. Dristas, and J. G. Ferandez, “Control of process settings for large-scale additive manufacturing with sustainable natural composites,” *ASME Journal of Mechanical Design*, vol. 141, no. 8, pp. 1−12, Aug. 2019.

[13] Z. Wang, R. Liu, T. Sparks, and F. Liou, “Large-scale deposition system by an industrial robot (I): design of fused pellet modeling system and extrusion process analysis,” *3D Printing and Additive Manufacturing*, vol. 3, no. 1, pp. 39−47, Mar. 2016.

[14] Sciaky Inc. “The EBAM 300® system.” Electron Beam Additive Manufacturing (EBAM®). https://www.sciaky.com/additive-manufacturing/industrial-metal-3d-printers (accessed Jul. 15, 2021).

[15] G. Cesaretti, E. Dini, X. De Kestelier, V. Colla, and L. Pambaguian, “Building components for an outpost on the Lunar soil by means of a novel 3D printing technology,” *Acta Astronautica*, vol. 93, no. 1, pp. 430−450, Jan. 2014.

[16] S. Donohoe, S. A. Velinsky, and T. A. Lasky, “Optimal force generation for an underconstrained planar cable robot,” *Mechanics Based Design of Structures and Machines*, vol. 43, no. 1, pp. 19−37, Sep. 2015.

[17] H. Jamshidifar, A. Khajepour, B. Fidan, and M. Rushton, “Kinematically-constrained redundant cable-driven parallel robots: modeling, redundancy analysis, and stiffness optimization,” *IEEE/ASME Trans. Mechatronics*, vol. 22, no. 2, pp. 921−930, Apr. 2017.

[18] S. Tadokoro, Y. Murao, M. Hiller, R. Murata, H. Kohkawa, and T. Matsushima, “A motion base with 6-DOF by parallel cable drive architecture,” *IEEE/ASME Trans. Mechatronics*, vol. 7, no. 2, pp. 115−123, Jun. 2002.

[19] R. G. Roberts, T. Graham, and J. M. Trumpower, “On the inverse kinematics and statics of cable-suspended robots,” in *Proc. 1997 IEEE Int. Conf. Syst., Man, Cybern.*, Orlando, FL, USA, Oct. 1997, pp. 4291−4296.

[20] R. G. Roberts, T. Graham, and T. Lippitt, “On the inverse kinematics, statics, and fault tolerance of cable-suspended robots,” *Journal of Robotic Systems*, vol. 15, no. 10, pp. 581−597, Dec. 1998.

[21] H. Yuan, E. Courteille, and D. Deblaise, “Static and dynamic stiffness analyses of cable-driven parallel robots with non-negligible cable mass and elasticity,” *Mechanism and Machine Theory*, vol. 85, no. 2015, pp. 64−81, Mar. 2015.

[22] J. J. Gorman, K. W. Jablokow, and D. J. Cannon, “The cable array robot: theory and experiment,” in *Proc. Int. Conf. Robotics and Automation*, Seoul, Korea, May 2001, pp. 2804−2810.

[23] E. Barnett and C. Gosselin, “Time-optimal trajectory planning of cable-driven parallel mechanisms for fully specified paths with G^1-discontinuities,” *Journal of Dynamic Systems, Measurement, and Control*, vol. 137, no. 1, pp. 1−12, Jul. 2015.

[24] W.-J. Shiang, D. Cannon, and J. Gorman, “Dynamic analysis of the cable array robotic crane,” in *Proc. IEEE Int. Conf. Robotics and Automation,* Detroit, MI, USA, May 1999, pp. 2495−2500.

[25] M. Gouttefarde, J.-F. Collard, N. Riehl, and C. Baradat, “Geometry selection of a redundantly actuated cable-suspended parallel robot,” *IEEE Trans. on Robotics*, vol. 31, no. 2, pp. 501−509, Apr. 2015.

[26] B. Wang, B. Zi, S. Qian, “Collision free force closure workspace determination of reconfigurable planar cable driven parallel robot,” in *Proc. 2016 Asia-Pacific Conf. Intelligent Robot Systems*, Tokyo, Japan, Jul.2016, pp. 26−30.

[27] D. Song, L. Zhang, and F. Xue, “Configuration optimization and a tension distribution algorithm for cable-driven parallel robots,” *IEEE Access*, vol. 6, no. 1, pp. 33928−33940, Jun. 2018.

[28] S.-R. Oh and S. K. Agrawal, “The feasible workspace analysis of a set point control for a cable-suspended robot with input constraints and disturbances,” *IEEE Trans. Control Systems Technology*, vol. 14, no. 4, pp. 735−742, Jul. 2006.

[29] Y. Zhong and S. Qian, “A cable-driven parallel robot for 3D printing,” in *Proc. IEEE Int. Conf. Mechatronics, Robotics, and Automation*, Hefei, China, May 2018, pp. 199−203.

[30] B. Zi, B. Y. Duan, J. L. Du, and H. Bao, “Dynamic modeling and active control of a cable-suspended parallel robot,” *Mechatronics*, vol. 18, no. 1, pp. 1−12, Feb. 2008.

[31] R. Mersi, S. Vali, M. S. Haghighi, G. Abbasnejad, and M. T. Masouleh, “Design and control of a suspended cable-driven parallel robot with four cables,” in *Proc. Int. Conf. Robotics and Mechatronics*, Tehran, Iran, Oct. 2018, pp. 470−475.

[32] Z. Amare, B. Zi, S. Qian, J. Du, and Q. J. Ge, “Three-dimensional static and dynamic stiffness analyses of the cable drive parallel robot with non-negligible cable mass and elasticity,” *Mechanics Based Design of Structures and Machines,* vol. 46, no. 4, pp. 455−482, Aug. 2017.

[33] D. Surdilovic, J. Radojicic, and J. Krüger, “Geometric stiffness analysis of wire robots: a mechanical approach,” in *Cable-Driven Parallel Robots*, T. Bruckmann and A. Pott, Berlin, D.E.: Springer, 2013, ch. 8, pp. 389–404.

[34] B. Zi, N. Wang, S. Qian, and K. Bao, “Design, stiffness analysis and experimental study of a cable-driven parallel 3D printer,” *Mechanism and Machine Theory,* vol. 132, no. 2019, pp. 207−222, Feb. 2019.

[35] J.-D. Deschenes, P. Lambert, S. Perreault, N. Martel-Brisson, N. Zoso, A. Zaccarin, and P. Hebert, “A cable-driven parallel mechanism for capturing object appearance from multiple viewpoints,” in *Proc. Sixth Int. Conf. 3-D Digital Imaging and Modeling,* Montreal, Canada, Aug. 2007, pp. 367−374.

[36] M. H. Korayem, H. Tourajizadeh, M. Taherifar, S. Khayatzadeh, M. Maddah, A. Imanian, and A. Tajik, “A novel method for recording the position and orientation of the end effector of a spatial cable-suspended robot and using for closed loop control,” *Int. J. Adv. Manufacturing Technologies*, vol. 72, no. 5, pp. 739−755, May 2014.

[37] J. Lin and C.-K. Chiang, “Motion control of a cable-suspended robot using image recognition with coordinate transformation,” *Journal of Systems and Control Engineering*, vol. 235, no. 1, pp. 52−67, Jul. 2020.

[38] J. Lin and G. –T. Liao, “A modularized cable-suspended robot: implementation and oscillation suppression control,” *Journal of Systems and Control Engineering*, vol. 230, no. 9, pp. 1030−1043, Aug. 2016.

[39] A. B. Alp and S. K. Agrawal, “Cable suspended robots: feedback controllers with positive inputs,” in *Proc. American Control Conference*, Anchorage, AK, USA, May 2002, pp. 815−820.

[40] A. Aflakiyan, H. Bayani, and M. T. Masouleh, “Computed torque control of a cable suspended parallel robot,” in *Proc. Int. Conf. Robotics and Mechatronics*, Tehran, Iran, Oct. 2015, pp. 749−754.

[41] S.-R. Oh and S. K. Agrawal, “A reference governor-based controller for a cable robot under input constraints,” *IEEE Trans. Control Systems Technology*, vol. 13, no. 4, pp. 639−645, Jul. 2005.

[42] M. H. Korayem, M. Yousefzadeh, and S. Manteghi, “Tracking control and vibration reduction of flexible cable-suspended parallel robots using a robust input shaper,” *Scientia Iranica B*, vol. 25, no. 1, pp. 230−252, Jan. 2018.

[43] S.-R. Oh and S. K. Agrawal, "Nonlinear sliding mode control and feasible workspace analysis for a cable suspended robot with input constraints and disturbances," in *Proc. American Control Conf.*, Boston, MA, USA, Jul. 2004, pp. 4631−4636.

[44] M. H. Korayem, H. Tourajizadeh, and M. Bamdad, "Dynamic load carrying capacity of flexible cable suspended robot: robust feedback linearization control approach," *Journal of Intelligent and Robotic Systems*, vol. 2010, no. 60, pp. 341−363, May 2010.

[45] G. Mottola, C. Gosselin, and M. Carricato, "Dynamically feasible motions of a class of purely-translational cable-suspended parallel robots," *Mechanisms and Machine Theory*, vol. 132, no. 2019, pp. 193−206, Feb. 2019.

[46] H. Tourajizadeh and M. H. Korayem, "Optimal regulation of a cable suspended robot equipped with cable interfering avoidance controller," *Advanced Robotics*, vol. 30, no. 19, pp. 1273−1287, May 2016.

[47] M. H. Korayem, H. Tourajizadeh, A. Zehfroosh, and A. H. Korayem, "Optimal path planning of a cable-suspended robot with moving boundary using optimal feedback linearization approach," *Nonlinear Dynamics*, vol. 78, no. 2, pp. 1515−1543, Oct. 2014.

[48] D. Lin, G. Mottola, M. Carricato, and X. Jiang, "Modeling and control of a cable suspended sling-like parallel robot for throwing operations," *Applied Sciences*, vol. 2020, no. 10, pp, 1−17, Dec. 2020.

[49] S.-R. Oh and S. K. Agrawal, "A control Lyapunov approach for feedback control of cable-suspended robots," in *Proc. IEEE Int. Conf. Robotics and Automation*, Roma, Italy, Apr. 2007, pp. 4544−4549.

[50] R. de Rijk, M. Rushton, and A. Khajepour, “Out-of-plane vibration control of a planar cable-driven parallel robot,” *IEEE/ASME Trans. Mechatronics*, vol. 23, no. 4, pp. 1684−1692, Aug. 2018.

[51] B. Khoshnevis, “Automated construction by contour crafting—related robotics and information technologies,” *Automation in Construction*, vol. 13, no. 1, pp. 5−19, Jan. 2004.

[52] P. Bosscher, R. L. Williams II, L. S. Bryson, and D. Castro-Lacouture, “Cable-suspended robotic contour crafting system,” *Automation in Construction*, vol. 17, no. 1, pp. 45−55, Feb. 2007.

[53] E. Barnett and C. Gosselin, “Large-scale 3D printing with a cable-suspended robot,” *Additive Manufacturing*, vol. 7, pp. 27−44, Jul. 2015.

[54] J.-B. Izard, A. Dubor, P.-E. Herve, E. Cabay, D. Culla, M. Rodriguez, and M. Barrado, “Large-scale 3D printing with cable-driven parallel robots,” *Construction Robotics*, vol. 1, no. 1, pp. 1−8, Dec. 2017.

[55] T. P. Tho and N. T. Thinh, “Using a cable-driven parallel robot with applications in 3D concrete printing,” *Applied Sciences*, vol. 2021, no. 11, pp. 1−23, Jan. 2021.

[56] R. Cravotta, “Flying over the action,” *EDN*, vol. 48, no. 26, pp. 30−33, Jul. 2003.

[57] R. R. Thompson and M. S. Blackstone, “Three-dimensional moving camera assembly with an informational cover housing,” U.S. Patent 6873355 B1, Mar. 29, 2005.

[58] Y.-W. Wei, W.-M. Chen, and H.-H. Tsai, “Accelerating the Bron-Kerbosch algorithm for maximal clique enumeration using GPUs,” *IEEE Trans. Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2352−2366, Sep. 2021.

[59] H. Jeong, S.-J. Han, S.-H. Choi, Y. J. Lee, S. T. Yi, and K. S. Kim, “Rheological property criteria for buildable 3D printing concrete,” *Materials,* vol. 2019, no. 12, pp. 1−21, Feb. 2019.

[60] F. Bos, R. Wolfs, Z. Ahmed, and T. Salet, “Additive manufacturing of concrete in construction: potentials and challenges of 3D concrete printing,” *Virtual and Physical Prototyping*, vol. 11, no. 3, pp. 209−225, Aug. 2016.

[61] M. Valente, A. Sibai, and M. Sambucci, “Extrusion-based additive manufacturing of concrete products: revolutionizing and remodeling the construction industry,” *Journal of Composites Science*, vol. 2019, no. 3, pp. 1−20, Sept. 2019. Available online at www.tandfonline.com.

[62] N. Roussel, “Rheological requirements for printable concretes,” *Cement and Concrete Research*, vol. 112, no. 1, pp. 76−85, Oct. 2018.

[63] R. A. Buswell, W. R. L. de Silva, S. Z. Jones, and J. Dirrenberger, “3D printing using concrete extrusion: a roadmap for research,” *Cement and Concrete Research*, vol. 112, no. 1, pp. 37−49, Oct. 2018.

[64] A. Perrot, D. Rangeard, and A. Pierre, “Structural built-up of cement-based materials used for 3D-printing extrusion techniques,” *Materials and Structures,* vol. 2016, no. 49, pp. 1213−2330, Feb. 2015.

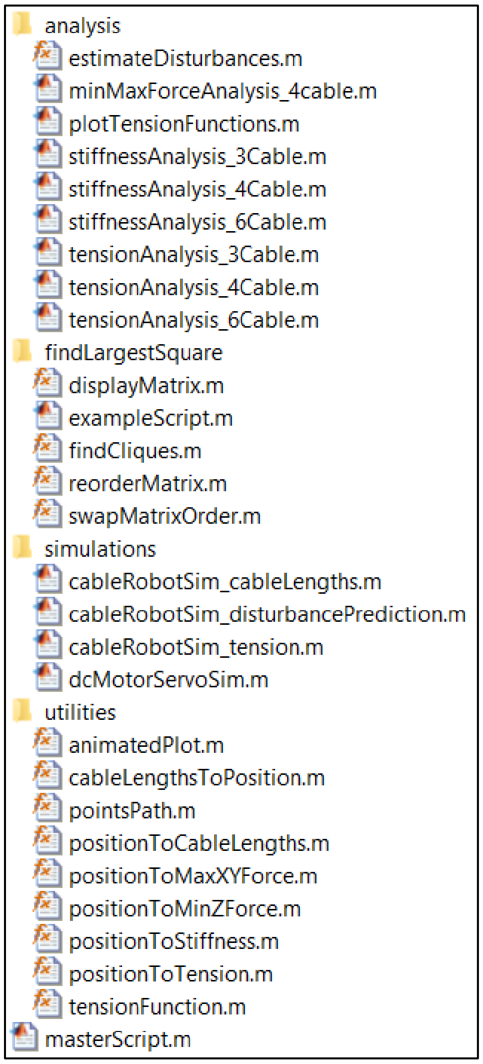
[65] T. T. Le, S. A. Austin, S. Lim, R. A. Buswell, A. G. F. Gibb, and T. Thorpe, “Mix design and fresh properties for high-performance printing concrete,” *Materials and Structures,* vol. 2012, no. 45, pp. 1221−1232, Jan. 2012.

[66] C. L. Phillips and H. T. Nagle, “Digital controller design,” in *Digital Control System Analysis and Design*, 3rd Ed., Upper Saddle River, NJ: Prentice Hall, Inc., 1995, ch. 8, sec. 9, pp. 314−315.

APPENDIX A:

ANALYSIS CODES

The tables in this appendix contain the MATLAB codes used in the analysis work of this dissertation. To run these codes, the files should be saved as shown in Figure 127.



```
analysis
    estimateDisturbances.m
    minMaxForceAnalysis_4cable.m
    plotTensionFunctions.m
    stiffnessAnalysis_3Cable.m
    stiffnessAnalysis_4Cable.m
    stiffnessAnalysis_6Cable.m
    tensionAnalysis_3Cable.m
    tensionAnalysis_4Cable.m
    tensionAnalysis_6Cable.m
findLargestSquare
    displayMatrix.m
    exampleScript.m
    findCliques.m
    reorderMatrix.m
    swapMatrixOrder.m
simulations
    cableRobotSim_cableLengths.m
    cableRobotSim_disturbancePrediction.m
    cableRobotSim_tension.m
    dcMotorServoSim.m
utilities
    animatedPlot.m
    cableLengthsToPosition.m
    pointsPath.m
    positionToCableLengths.m
    positionToMaxXYForce.m
    positionToMinZForce.m
    positionToStiffness.m
    positionToTension.m
    tensionFunction.m
masterScript.m
```

Figure 127. Folder and file structure for the MATLAB analysis codes in this dissertation.

Table A1. *masterScript.m* is the script that runs all of the other analysis codes.

```
% ==========================================================================
%                                Analysis
% ==========================================================================

%% Cable tension function
addpath('analysis')
addpath('utilities')
close all; clear all; clc;

% Run the tension analysis scripts (found in the analysis folder)
plotTensionFunctions


%% Tension
addpath('analysis')
addpath('utilities')
close all; clear all; clc;

% Run the tension analysis scripts (found in the analysis folder)

% These scripts plot the cable tensions at different positions
tensionAnalysis_4Cable
tensionAnalysis_3Cable
tensionAnalysis_6Cable

% This anaalysis plots the x-y or z force requirements for a cable robot
minMaxForceAnalysis_4cable


%% Stiffness
addpath('analysis')
addpath('utilities')
close all; clear all; clc;

% Run the tension analysis scripts (found in the analysis folder)
stiffnessAnalysis_4Cable
stiffnessAnalysis_3Cable
stiffnessAnalysis_6Cable


%% Position to cable lengths
addpath('utilities')
close all; clear all; clc;

% Positions to calculate cable lengths for
positions = [0.3 0.4 0.5];
% Cable anchor point locations
B = 1;      % square base length [m]
H = 1;      % height [m]
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];
% Calculate ideal cable lengths for the positions and cable anchor points
cableLengths = positionToCableLengths(positions, anchorPoints);
% Display the cable suspended robot
time = 0;
plotCables = true;
animatedPlot(time, positions, cableLengths, anchorPoints, plotCables);


%% Cable lengths to position
addpath('utilities')
addpath('findLargestSquare')
close all; clear all; clc;

% Build area size
```

```
B = 1;      % square base length [m]
H = 1;      % height [m]

% Square example
% Cable lengths in [m]
cableLengths = [0.7 0.9 1.0 0.8];
% Cable anchor point locations in [m]
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];
% Calculate the resulting end effector position
[position, ~, ~] = cableLengthsToPosition(cableLengths, anchorPoints);
% Display the cable suspended robot
if isnan(position)
    disp('Valid position could not be found')
else
    time = 0;
    plotCables = true;
    animatedPlot(time, position, cableLengths, anchorPoints, plotCables);
end

% Hexagon example
cableLengths = [0.8,0.7,0.2,0.5,0.5,0.6];
anchorPoints = [B/2-(B/2)*(sqrt(3)/2) B/2-B/4 H;
                B/2 0 0.9*H
                B/2+(B/2)*(sqrt(3)/2) B/2-B/4 0.8*H;
                B/2+(B/2)*(sqrt(3)/2) B/2+B/4 H;
                B/2 B H
                B/2-(B/2)*(sqrt(3)/2) B/2+B/4 1.1*H;];
% Calculate the resulting end effector position
[position, activeCables, redundantCables] = cableLengthsToPosition(cableLengths,
anchorPoints);
% Display the cable suspended robot
if isnan(position)
    disp('Valid position could not be found')
else
    time = 0;
    plotCables = true;
    animatedPlot(time, position, cableLengths, anchorPoints, plotCables);
end
% NOTE: solution not guaranteed to use largest number of cables, since only
% one cable clique is processed (there could be a larger clique)


% =========================================================================
%                               Simulation
% =========================================================================

%% DC motor servo control
addpath('simulations')
addpath('utilities')
close all; clear all; clc;

dcMotorServoSim
%% Closed-loop control using tension feedback
addpath('simulations')
addpath('utilities')
close all; clear all; clc;

cableRobotSim_tension


%% Closed-loop control using cable length feedback
addpath('simulations')
addpath('utilities')
close all; clear all; clc;

cableRobotSim_cableLengths
```

```
%% Cable anchorpoint disturbance prediction
addpath('simulations')
addpath('utilities')
addpath('analysis')
close all; clear all; clc;

cableRobotSim_disturbancePrediction
```

Table A2. *esimateDisturbances.m* is a function that estimates the anchor point displacements from a set of simulation or experimental data.

```
function [dist_est] = estimateDisturbances(t, positions, cableLengths, anchorPoints,
timeStamps, method)
% Estimates the cable anchor point disturbances for a cable-suspended
% robot given simluation or experiment data.
%   t: a time vector in [s]
%   positions: the Cartesian positions in [m] associated with the times [t]
%   cableLengths: the cable lengths in [m] associated with the times [t]
%   anchorPoints: locations of the anchor point locations in [m]
%   timeStamps: the time instances used in the solving method [s]
%   method: selects the solving method. See the switch statement below.
%   dist_est: the estimated anchorpoint disturbances, one row for each
%       cable, each in the Cartesian form [dx,dy,dz], in [m]

    % Number of time points to sample
    nTimes = length(timeStamps);

    % Number of cables
    [nCables,~] = size(anchorPoints);
    dist_est = NaN(nCables, 3);

    % Initialize matrices for methods 1 to 4
    if method<5
        A = NaN(nTimes*nCables,4);
        B = NaN(nTimes*nCables,1);
    end

    % Initialize matrices for method 5
    if method==5
        P = NaN(nTimes*nCables,3);
        L = NaN(nTimes*nCables,1);
    end

    % Initialize matrices for method 6
    if method==6
        res = 0.001;        % grid unit size in [m]
        maxDist = 0.05;     % maximum allowable disturbance [m]
        maxVal = 0;
        n = 2*ceil(maxDist/res)+1;
        gridMaxDist = res*(n+1)/2;
        grid = zeros(n,n,n*nTimes);
    end

    % Loop through timeStamps
    for i=1:nTimes
        % Find the index number for the current time sample
        sampleIndex = find(t>=timeStamps(i),1,'first');

        % Extract data for current time sample
        x = positions(sampleIndex,1);
        y = positions(sampleIndex,2);
        z = positions(sampleIndex,3);
        l = cableLengths(sampleIndex,1:4);

        % Populate matrices to solve for disturbances
        if method<=5
            % Calculate ideal cable lengths, assuming no disturbances
            l_i = positionToCableLengths([x,y,z], anchorPoints);
            % For methods 1 to 4:
            %   A*disturbance = B
            % For method 5:
            %   P=cartesian location, L=cable length
            for c=1:nCables
                matrixRow = (c-1)*nTimes + i;
                ancX = anchorPoints(c,1);
                ancY = anchorPoints(c,2);
                ancZ = anchorPoints(c,3);
                A(matrixRow,:) = [1 -2*(x-ancX) -2*(y-ancY) -2*(z-ancZ)];
                B(matrixRow) = l(c)^2 - l_i(c)^2;
                P(matrixRow,:) = [x,y,z];
                L(matrixRow) = l(c);
            end

        elseif method==6
            % For method 6 (discrete grid approximation)
            for gx=1:n
                for gy=1:n
                    for gz=1:n
                        for c=1:nCables
                            dx = anchorPoints(c,1) - gridMaxDist + res*gx;
                            dy = anchorPoints(c,2) - gridMaxDist + res*gy;
                            dz = anchorPoints(c,3) - gridMaxDist + res*gz;
                            dist = norm([dx-x, dy-y, dz-z]);
                            if abs(dist-l(c))<res/2
                                gz0 = (c-1)*n;
                                grid(gx,gy,gz+gz0) = grid(gx,gy,gz+gz0) + 1;
                            end
                        end
                    end
                end
            end
        end
    end

    % Solve using selected method
    switch method
        case 1  % Method 1: Solve as a set of linear equations using pseudo inverse
            for c=1:nCables
                matrixRows = (c-1)*nTimes+1:(c-1)*nTimes+nTimes;
                est = pinv(A(matrixRows,:))*B(matrixRows);
                % Keep last three elements only
                dist_est(c,:) = est(2:4)';
            end

        case 2  % Method 2: Solve as a set of linear equations using pseudo inverse
                % Remove smallest singular value
            for c=1:nCables
                matrixRows = (c-1)*nTimes+1:(c-1)*nTimes+nTimes;
                [~,sVals,~] = svd(A(matrixRows,:));
                tol = (sVals(end)+sVals(end-1))/2;
                est = pinv(A(matrixRows,:), tol)*B(matrixRows);
                % Keep last three elements only
                dist_est(c,:) = est(2:4)';
            end

        case 3  % Method 3: Solve as least squares problem using linear equations
            x0 = [0,0,0,0];
            ub = [0.0075,0.05,0.05,0.05];
            lb = [0,-0.05,-0.05,-0.05];
            for c=1:nCables
                matrixRows = (c-1)*nTimes+1:(c-1)*nTimes+nTimes;
                fun = @(x) norm(B(matrixRows)-A(matrixRows,:)*x');
                est = fmincon(fun,x0,[],[],[],[],lb,ub);
                % Keep last three elements only
                dist_est(c,:) = est(2:4)';
            end

        case 4  % Method 4: Solve as least squares problem using nonlinear equations
            x0 = [0,0,0];
            lb = [-0.05,-0.05,-0.05];
            ub = [0.05,0.05,0.05];
            for c=1:nCables
                matrixRows = (c-1)*nTimes+1:(c-1)*nTimes+nTimes;
                fun = @(x) norm(B(matrixRows)-A(matrixRows,2:4)*x'-norm(x)^2);
                dist_est(c,:) = fmincon(fun,x0,[],[],[],[],lb,ub);
            end

        case 5  % Method 5: Solve as minimization problem using graphical method
            for c=1:nCables
                x0 = anchorPoints(c,:);
                lb = x0 - 0.05*[1 1 1];
                ub = x0 + 0.05*[1 1 1];
                fun = @(x) distanceSum(x,c);
                est = fmincon(fun,x0,[],[],[],[],lb,ub);
                dist_est(c,:) = est' - x0';
            end

        case 6  % Method 6: Discrete graphical method
            for c=1:nCables
                gzRange = (c-1)*n+1:(c-1)*n+n;
                cableGrid = grid(:,:,gzRange);
                maxVal = max(cableGrid,[],'all');
                maxInd = find(cableGrid==maxVal);
                numInd = length(maxInd);
                xIdAvg = 0;
                yIdAvg = 0;
                zIdAvg = 0;
                for i=1:numInd
                    [xId,yId,zId] = ind2sub(size(cableGrid),maxInd(i));
                    xIdAvg = xIdAvg + xId/numInd;
                    yIdAvg = yIdAvg + yId/numInd;
                    zIdAvg = zIdAvg + zId/numInd;
                end
                xest = anchorPoints(c,1)-gridMaxDist+res*xIdAvg;
                yest = anchorPoints(c,2)-gridMaxDist+res*yIdAvg;
                zest = anchorPoints(c,3)-gridMaxDist+res*zIdAvg;
                dist_est(c,:) = [xest,yest,zest]-anchorPoints(c,:);
            end
    end

    % For use with method 5
    function [distSum] = distanceSum(x,c)
        distSum = 0;
        zeroMatrixRow = (c-1)*nTimes;
        for point=1:nTimes
            row = zeroMatrixRow+point;
            distSum = distSum + abs(norm(x-P(row,:))-L(row));
        end
    end
end
```

Table A3. *minMaxForceAnalysis.m* is a script that that calculates the allowable *x*, *y*, and *z* forces in a four-cable CSR such that positive cable tensions are maintained.

```
% Force analysis, 4 cables
% Calculates, for set of points:
%   1. The minimum z force given an x-y force
%   2. The maximum x-y force given a z force
clear all;


% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Square shape with side lengths B)
% ====================

%            d_________________c
%           /|                /|
%         a/_|______________b/ |
%         |  |              |  |
%         |  |              |  |  H
%         |  |              |  |
%         |  |              |  |
%         |  |______________|__|
%         | /               | /  B
%         |/________________|/
%                 B


% ====================
% Define some variables
% ====================
global B H anchorPoints

% Anchor point dimensions (see physical configuration above)
B = 1.0;        % length of each side [m]
H = 1.0;        % height of cable anchor points [m]
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];    % anchor points for 4 cables

% Mass of the end effector in [kg]
mass = 1;
g = 9.81;       % gravity acceleration [m/s^2]

% ====================
% Run analysis
% ====================

% Select point in the CSR workspace
x = B/4;
y = 3*B/4;
z = 0.3;
position = [x,y,z];

% Create the cable direction matrix (for some analysis)
[numCables, ~] = size(anchorPoints);
cableDirections = NaN(3,numCables);
for c=1:numCables
    dx = anchorPoints(c,1)-position(1);
    dy = anchorPoints(c,2)-position(2);
    dz = anchorPoints(c,3)-position(3);
    magnitude = norm([dx dy dz]);
    cableDirections(:,c) = [dx;dy;dz]/magnitude;

    zz = norm(cableDirections(1:2,c),2)/cableDirections(3,c);
    xx = cableDirections(3,c)/cableDirections(1,c);
end
```

```
% Vary the x-y force and calculate the minimum z force
n = 21;
Fx = NaN(n*n,1);
Fy = NaN(n*n,1);
minFz = NaN(n*n,1);
f = linspace(-2*mass*g,2*mass*g,n);
sampleIndex = 0;
for i=1:length(f)
    for j=1:length(f)
        % Increase sample index number
        sampleIndex = sampleIndex+1;

        % Calculate x,y,z location in [m] based on sample number
        Fx(sampleIndex) = f(i);
        Fy(sampleIndex) = f(j);

        % Solve for the tensions using an optimization method
        [Fz, ~] = positionToMinZForce(anchorPoints, position, Fx(sampleIndex),
Fy(sampleIndex));
        minFz(sampleIndex) = Fz;
    end
end
% Plot the results in a 3D figure
figure(1);
surf(reshape(100*Fx/(mass*g),[n,n]),reshape(100*Fy/(mass*g),[n,n]),reshape(100*minFz/(
mass*g),[n,n]));
grid on;
xlabel('F_x [% weight]')
ylabel('F_y [% weight]')
zlabel('Minimum F_z [% weight]')


% Vary the z force and calculate the maximum x-y force magnitude
n = 101;
Fz = NaN(n,1);
maxFxy = NaN(n,1);
f = linspace(0,2*mass*g,n);
sampleIndex = 0;
for i=1:length(f)
    % Increase sample index number
    sampleIndex = sampleIndex+1;

    % Calculate x,y,z location in [m] based on sample number
    Fz(sampleIndex) = f(i);

    % Solve for the tensions using an optimization method
    [Fxy, ~] = positionToMaxXYForce(anchorPoints, position, Fz(sampleIndex));
    maxFxy(sampleIndex) = Fxy;
end
% Plot the results in a 2D plot
figure(2);
plot(100*Fz/(mass*g),100*maxFxy/(mass*g),'k.')
grid on;
xlabel('F_z [% weight]')
ylabel('Maximum F_{xy} [% weight]')
```

Table A4. *plotTensionFunctions.m* is a script that that plots the cable tension versus cable stretch amount. Several cable models are tested.

```
% Plot cable tension functions
%   Four different tension functions are plotted. The function is changed
%   using the 'fun' variable below.
clear all;


% ====================
% Select cable lengths to try
% ====================

% Three different unstretched cable lengths to use in calculations.
%   The tension functions calculate cable tension based on unstretched and
%   stretched cable lengths.
l0 = [0.1 0.5 1];


% ====================
% Calculate and plot cable tension (and stiffness)
% ====================

% Calculate tension and stiffness for each sample point, tension function 1
fun = 1;
figure()
subplot(1,2,1)
for u=1:length(l0)  % loop through the unstretched cable lengths
    unstretchedLength = l0(u);

    % Create array of stretched cable lengths
    l = linspace(0.75*unstretchedLength, 1.2*unstretchedLength);

    % Calculate the tensions and stiffness for each stretched cable length
    tension = NaN(length(l),1);
    stiffness = NaN(length(l),1);
    for s=1:length(l)  % loop through stretched cable lengths
        stretchedLength = l(s);
        % Call the tension function
        [tension(s),stiffness(s)] = tensionFunction(fun, stretchedLength,
unstretchedLength, []);
    end

    % Plot tension vs cable strech
    subplot(1,2,1)
    plot(l, tension);
    title('Constant stiffness, no negative tension allowed')
    xlabel('Stretched cable length [m]')
    ylabel('Tension [N]')
    grid on
    hold on

    % Plot stiffness vs cable strech
    subplot(1,2,2)
    plot(l, stiffness);
    xlabel('Stretched cable length [m]')
    ylabel('Stiffness [N/m]')
    grid on
    hold on
end
% Add legend for three unstretched cable lengths
leg = legend('L0=0.1 m','L0=0.5m','L0=1.0m');
title(leg,'Unstretched cable length')
```

```
% Calculate tension and stiffness for each sample point, tension function 2
fun = 2;
figure()
subplot(1,2,1)
for u=1:length(l0)  % loop through the unstretched cable lengths
    unstretchedLength = l0(u);

    % Create array of stretched cable lengths
    l = linspace(0.75*unstretchedLength, 1.2*unstretchedLength);

    % Calculate the tensions and stiffness for each stretched cable length
    tension = NaN(length(l),1);
    stiffness = NaN(length(l),1);
    for s=1:length(l)  % loop through stretched cable lengths
        stretchedLength = l(s);
        % Call the tension function
        [tension(s),stiffness(s)] = tensionFunction(fun, stretchedLength,
unstretchedLength, []);
    end

    % Plot tension vs cable strech
    subplot(1,2,1)
    plot(l, tension);
    title('Rod stiffness')
    xlabel('Stretched cable length [m]')
    ylabel('Tension [N]')
    grid on
    hold on

    % Plot stiffness vs cable strech
    subplot(1,2,2)
    plot(l, stiffness);
    xlabel('Stretched cable length [m]')
    ylabel('Stiffness [N/m]')
    grid on
    hold on
end
% Add legend for three unstretched cable lengths
leg = legend('L0=0.1 m','L0=0.5m','L0=1.0m');
title(leg,'Unstretched cable length')


% Calculate tension and stiffness for each sample point, tension function 3
fun = 3;
figure()
subplot(1,2,1)
for u=1:length(l0)  % loop through the unstretched cable lengths
    unstretchedLength = l0(u);

    % Create array of stretched cable lengths
    l = linspace(0.75*unstretchedLength, 1.2*unstretchedLength);

    % Calculate the tensions and stiffness for each stretched cable length
    tension = NaN(length(l),1);
    stiffness = NaN(length(l),1);
    for s=1:length(l)  % loop through stretched cable lengths
        stretchedLength = l(s);
        % Call the tension function
        [tension(s),stiffness(s)] = tensionFunction(fun, stretchedLength,
unstretchedLength, []);
    end

    % Plot tension vs cable strech
    subplot(1,2,1)
    plot(l, tension);
    title('Rod stiffness, no negative tension allowed')
    xlabel('Stretched cable length [m]')
    ylabel('Tension [N]')
    grid on
    hold on

    % Plot stiffness vs cable strech
    subplot(1,2,2)
    plot(l, stiffness);
    xlabel('Stretched cable length [m]')
    ylabel('Stiffness [N/m]')
    grid on
    hold on
end
% Add legend for three unstretched cable lengths
leg = legend('L0=0.1 m','L0=0.5m','L0=1.0m');
title(leg,'Unstretched cable length')


% Calculate tension and stiffness for each sample point, tension function 4
fun = 4;
figure()
subplot(1,2,1)
for u=1:length(l0)  % loop through the unstretched cable lengths
    unstretchedLength = l0(u);

    % Create array of stretched cable lengths
    l = linspace(0.75*unstretchedLength, 1.2*unstretchedLength);

    % Calculate the tensions and stiffness for each stretched cable length
    tension = NaN(length(l),1);
    stiffness = NaN(length(l),1);
    for s=1:length(l)  % loop through stretched cable lengths
        stretchedLength = l(s);
        % Call the tension function
        [tension(s),stiffness(s)] = tensionFunction(fun, stretchedLength,
unstretchedLength, []);
    end

    % Plot tension vs cable strech
    subplot(1,2,1)
    plot(l, tension);
    title('Modified rod stiffness, small tension for small slack')
    xlabel('Stretched cable length [m]')
    ylabel('Tension [N]')
    grid on
    hold on

    % Plot stiffness vs cable strech
    subplot(1,2,2)
    plot(l, stiffness);
    xlabel('Stretched cable length [m]')
    ylabel('Stiffness [N/m]')
    grid on
    hold on
end
% Add legend for three unstretched cable lengths
leg = legend('L0=0.1 m','L0=0.5m','L0=1.0m');
title(leg,'Unstretched cable length')
```

Table A5. *stiffnessAnalysis_3Cable.m* is a script that that calculates and plots the directional stiffness of a three-cable CSR at a given height.

```
% Stiffness analysis
clear all;


% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Equilateral triangle that fits within a BxB square)
% ====================

%                    c
%                    |
%         a          |      b
%         |          |      |
%         |          |      |
%         |          |      | H
%         |          |      |
%         |          |      |
%         |                 |
%         |                 |
%


% ====================
% Define some variables
% ====================
global B H anchorPoints mass

% Anchor point dimensions (see physical configuration above)
B = 1.0;        % length of each side [m]
H = 1.0;        % height of cable anchor points [m]
anchorPoints = [B/2-(B/2)*(sqrt(3)/2) B/2-B/4 H;
                B/2+(B/2)*(sqrt(3)/2) B/2-B/4 H;
                B/2 B H];    % anchor points for 3 cables

% Mass of the end effector in [kg]
mass = 1;


% ====================
% Create sample points and run analysis
% ====================

% Create evenly-spaced grid sample points at a single height
n = 25;                 % number of points per grid dimension
x = NaN(n*n,1);
y = NaN(n*n,1);
z = 0.3*ones(n*n,1);    % height [m]
sampleIndex = 0;
for i=1:n
    for j=1:n
        % Increase sample index number
        sampleIndex = sampleIndex+1;

        % Calculate x,y,z location in [m] based on sample number
        x(sampleIndex) = i*B/(n+1);
        y(sampleIndex) = j*B/(n+1);
    end
end
% Solve and plot sampling of tensions
method = 2;
plotOption = 1;
plotStiffness([x y z], method, plotOption);
```

```
% ====================
% Solve and plot function
% ====================
function plotStiffness(samplePoints, method, plotOption)
    % Access some global variables
    global B H anchorPoints mass

    % Initialize arrays to store results
    [numSamples,~] = size(samplePoints);    % number of samples
    O = NaN(numSamples,1);
    % Position
    X = O;
    Y = O;
    Z = O;
    % Stiffness magnitudes
    Kx = O;
    Ky = O;
    Kz = O;

    % Calculate the stiffness for each sample location
    for i=1:numSamples
        % Calculate tension at the location
        position = [samplePoints(i,1) samplePoints(i,2) samplePoints(i,3)];
        K = positionToStiffness(position, anchorPoints, mass, method);

        % Store position
        X(i) = position(1);
        Y(i) = position(2);
        Z(i) = position(3);

        % Store stiffness values
        Kx(i) = K(1);
        Ky(i) = K(2);
        Kz(i) = K(3);
    end

    switch plotOption
        % Plot cable stiffness, seperate plot for each direction
        case 1
            figure('Name',sprintf('Method %i',method))
            % x plot
            subplot(1,3,1);
            plot3(X,Y,Kx,'.k');
            title('X stiffness')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Stiffness [N/m]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square
            % y plot
            subplot(1,3,2);
            plot3(X,Y,Ky,'.k');
            title('Y stiffness')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Stiffness [N/m]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square
            % z plot
            subplot(1,3,3);
            plot3(X,Y,Kz,'.k');
            title('Z stiffness')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Stiffness [N/m]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square

        % Plot cable tensions, all tension vectors together in one plot
        case 2
            O = zeros(numSamples,1);
            figure()
            subplot(1,1,1)
            hold on
            quiver3(X,Y,Z, Kx,O,O, 'r');
            quiver3(X,Y,Z, O,Ky,O, 'g');
            quiver3(X,Y,Z, O,O,Kz, 'b');
            hold off
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square
    end
end
```

Table A6. *stiffnessAnalysis_4Cable.m* is a script that that calculates and plots the directional stiffness of a four-cable CSR at a given height.

```
% Stiffness analysis, 4 cables
clear all;


% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Square shape with side lengths B)
% ====================

%            d_________________c
%           /|                /|
%         a/_|______________b/ |
%         |  |              |  |
%         |  |              |  |  H
%         |  |              |  |
%         |  |              |  |
%         |  |______________|__|
%         | /               | /  B
%         |/________________|/
%                 B


% ====================
% Define some variables
% ====================
global B H anchorPoints mass
```

```matlab
% Anchor point dimensions (see physical configuration above)
B = 1.0;        % length of each side [m]
H = 1.0;        % height of cable anchor points [m]
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];    % anchor points for 4 cables

% Mass of the end effector in [kg]
mass = 1;


% ====================
% Create sample points and run analysis
% ====================

% Create evenly-spaced grid sample points at a single height
n = 21;                 % number of points per grid dimension
x = NaN(n*n,1);
y = NaN(n*n,1);
z = 0.3*ones(n*n,1);    % height [m]
sampleIndex = 0;
for i=1:n
    for j=1:n
        % Increase sample index number
        sampleIndex = sampleIndex+1;

        % Calculate x,y,z location in [m] based on sample number
        x(sampleIndex) = i*B/(n+1);
        y(sampleIndex) = j*B/(n+1);
    end
end
% Solve and plot sampling of tensions
method = 2;
plotOption = 1;
plotStiffness([x y z], method, plotOption);


% ====================
% Solve and plot function
% ====================
function plotStiffness(samplePoints, method, plotOption)
    % Access some global variables
    global B H anchorPoints mass

    % Initialize arrays to store results
    [numSamples,~] = size(samplePoints);    % number of samples
    O = NaN(numSamples,1);
    % Position
    X = O;
    Y = O;
    Z = O;
    % Stiffness magnitudes
    Kx = O;
    Ky = O;
    Kz = O;

    % Calculate the stiffness for each sample location
    for i=1:numSamples
        % Calculate tension at the location
        position = [samplePoints(i,1) samplePoints(i,2) samplePoints(i,3)];
        K = positionToStiffness(position, anchorPoints, mass, method);

        % Store position
        X(i) = position(1);
        Y(i) = position(2);
        Z(i) = position(3);

        % Store stiffness values
        Kx(i) = K(1);
        Ky(i) = K(2);
        Kz(i) = K(3);
    end

    switch plotOption
        % Plot cable stiffness, seperate plot for each direction
        case 1
            figure('Name',sprintf('Method %i',method))
            % x plot
            subplot(1,3,1);
            plot3(X,Y,Kx,'.k');
            title('X stiffness')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Stiffness [N/m]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square
            % y plot
            subplot(1,3,2);
            plot3(X,Y,Ky,'.k');
            title('Y stiffness')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Stiffness [N/m]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square
            % z plot
            subplot(1,3,3);
            plot3(X,Y,Kz,'.k');
            title('Z stiffness')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Stiffness [N/m]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square

        % Plot cable tensions, all tension vectors together in one plot
        case 2
            O = zeros(numSamples,1);
            figure()
            subplot(1,1,1)
            hold on
            quiver3(X,Y,Z, Kx,O,O, 'r');
            quiver3(X,Y,Z, O,Ky,O, 'g');
            quiver3(X,Y,Z, O,O,Kz, 'b');
            hold off
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square
    end
end
```

Table A7. *stiffnessAnalysis_6Cable.m* is a script that that calculates and plots the directional stiffness of a six-cable CSR at a given height.

```
% Stiffness analysis
clear all;


% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Regular hexagon that fits in a BxB square)
% ====================

%                   e
%             f     |    d
%          a  |     |    | c
%          |  |   b |    ||
%          |  |   | |    ||
%          |  |   | |    || H
%          |  |   | |    ||
%          |  |   | |    ||
%          |      |       |
%          |      |       |
%                 |


% ====================
% Define some variables
% ====================
global B H anchorPoints mass

% Anchor point dimensions (see physical configuration above)
B = 1.0;        % length of each side [m]
H = 1.0;        % height of cable anchor points [m]
anchorPoints = [B/2-(B/2)*(sqrt(3)/2) B/2-B/4 H;
                B/2 0 H
                B/2+(B/2)*(sqrt(3)/2) B/2-B/4 H;
                B/2+(B/2)*(sqrt(3)/2) B/2+B/4 H;
                B/2 B H
                B/2-(B/2)*(sqrt(3)/2) B/2+B/4 H;];    % anchor points for 3 cables

% Mass of the end effector in [kg]
mass = 1;


% ====================
% Create sample points and run analysis
% ====================

% Create evenly-spaced grid sample points at a single height
n = 21;                 % number of points per grid dimension
x = NaN(n*n,1);
y = NaN(n*n,1);
z = 0.3*ones(n*n,1);    % height [m]
sampleIndex = 0;
for i=1:n
    for j=1:n
        % Increase sample index number
        sampleIndex = sampleIndex+1;

        % Calculate x,y,z location in [m] based on sample number
        x(sampleIndex) = i*B/(n+1);
        y(sampleIndex) = j*B/(n+1);
    end
end
% Solve and plot sampling of tensions
```

```
method = 2;
plotOption = 1;
plotStiffness([x y z], method, plotOption);


% ====================
% Solve and plot function
% ====================
function plotStiffness(samplePoints, method, plotOption)
    % Access some global variables
    global B H anchorPoints mass

    % Initialize arrays to store results
    [numSamples,~] = size(samplePoints);    % number of samples
    O = NaN(numSamples,1);
    % Position
    X = O;
    Y = O;
    Z = O;
    % Stiffness magnitudes
    Kx = O;
    Ky = O;
    Kz = O;

    % Calculate the stiffness for each sample location
    for i=1:numSamples
        % Calculate tension at the location
        position = [samplePoints(i,1) samplePoints(i,2) samplePoints(i,3)];
        K = positionToStiffness(position, anchorPoints, mass, method);

        % Store position
        X(i) = position(1);
        Y(i) = position(2);
        Z(i) = position(3);

        % Store stiffness values
        Kx(i) = K(1);
        Ky(i) = K(2);
        Kz(i) = K(3);
    end

    switch plotOption
        % Plot cable stiffness, seperate plot for each direction
        case 1
            figure('Name',sprintf('Method %i',method))
            % x plot
            subplot(1,3,1);
            plot3(X,Y,Kx,'.k');
            title('X stiffness')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Stiffness [N/m]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square
            % y plot
            subplot(1,3,2);
            plot3(X,Y,Ky,'.k');
            title('Y stiffness')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Stiffness [N/m]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square
            % z plot
            subplot(1,3,3);
            plot3(X,Y,Kz,'.k');
            title('Z stiffness')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Stiffness [N/m]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square

        % Plot cable tensions, all tension vectors together in one plot
        case 2
            O = zeros(numSamples,1);
            figure()
            subplot(1,1,1)
            hold on
            quiver3(X,Y,Z, Kx,O,O, 'r');
            quiver3(X,Y,Z, O,Ky,O, 'g');
            quiver3(X,Y,Z, O,O,Kz, 'b');
            hold off
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square
    end
end
```

Table A8. *tensionAnalysis_3Cable.m* is a script that that calculates and plots the cable tensions of a three-cable CSR at a given height.

```
% Tension analysis, 3 cables
%   Calculates cable tensions at static equilibrium.
%   Can calculate tensions for other cartesian forces by changing
%   'cartesianForces' in the plotTensions() function below.
clear all;


% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Equilateral triangle that fits within a BxB square)
% ====================

%                    c
%                    |
%          a         |      b
%          |         |      |
%          |         |      |
%          |         |      | H
%          |         |      |
%          |         |      |
%          |                |
%          |                |
%
```

```
% ====================
% Define some variables
% ====================
global B H anchorPoints

% Anchor point dimensions (see physical configuration above)
B = 1.0;        % length of each side [m]
H = 1.0;        % height of cable anchor points [m]
anchorPoints = [B/2-(B/2)*(sqrt(3)/2) B/2-B/4 H;
                B/2+(B/2)*(sqrt(3)/2) B/2-B/4 H;
                B/2 B H];    % anchor points for 3 cables

% Mass of the end effector in [kg]
mass = 1;
g = 9.81;       % gravity acceleration [m/s^2]


% ====================
% Create sample points and run analysis
% ====================

% Create evenly-spaced grid sample points at a single height
global n
n = 21;                 % number of points per grid dimension, global because used in
plotting
x = NaN(n*n,1);
y = NaN(n*n,1);
z = 0.3*ones(n*n,1);    % height [m]
sampleIndex = 0;
for i=1:n
    for j=1:n
        % Increase sample index number
        sampleIndex = sampleIndex+1;

        % Calculate x,y,z location in [m] based on sample number
        x(sampleIndex) = i*B/(n+1);
        y(sampleIndex) = j*B/(n+1);
    end
end
% Solve and plot sampling of tensions
method = 0;     % for 3 cables, the method doesn't matter
cartesianForces = [0;0;mass*g];   % static equilibrium
plotOption = 3;
plotTensions([x y z], cartesianForces, method, plotOption);


% ====================
% Solve and plot function
% ====================
function [] = plotTensions(samplePoints, cartesianForces, method, plotOption)
    % Access some global variables
    global B H anchorPoints
    global n

    % Initialize arrays to store results
    [numSamples,~] = size(samplePoints);    % number of samples
    O = NaN(numSamples,1);
    % Position
    X = O;
    Y = O;
    Z = O;
    % Tension directions
    va = NaN(numSamples,3);
    vb = NaN(numSamples,3);
    vc = NaN(numSamples,3);
    % Tension magnitude
    Ta = 0;
    Tb = 0;
    Tc = 0;
    % Tension sum
    Tsum = 0;

    % Calculate tension for each sample location
    for i=1:numSamples
        % Calculate tension at the location
        position = [samplePoints(i,1) samplePoints(i,2) samplePoints(i,3)];
        [cableTensions,tensionDirections] = positionToTension(position, anchorPoints,
cartesianForces, method);

        % Store position
        X(i) = position(1);
        Y(i) = position(2);
        Z(i) = position(3);

        % Store tension directions
        va(i,:) = tensionDirections(1,:);
        vb(i,:) = tensionDirections(2,:);
        vc(i,:) = tensionDirections(3,:);

        % Store cable tensions
        Ta(i) = cableTensions(1);
        Tb(i) = cableTensions(2);
        Tc(i) = cableTensions(3);

        % Store sum of cable tensions^2
%         Tsum(i) = norm(cableTensions)^2;
        Tsum(i) = sum(cableTensions, 'all');
    end

    switch plotOption
        % Plot cable tensions, seperate plots by cable
        case 1
            figure('Name',sprintf('Method %i',method))
            % cable a plot
            subplot(1,3,1);
            quiver3(X,Y,Z, Ta.*va(:,1),Ta.*va(:,2),Ta.*va(:,3),'k');
            hold on
            [maxVal, maxTensionIndex] = max(Ta);
            avgVal = mean(Ta,'all', 'omitnan');
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable a',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','k')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')
            % cable b plot
            subplot(1,3,2);
            quiver3(X,Y,Z, Tb.*vb(:,1),Tb.*vb(:,2),Tb.*vb(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Tb);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable b',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','k')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')
            % cable c plot
            subplot(1,3,3);
            quiver3(X,Y,Z, Tc.*vc(:,1),Tc.*vc(:,2),Tc.*vc(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Tc);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable c',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','k')

        % Plot cable tensions, all tension vectors together in one plot
        case 2
            figure('Name',sprintf('Method %i',method))
            subplot(1,1,1)
            plot3(-1,-1,-1,'k', -1,-1,-1,'r', -1,-1,-1,'g')    % dummy points for
custom legend
            legend('Cable a','Cable b','Cable
c','AutoUpdate','off','Location','SouthOutside')
            hold on
            quiver3(X,Y,Z, Ta.*va(:,1),Ta.*va(:,2),Ta.*va(:,3), 'k');
            quiver3(X,Y,Z, Tb.*vb(:,1),Tb.*vb(:,2),Tb.*vb(:,3), 'r');
            quiver3(X,Y,Z, Tc.*vc(:,1),Tc.*vc(:,2),Tc.*vc(:,3), 'g');
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','k')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','g')
```

```
        % Plot cable tensions, all tension vectors together in one plot
        % In addition, plot the sum of the tensions in a surface
        case 3
            % Cable tensions
            figure('Name',sprintf('Method %i',method))
            subplot(1,2,1)
            plot3(-1,-1,-1,'k', -1,-1,-1,'r', -1,-1,-1,'g')    % dummy points for
custom legend
            legend('Cable a','Cable b','Cable
c','AutoUpdate','off','Location','SouthOutside')
            title('Tension vectors by cable')
            hold on
            quiver3(X,Y,Z, Ta.*va(:,1),Ta.*va(:,2),Ta.*va(:,3), 'k');
            quiver3(X,Y,Z, Tb.*vb(:,1),Tb.*vb(:,2),Tb.*vb(:,3), 'r');
            quiver3(X,Y,Z, Tc.*vc(:,1),Tc.*vc(:,2),Tc.*vc(:,3), 'g');
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','k')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','g')
            % Tension sums
            subplot(1,2,2)
            surf(reshape(X,[n,n]),reshape(Y,[n,n]),reshape(Tsum,[n,n]));
            title('Tension sum')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Tension [N]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square
    end
end
```

Table A9. *tensionAnalysis_4Cable.m* is a script that that calculates and plots the cable tensions of a four-cable CSR at a given height.

```
% Tension analysis, 4 cables
%   Calculates cable tensions at static equilibrium.
%   Can calculate tensions for other cartesian forces by changing
%   'cartesianForces' in the plotTensions() function below.
clear all;


% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Square shape with side lengths B)
% ====================

%            d_________________c
%           /|                /|
%         a/_|______________b/ |
%         |  |              |  |
%         |  |              |  |  H
%         |  |              |  |
%         |  |              |  |
%         |  |______________|__|
%         | /               | /  B
%         |/________________|/
%                 B


% ====================
% Define some variables
% ====================
global B H anchorPoints

% Anchor point dimensions (see physical configuration above)
B = 1.0;        % length of each side [m]
H = 1.0;        % height of cable anchor points [m]
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];    % anchor points for 4 cables

% Mass of the end effector in [kg]
mass = 1;
g = 9.81;       % gravity acceleration [m/s^2]


% ====================
% Create sample points and run analysis
% ====================

% Create evenly-spaced grid sample points at a single height
global n
n = 21;                 % number of points per grid dimension, global because used in
plotting
x = NaN(n*n,1);
y = NaN(n*n,1);
z = 0.3*ones(n*n,1);    % height [m]
sampleIndex = 0;
for i=1:n
    for j=1:n
        % Increase sample index number
        sampleIndex = sampleIndex+1;

        % Calculate x,y,z location in [m] based on sample number
        x(sampleIndex) = i*B/(n+1);
        y(sampleIndex) = j*B/(n+1);
    end
end
% Solve and plot sampling of tensions
method = 2;
cartesianForces = [0;0;mass*g];   % static equilibrium
plotOption = 3;
plotTensions([x y z], cartesianForces, method, plotOption);
% Solve and plot sampling of tensions
method = 3;
cartesianForces = [0;0;mass*g];   % static equilibrium
plotOption = 3;
plotTensions([x y z], cartesianForces, method, plotOption);
% Solve and plot sampling of tensions
method = 4;
cartesianForces = [0;0;mass*g];   % static equilibrium
plotOption = 3;
plotTensions([x y z], cartesianForces, method, plotOption);


% ====================
% Solve and plot function
% ====================
function [] = plotTensions(samplePoints, cartesianForces, method, plotOption)
    % Access some global variables
    global B H anchorPoints
    global n

    % Initialize arrays to store results
    [numSamples,~] = size(samplePoints);    % number of samples
    O = NaN(numSamples,1);
    % Position
    X = O;
    Y = O;
    Z = O;
    % Tension directions
    va = NaN(numSamples,3);
    vb = NaN(numSamples,3);
    vc = NaN(numSamples,3);
    vd = NaN(numSamples,3);
    % Tension magnitude
    Ta = O;
    Tb = O;
    Tc = O;
    Td = O;
    % Tension sum
    Tsum = O;

    % Calculate tension for each sample location
    for i=1:numSamples
        % Calculate tension at the location
        position = [samplePoints(i,1) samplePoints(i,2) samplePoints(i,3)];
        [cableTensions, tensionDirections] = positionToTension(position, anchorPoints,
cartesianForces, method);

        % Store position
        X(i) = position(1);
        Y(i) = position(2);
        Z(i) = position(3);

        % Store tension directions
        va(i,:) = tensionDirections(1,:);
        vb(i,:) = tensionDirections(2,:);
        vc(i,:) = tensionDirections(3,:);
        vd(i,:) = tensionDirections(4,:);

        % Store cable tensions
        Ta(i) = cableTensions(1);
        Tb(i) = cableTensions(2);
        Tc(i) = cableTensions(3);
        Td(i) = cableTensions(4);

        % Store sum of cable tensions^2
%         Tsum(i) = norm(cableTensions)^2;
        Tsum(i) = sum(cableTensions, 'all');
    end

    switch plotOption
        % Plot cable tensions, seperate plots by cable
        case 1
            figure('Name',sprintf('Method %i',method))
            % cable a plot
            subplot(2,2,1);
            quiver3(X,Y,Z, Ta.*va(:,1),Ta.*va(:,2),Ta.*va(:,3),'k');
            hold on
            [maxVal, maxTensionIndex] = max(Ta);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            avgVal = mean(Ta,'all');
            title({'Cable a',sprintf('Max tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','k')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','r')
            % cable b plot
            subplot(2,2,2);
            quiver3(X,Y,Z, Tb.*vb(:,1),Tb.*vb(:,2),Tb.*vb(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Tb);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable b',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','k')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','r')
            % cable c plot
            subplot(2,2,3);
            quiver3(X,Y,Z, Tc.*vc(:,1),Tc.*vc(:,2),Tc.*vc(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Tc);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable c',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','k')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','r')
            % cable d plot
            subplot(2,2,4);
            quiver3(X,Y,Z, Td.*vd(:,1),Td.*vd(:,2),Td.*vd(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Td);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable d',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','k')

        % Plot cable tensions, all tension vectors together in one plot
        case 2
            figure('Name',sprintf('Method %i',method))
            subplot(1,1,1)
            plot3(-1,-1,-1,'k', -1,-1,-1,'r', -1,-1,-1,'g', -1,-1,-1,'b')    % dummy
points for custom legend
            legend('Cable a','Cable b','Cable c','Cable
d','AutoUpdate','off','Location','SouthOutside')
            hold on
            quiver3(X,Y,Z, Ta.*va(:,1),Ta.*va(:,2),Ta.*va(:,3), 'k');
            quiver3(X,Y,Z, Tb.*vb(:,1),Tb.*vb(:,2),Tb.*vb(:,3), 'r');
            quiver3(X,Y,Z, Tc.*vc(:,1),Tc.*vc(:,2),Tc.*vc(:,3), 'g');
            quiver3(X,Y,Z, Td.*vd(:,1),Td.*vd(:,2),Td.*vd(:,3), 'b');
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','k')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','g')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','b')

        % Plot cable tensions, all tension vectors together in one plot
        % In addition, plot the sum of the tensions in a surface
        case 3
            % Cable tensions
            figure('Name',sprintf('Method %i',method))
            subplot(1,2,1)
            plot3(-1,-1,-1,'k', -1,-1,-1,'r', -1,-1,-1,'g', -1,-1,-1,'b')    % dummy
points for custom legend
            legend('Cable a','Cable b','Cable c','Cable
d','AutoUpdate','off','Location','SouthOutside')
            title('Tension vectors by cable')
            hold on
            quiver3(X,Y,Z, Ta.*va(:,1),Ta.*va(:,2),Ta.*va(:,3), 'k');
            quiver3(X,Y,Z, Tb.*vb(:,1),Tb.*vb(:,2),Tb.*vb(:,3), 'r');
            quiver3(X,Y,Z, Tc.*vc(:,1),Tc.*vc(:,2),Tc.*vc(:,3), 'g');
            quiver3(X,Y,Z, Td.*vd(:,1),Td.*vd(:,2),Td.*vd(:,3), 'b');
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','k')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','g')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','b')
            % Tension sums
            subplot(1,2,2)
            surf(reshape(X,[n,n]),reshape(Y,[n,n]),reshape(Tsum,[n,n]));
            title('Tension sum')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Tension [N]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square
    end
end
```

Table A10. *tensionAnalysis_6Cable.m* is a script that that calculates and plots the cable tensions of a six-cable CSR at a given height.

```
% Tension analysis, 6 cables
%   Calculates cable tensions at static equilibrium.
%   Can calculate tensions for other cartesian forces by changing
%   'cartesianForces' in the plotTensions() function below.
clear all;


% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Regular hexagon that fits in a BxB square)
% ====================

%                   e
%             f     |    d
%          a  |     |    | c
%          |  |   b |    ||
%          |  |   | |    ||
%          |  |   | |    || H
%          |  |   | |    ||
%          |  |   | |    ||
%          |      |       |
%          |      |       |
%                 |
% ====================
% Define some variables
% ====================
global B H anchorPoints

% Anchor point dimensions (see physical configuration above)
B = 1.0;        % length of each side [m]
H = 1.0;        % height of cable anchor points [m]
anchorPoints = [B/2-(B/2)*(sqrt(3)/2) B/2-B/4 H;
                B/2 0 H
                B/2+(B/2)*(sqrt(3)/2) B/2-B/4 H;
                B/2+(B/2)*(sqrt(3)/2) B/2+B/4 H;
                B/2 B H
                B/2-(B/2)*(sqrt(3)/2) B/2+B/4 H;];    % anchor points for 3 cables

% Mass of the end effector in [kg]
mass = 1;
g = 9.81;       % gravity acceleration [m/s^2]


% ====================
% Create sample points and run analysis
% ====================

% Create evenly-spaced grid sample points at a single height
global n
n = 21;                 % number of points per grid dimension, global because used in
plotting
x = NaN(n*n,1);
y = NaN(n*n,1);
z = 0.3*ones(n*n,1);    % height [m]
sampleIndex = 0;
for i=1:n
    for j=1:n
        % Increase sample index number
        sampleIndex = sampleIndex+1;

        % Calculate x,y,z location in [m] based on sample number
        x(sampleIndex) = i*B/(n+1);
        y(sampleIndex) = j*B/(n+1);
    end
end
% Solve and plot sampling of tensions
method = 2;
cartesianForces = [0;0;mass*g];   % static equilibrium
plotOption = 3;
plotTensions([x y z], cartesianForces, method, plotOption);


% ====================
% Solve and plot function
% ====================
function [] = plotTensions(samplePoints, cartesianForces, method, plotOption)
    % Access some global variables
    global B H anchorPoints
    global n

    % Initialize arrays to store results
    [numSamples,~] = size(samplePoints);    % number of samples
    O = NaN(numSamples,1);
    % Position
    X = O;
    Y = O;
    Z = O;
    % Tension directions
    va = NaN(numSamples,3);
    vb = NaN(numSamples,3);
    vc = NaN(numSamples,3);
    vd = NaN(numSamples,3);
    ve = NaN(numSamples,3);
    vf = NaN(numSamples,3);
    % Tension magnitude
    Ta = 0;
    Tb = 0;
    Tc = 0;
    Td = 0;
    Te = 0;
    Tf = 0;
    % Tension sum
    Tsum = 0;

    % Calculate tension for each sample location
    for i=1:numSamples
        % Calculate tension at the location
        position = [samplePoints(i,1) samplePoints(i,2) samplePoints(i,3)];
        [cableTensions, tensionDirections] = positionToTension(position, anchorPoints, 
cartesianForces, method);

        % Store position
        X(i) = position(1);
        Y(i) = position(2);
        Z(i) = position(3);

        % Store tension directions
        va(i,:) = tensionDirections(1,:);
        vb(i,:) = tensionDirections(2,:);
        vc(i,:) = tensionDirections(3,:);
        vd(i,:) = tensionDirections(4,:);
        ve(i,:) = tensionDirections(5,:);
        vf(i,:) = tensionDirections(6,:);

        % Store cable tensions
        Ta(i) = cableTensions(1);
        Tb(i) = cableTensions(2);
        Tc(i) = cableTensions(3);
        Td(i) = cableTensions(4);
        Te(i) = cableTensions(5);
        Tf(i) = cableTensions(6);

        % Store sum of cable tensions^2
%         Tsum(i) = norm(cableTensions)^2;
        Tsum(i) = sum(cableTensions, 'all');
    end

    switch plotOption
        % Plot cable tensions, seperate plots by cable
        case 1
            figure('Name',sprintf('Method %i',method))
            % cable a plot
            subplot(2,3,1);
            quiver3(X,Y,Z, Ta.*va(:,1),Ta.*va(:,2),Ta.*va(:,3),'k');
            hold on
            [maxVal, maxTensionIndex] = max(Ta);
            avgVal = mean(Ta,'all', 'omitnan');
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable a',sprintf('Average tension: %0.3f [N]',avgVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','k')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','r')

stem3(anchorPoints(5,1),anchorPoints(5,2),anchorPoints(5,3),'filled','Color','r')

stem3(anchorPoints(6,1),anchorPoints(6,2),anchorPoints(6,3),'filled','Color','r')
            % cable b plot
            subplot(2,3,2);
            quiver3(X,Y,Z, Tb.*vb(:,1),Tb.*vb(:,2),Tb.*vb(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Tb);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable b',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','k')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','r')

stem3(anchorPoints(5,1),anchorPoints(5,2),anchorPoints(5,3),'filled','Color','r')

stem3(anchorPoints(6,1),anchorPoints(6,2),anchorPoints(6,3),'filled','Color','r')
            % cable c plot
            subplot(2,3,3);
            quiver3(X,Y,Z, Tc.*vc(:,1),Tc.*vc(:,2),Tc.*vc(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Tc);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable c',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','k')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','r')

stem3(anchorPoints(5,1),anchorPoints(5,2),anchorPoints(5,3),'filled','Color','r')
```

```
stem3(anchorPoints(6,1),anchorPoints(6,2),anchorPoints(6,3),'filled','Color','r')
            % cable d plot
            subplot(2,3,4);
            quiver3(X,Y,Z, Td.*vd(:,1),Td.*vd(:,2),Td.*vd(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Td);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable d',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','k')

stem3(anchorPoints(5,1),anchorPoints(5,2),anchorPoints(5,3),'filled','Color','r')

stem3(anchorPoints(6,1),anchorPoints(6,2),anchorPoints(6,3),'filled','Color','r')
            % cable e plot
            subplot(2,3,5);
            quiver3(X,Y,Z, Te.*ve(:,1),Te.*ve(:,2),Te.*ve(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Te);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable e',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','r')

stem3(anchorPoints(5,1),anchorPoints(5,2),anchorPoints(5,3),'filled','Color','k')

stem3(anchorPoints(6,1),anchorPoints(6,2),anchorPoints(6,3),'filled','Color','r')
            % cable f plot
            subplot(2,3,6);
            quiver3(X,Y,Z, Tf.*vf(:,1),Tf.*vf(:,2),Tf.*vf(:,3),'k');
            hold on
            [~, maxTensionIndex] = max(Tf);
            plot3(X(maxTensionIndex),Y(maxTensionIndex),Z(maxTensionIndex),'r.')
            title({'Cable f',sprintf('Maximum tension: %0.3f [N]',maxVal)})
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','r')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','r')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','r')

stem3(anchorPoints(5,1),anchorPoints(5,2),anchorPoints(5,3),'filled','Color','r')

stem3(anchorPoints(6,1),anchorPoints(6,2),anchorPoints(6,3),'filled','Color','k')

        % Plot cable tensions, all tension vectors together in one plot
        case 2
            figure('Name',sprintf('Method %i',method))
            subplot(1,1,1)
            plot3(-1,-1,-1,'k', -1,-1,-1,'r', -1,-1,-1,'g', -1,-1,-1,'b', -1,-1,-
1,'m', -1,-1,-1,'c')     % dummy points for custom legend
            legend('Cable a','Cable b','Cable c','Cable d','Cable e','Cable
f','AutoUpdate','off','Location','SouthOutside')
            hold on
            quiver3(X,Y,Z, Ta.*va(:,1),Ta.*va(:,2),Ta.*va(:,3), 'k');
            quiver3(X,Y,Z, Tb.*vb(:,1),Tb.*vb(:,2),Tb.*vb(:,3), 'r');
            quiver3(X,Y,Z, Tc.*vc(:,1),Tc.*vc(:,2),Tc.*vc(:,3), 'g');
            quiver3(X,Y,Z, Td.*vd(:,1),Td.*vd(:,2),Td.*vd(:,3), 'b');
            quiver3(X,Y,Z, Te.*ve(:,1),Te.*ve(:,2),Te.*ve(:,3), 'm');
            quiver3(X,Y,Z, Tf.*vf(:,1),Tf.*vf(:,2),Tf.*vf(:,3), 'c');
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','k')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','g')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','b')

stem3(anchorPoints(5,1),anchorPoints(5,2),anchorPoints(5,3),'filled','Color','m')

stem3(anchorPoints(6,1),anchorPoints(6,2),anchorPoints(6,3),'filled','Color','c')

        % Plot cable tensions, all tension vectors together in one plot
        % In addition, plot the sum of the tensions in a surface
        case 3
            % Cable tensions
            figure('Name',sprintf('Method %i',method))
            subplot(1,2,1)
            plot3(-1,-1,-1,'k', -1,-1,-1,'r', -1,-1,-1,'g', -1,-1,-1,'b', -1,-1,-
1,'m', -1,-1,-1,'c')     % dummy points for custom legend
            legend('Cable a','Cable b','Cable c','Cable d','Cable e','Cable
f','AutoUpdate','off','Location','SouthOutside')
```

```
            title('Tension vectors by cable')
            hold on
            quiver3(X,Y,Z, Ta.*va(:,1),Ta.*va(:,2),Ta.*va(:,3), 'k');
            quiver3(X,Y,Z, Tb.*vb(:,1),Tb.*vb(:,2),Tb.*vb(:,3), 'r');
            quiver3(X,Y,Z, Tc.*vc(:,1),Tc.*vc(:,2),Tc.*vc(:,3), 'g');
            quiver3(X,Y,Z, Td.*vd(:,1),Td.*vd(:,2),Td.*vd(:,3), 'b');
            quiver3(X,Y,Z, Te.*ve(:,1),Te.*ve(:,2),Te.*ve(:,3), 'm');
            quiver3(X,Y,Z, Tf.*vf(:,1),Tf.*vf(:,2),Tf.*vf(:,3), 'c');
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Z [m]')
            xlim([0 B])
            ylim([0 B])
            zlim([0 H])
            view(-20,20)
            grid on
            axis square

stem3(anchorPoints(1,1),anchorPoints(1,2),anchorPoints(1,3),'filled','Color','k')

stem3(anchorPoints(2,1),anchorPoints(2,2),anchorPoints(2,3),'filled','Color','r')

stem3(anchorPoints(3,1),anchorPoints(3,2),anchorPoints(3,3),'filled','Color','g')

stem3(anchorPoints(4,1),anchorPoints(4,2),anchorPoints(4,3),'filled','Color','b')

stem3(anchorPoints(5,1),anchorPoints(5,2),anchorPoints(5,3),'filled','Color','m')

stem3(anchorPoints(6,1),anchorPoints(6,2),anchorPoints(6,3),'filled','Color','c')
            % Tension sums
            subplot(1,2,2)
            surf(reshape(X,[n,n]),reshape(Y,[n,n]),reshape(Tsum,[n,n]));
            title('Tension sum')
            xlabel('X [m]')
            ylabel('Y [m]')
            zlabel('Tension [N]')
            xlim([0 B])
            ylim([0 B])
            view(-20,20)
            grid on
            axis square
    end
end
```

Table A11. *displayMatrix.m* is a function that plots a graphical representation of a binary matrix (i.e. a matrix with only zeros and ones).

```
function displayMatrix(matrix)
% Creates a figure that displays the given matrix. 1's in the matrix are
% filled with a solid color. 0's are not filled.
%
%   matrix: any square matrix composed of 0's and 1's

    % Size of matrix
    n = length(matrix);

    % Create new figure and format for friendly viewing of the matrix
    figure
    subplot(1,1,1)
    xticks(1:n);
    set(gca,'xaxisLocation','top')
    yticks(1:n);
    yticklabels(n:-1:1);
    xlim([0 n+1])
    ylim([0 n+1])
    hold on
    axis square
    box on

    % Fill 1's in the matrix with a red square
    % 0's are not filled in
    for i=1:n
        for j=1:n
            if matrix(i,j)
                rectangle('Position',[j-1+0.5,-i+n+0.5,1,1],'FaceColor',[1 0 0])
            end
        end
    end
end
```

Table A12. *exampleScript.m* is a script that demonstrates a matrix-ordering algorithm, in relation to the largest-maximal-clique analysis.

```
% Demonstration of matrix ordering to find the largest filled square in a
% symmetric matrix.

close all; clear all; clc;

% Create the original matrix semi-randomly.
% This is a symmetric matrix, with 1's on the diagonal and otherwise
% randomly filled with 1's and 0's.
matrixSize = 20;
% Start with the identity matrix
originalMatrix = eye(matrixSize);
% Fill the upper right triangle with 1's and 0's
% Reflect the upper right values accross to the lower right
for i=1:matrixSize-1
        for j = i+1:matrixSize
            val = round(rand(1)+0.2);
            originalMatrix(i,j) = val;
            originalMatrix(j,i) = val;
        end
end

adjacencyMatrix = originalMatrix;
for i=1:length(adjacencyMatrix)
    adjacencyMatrix(i,i)=0;
end

%% Reorder matrix to give largest square
tic
[reorderedMatrix, squareSizes, order] = findCliques(originalMatrix, true);
sortingTime = toc;

% Display the original and sorted matrices, with some information in the figure titles
displayMatrix(originalMatrix);
title("Original matrix")
displayMatrix(reorderedMatrix);
title({'Reordered matrix',sprintf( 'Largest square size:
%i',max(squareSizes)),sprintf('Algorithm time: %f [s]',sortingTime)})
```

Table A13. *findCliques.m* is a function that is used to find maximal cliques in a network of simply-connected nodes. This function relies on a matrix representation of the nodes.

```
function [newMatrix,cliqueSize,newOrder] = findCliques(originalMatrix, recursion)
% Finds the cliques in a symmetric, binary matrix with 1's on the diagonal.
%
%   originalMatrix: symmetric matrix with 1's on the diagonal and 0's or
%       1's elsewhere. Such a matrix can represent a relationship between
%       nodes in a graph. Cell {i,j} of the matrix is "1" if nodes i and j
%       are related in some way.
%   recursion: boolean that selects whether the algorithm is applied
%       recurseively. If set to true, every clique will be found. If set to
%       false, only the first clique will be found. The first clique is not
%       necessarily the biggest.
%   newMatrix: the original matrix but with rows/solumns reordered to display
%       the clique(s) found
%   squareSizes: array the lists the dimension of each clique found. The
%       clique sizes are in the same order as the newMatrix.
%   order: the order of rows/columns of newMatrix, with respect to the
%       originalMatrix order.


    % Extract some information
    n = length(originalMatrix);     % number of matrix rows (or columns)
    Q = originalMatrix;             % Copy of the original matrix
    I = ones(n);                    % Matrix of ones, same size as original matrix

    % Stop if the input matrix is a single element
    if n==1
        newMatrix = 1;
        cliqueSize = 1;
        newOrder = 1;
        return
    end

    % Main algorithm
    %   Method: remove rows/columns of the original matrix such that
    %   the number of 0's in the new matrix is less than the original matrix.
    %   A clique is found when the remaining matrix has no 0's.
    %
    %   x is a vector that states what rows/columns of the original matrix
    %   are kept in the new matrix. A "0" in position i of x means that
    %   row/column i of the original matrix is removed. A "1" in that
    %   position means that row/column i is kept.
    %
    %   The number of 1's in the new matrix is found by calculating x'*Q*x,
    %   where Q is the original matrix. The number of zeros in the new matrix
    %   is found by calculating x'*I*x - x'*Q*x = x'*(I-Q)*x. A clique is
    %   found when x'*(I-Q)*x is equal to zero. The largest clique is given
    %   by the x with the most 1's such that x'*(I-Q)*x=0.
    %
    %   At each loop of the algorithm J = x'*(I-Q)*x is calculated using a
    %   new candidate x. The candidate x is created by taking the previous
    %   x and changing one of its elements from 1 to 0. The candidate x that
    %   reduces J the greatest is taken as the "best" x because it will
    %   lead to finding a clique.

    x = ones(n,1);  % starting x is all 1's
    for i=1:10000   % number of iteration is arbitrarily capped at 10000
        % Calculate the number of zeros in the new matrix
        J = x'*(I-Q)*x;

        % If there are no zeros in the new matrix, a clique has been found
        if J==0
            % Attempt to make the current clique larger
            % Loop through the elements of x
            for e=1:n
                % If element e of x is a 0
                if x(e)==0
                    % Create a copy of x and store it in y
                    y = x;
                    % Change element e from 0 to 1
                    y(e)=1;
                    % If the number of zeros in the new matrix is still
                    % zero, we have found a larger clique, so we add the
                    % new row to the new matrix
                    if y'*(I-Q)*y==0
                        x(e) = 1;
                    end
                end
            end

            % Extract indices of 1's and 0's in the final x
            a = find(x==1);
            b = find(x==0);

            % If recursion was requested
            if recursion==true
                % If there are rows left over in the original matrix
                if ~isempty(b)
                    % Create a matrix from the remaining rows/columns
                    subMatrix = originalMatrix(b,b);
                    % Send the matrix to the clique algorithm
                    [~,a_sub,b_sub] = findCliques(subMatrix, true);
                    % Append the results of the recursion
                    newOrder = [a; b(b_sub)];
                    cliqueSize = [length(a) a_sub];
                    newMatrix = reorderMatrix(originalMatrix,newOrder);
                % If recursion was requested, but there are no remaining
                % rows. This happens if the entire original matrix is a
                % clique.
                else
                    % The order is just the rows in the current clique
                    newOrder = a;
                    newMatrix = reorderMatrix(originalMatrix,newOrder);
                    cliqueSize = length(a);
                end

            % If recursion was not requested
            else
                % Create matrix order
                newOrder = [a;b];
                % Count clique size
                cliqueSize = length(a);
                % Create new matrix using the new order
                newMatrix = reorderMatrix(originalMatrix,newOrder);
            end

            % Return the solution
            return
        end

        % J is positive, so remove rows to reduce J
        maxJchange = 0;      % The maximum change in J is intially set to 0
        % Loop through the elements of x
        for e=1:n
            % If element e of x is a 1
            if(x(e)==1)
                % Create a copy of x and store it in y
                y = x;
                % Change element e from 1 to 0
                y(e)=0;
```

```
                 % Calculate the change in J using the modified x vector
                 Jchange = J - y'*(I-Q)*y;

                 % If Jchange is larger than the maximum J change
                 if Jchange>maxJchange
                     % Update the maximum J change
                     maxJchange = Jchange;
                     % Update the best x
                     bestx = y;
                 end
             end
         end
         % Save the best x for the next loop iteration
         x = bestx;
     end
end
```

Table A14. *reorderMatrix.m* is a helper function that is used to reorder the columns and rows of a given matrix.

```
function [newMatrix] = reorderMatrix(originalMatrix, newOrder)
% Reorders the colums/rows of the given matrix to achieve the new order.
% Utilizes the swapMatrixorder() function.
%
%   originalMatrix: any square matrix
%   newOrder: vector with the desired matrix order, every matrix row/column
%       number of the original matrix should appear exactly once
%   newMatrix: returned matrix, same as originalMatrix but with the
%       rows/columns ordered according to newOrder

    % Copy original matrix
    newMatrix = originalMatrix;

    % The starting order is a vector from 1 to matrix size
    order = 1:length(originalMatrix);

    % Swap matrix rows/columns until the desired order is reached
    for i=1:length(order)
        % Loop through current order, an compare to new order
        if order(i)~=newOrder(i)
            % Location of current row/column (that is NOT the desired one)
            a = i;
            % Location of wanted row/column (that is the desired one)
            b = find(order==newOrder(i),1);
            % Swap order in the matrix so that current row is the desired one
            newMatrix = swapMatrixOrder(newMatrix, a, b);
            % Update the current matrix order
            order(b) = order(i);
            order(i) = newOrder(i);
        end
    end
end
```

Table A15. *swapMatrixOrder.m* is a helper function that is used to swap two columns/rows of a given matrix.

```
function [newMatrix] = swapMatrixOrder(originalMatrix, a, b)
% Swaps the rows and columns a and b in the given matrix.
%
```

```
%   originalMatrix: any square matrix
%   a: any row or column number of the original matrix, row number 1 is the
%       top row and column number 1 is the left-most column
%   b: any row or column number of the original matrix
%   newMatrix: returned matrix, same as originalMatrix but with row and column
%       a and b swapped

    % Copy the original matrix
    newMatrix = originalMatrix;
    % Swap rows a and b of the new matrix
    newMatrix([a b],:) = newMatrix([b a],:);
    % Swap columns a and b of the new matrix
    newMatrix(:,[a b]) = newMatrix(:,[b a]);
end
```

Table A16. *cableRobotSim_cableLengths.m* is a script that runs simulations of a four-cable CSR using cable-length-based control with camera-position feedback.

```
% Cable Robot Simulation
close all; clear all;


% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Square shape with side lengths B)
% ====================

%         Z
%         |      |_camera_|
%         |         |_|
%         |  d_________________c
%         | /|                /|
%         a/_|______________b/ |
%         |  |              |  |
%         |  |  Y           |  |   H
%         |  | /            |  |
%         |  |/             |  |
%         |  |______________|__|
%         | /               | /   B
%         |/________________|/___________ X
%                 B


% ====================
% Initialize variables
% ====================

% Anchor point dimensions (see physical configuration above)
global B H anchorPoints
B = 0.75;               % length of each side [m]
H = 0.795;              % height of support [m]
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];    % anchor points for 4 cables

% Mass dynamics
global m c g
m = 1;                  % mass [kg]
c = 2;                  % viscous damping [Ns/m]
g = 9.81;               % gravity  [m/s^2]

% Cable length servo dynamics
global wn zeta
wn = 50;                % natural frequency [rad/s] of second-order length response
zeta = 0.9;             % damping factor of second-order length response
```

```
% ====================
% Disturbances
% ====================

% Camera position error
global x_camError y_camError z_camError;
x_camError = 0.00;           % x measurement error [m]
y_camError = 0.00;           % y measurement error [m]
z_camError = 0.00;           % z measurement error [m]
% x_camError = 0.05;         % x measurement error [m]
% y_camError = 0.02;         % y measurement error [m]
% z_camError = -0.03;        % z measurement error [m]

% Encoder cable length error
global la_error lb_error lc_error ld_error;
la_error = 0.00;             % cable a length error [m]
lb_error = 0.00;             % cable b length error [m]
lc_error = 0.00;             % cable c length error [m]
ld_error = 0.00;             % cable d length error [m]
% la_error = 0.03;           % cable a length error [m]
% lb_error = 0.000;          % cable b length error [m]
% lc_error = -0.01;          % cable c length error [m]
% ld_error = 0.02;           % cable d length error [m]

% Anchorpoint position errors
global da db dc dd
% da = [0.01, -0.02, 0];
% db = [-0.03, 0, 0.01];
% dc = [0.01, 0.01, 0];
% dd = [0, 0.02, 0.005];
% Create random [x, y, z] disturbance for each cable anchor point
maxDist = 0.00;         % maximum disturbance amount [m]
da = round(2*(rand(1,3)-0.5)*maxDist,4);
db = round(2*(rand(1,3)-0.5)*maxDist,4);
dc = round(2*(rand(1,3)-0.5)*maxDist,4);
dd = round(2*(rand(1,3)-0.5)*maxDist,4);

% ====================
% Simulate and plot
% ====================

% States
% x = [x,y,z, vx,vy,vz, la_enc,lb_enc,lc_enc,ld_enc, la_dot,lb_dot,lc_dot,ld_dot];
%   = [position, velocity, cable lengths according to encoder, cable length time
derivatives, camera cable length error integrals]
x0 = B/2;
y0 = B/2;
z0 = B;
cableLengths = positionToCableLengths([x0,y0,z0], anchorPoints);
la0 = cableLengths(1);
lb0 = cableLengths(2);
lc0 = cableLengths(3);
ld0 = cableLengths(4);
X0 = [x0,y0,z0, 0,0,0, la0,lb0,lc0,ld0, 0,0,0,0, 0,0,0,0];

% Run simulation
tf = 40;
t = [0 tf];

[t,x] = ode45(@dynamics, t, X0);

% Clip simulation
tstart = 10;
ind = find(t>=tstart,1,'first');
t = t(ind:end);
x = x(ind:end, :);
```

```
% Plot result
basicPlot(t,x);
positions = x(:,1:3);
cableLengths = x(:,7:10)+[la_error,lb_error,lc_error,ld_error];
trueAnchorPoints = anchorPoints+[da; db; dc; dd];
plotCables = false;
animatedPlot(t, positions, cableLengths, trueAnchorPoints, plotCables);
% Add reference curves to the animated plot
% theta = linspace(0,2*pi);
% xref = 0.75/2+0.25*sin(theta);
% yref = 0.75/2+0.25*cos(theta);
% zref = 0.1*ones(length(theta),1);
[xref, yref, zref] = spiral(B/2, B/2, B/3, B, 0.1, t);
plot3(xref,yref,zref,'r');
plot3(xref-x_camError,yref-y_camError,zref-z_camError,'g');
% Add some fake points for custom legend
h = zeros(2, 1);
h(1) = plot(NaN,NaN,'-r');
h(2) = plot(NaN,NaN,'-g');
legend(h,'True reference path','Reference path according to
camera','Location','southoutside');

% % Plot resulting cable tensions
% plotCableTensions(t,x)


% ====================
% Model
% ====================

function dx = dynamics(t,x)
    % Select controller
    controller = 2;

    % Access global variables
    global B H anchorPoints
    global m c g wn zeta
    global x_camError y_camError z_camError
    global la_error lb_error lc_error ld_error
    global da db dc dd

    % True anchor points
    trueAnchorPoints = anchorPoints+[da; db; dc; dd];

    % True unstretched cable lengths
    la = x(7) + la_error;
    lb = x(8) + lb_error;
    lc = x(9) + lc_error;
    ld = x(10) + ld_error;

    % Mass position as measured by camera
    x_cam = x(1) + x_camError;
    y_cam = x(2) + y_camError;
    z_cam = x(3) + z_camError;

    % Cable lengths estimated from camera position measurements
    cableLengths = positionToCableLengths([x_cam,y_cam,z_cam], anchorPoints);
    la_cam = cableLengths(1);
    lb_cam = cableLengths(2);
    lc_cam = cableLengths(3);
    ld_cam = cableLengths(4);

    % Cable lengths calculated from true mass position
    cableLengths = positionToCableLengths([x(1),x(2),x(3)], trueAnchorPoints);
    FL = 0.025;
    la_m = cableLengths(1) + FL/2 + (FL/2)*cos(t*2*pi*0.1);
```

```
lb_m = cableLengths(2);
lc_m = cableLengths(3) + FL/2 + (FL/2)*sin(t*2*pi*0.1);
ld_m = cableLengths(4);

% Calculate the cable tensions based on cable stretch amount
tensionFunctionSelect = 3;
[Ta,~] = tensionFunction(tensionFunctionSelect, la_m, la, []);
[Tb,~] = tensionFunction(tensionFunctionSelect, lb_m, lb, []);
[Tc,~] = tensionFunction(tensionFunctionSelect, lc_m, lc, []);
[Td,~] = tensionFunction(tensionFunctionSelect, ld_m, ld, []);

% Resolve cable tensions into Cartesian components
TaC = Ta*[-x(1), -x(2), H-x(3)]/la_m;
TbC = Tb*[B-x(1), -x(2), H-x(3)]/lb_m;
TcC = Tc*[B-x(1), B-x(2), H-x(3)]/lc_m;
TdC = Td*[-x(1), B-x(2), H-x(3)]/ld_m;
Tx = TaC(1) + TbC(1) + TcC(1) + TdC(1);
Ty = TaC(2) + TbC(2) + TcC(2) + TdC(2);
Tz = TaC(3) + TbC(3) + TcC(3) + TdC(3);

% Get reference position
[xr, yr, zr] = spiral(B/2, B/2, B/3, B, 0.1, t);

% Convert reference position to reference cable lengths
cableLengths = positionToCableLengths([xr,yr,zr], anchorPoints);
la_r = cableLengths(1);
lb_r = cableLengths(2);
lc_r = cableLengths(3);
ld_r = cableLengths(4);

% Calculate cable length errors (reference minus camera estimated)
la_e_cam = la_r - la_cam;
lb_e_cam = lb_r - lb_cam;
lc_e_cam = lc_r - lc_cam;
ld_e_cam = ld_r - ld_cam;

% ------------
% Control laws
% ------------

switch controller
    case 1  % Simple cable length servo without camera feedback
        u = [la_r lb_r lc_r ld_r];

    case 2  % Servo plus integral feedback from camera measurements
        ki = 2.0;
        u = [la_r lb_r lc_r ld_r] + ki*[x(15) x(16) x(17) x(18)];
end

% ------------
% Calculate state derrivatives
% ------------

dx = zeros(18,1);

% Mass dynamics
dx(1) = x(4);
dx(2) = x(5);
dx(3) = x(6);
dx(4) = (Tx - c*x(4))/m;
dx(5) = (Ty - c*x(5))/m;
dx(6) = (Tz - c*x(6))/m - g;

% Cable length dynamics (second order response)
dx(7) = x(11);
dx(8) = x(12);
dx(9) = x(13);
```

```
    dx(10) = x(14);
    dx(11) = u(1)*wn^2 - 2*wn*zeta*x(11) - wn^2*x(7);
    dx(12) = u(2)*wn^2 - 2*wn*zeta*x(12) - wn^2*x(8);
    dx(13) = u(3)*wn^2 - 2*wn*zeta*x(13) - wn^2*x(9);
    dx(14) = u(4)*wn^2 - 2*wn*zeta*x(14) - wn^2*x(10);

    % Camera cable length error dynamics
    dx(15) = la_e_cam;
    dx(16) = lb_e_cam;
    dx(17) = lc_e_cam;
    dx(18) = ld_e_cam;
end


% ====================
% Reference path functions
% ====================

function [x,y,z] = spiral(cx, cy, r, z0, z1, t)
    % cx, cy is the x,y center of the spiral
    % r is the final radius of the circle in [m]
    % z0 is the initial height of the circle [m]
    % z1 is the final height of the circle [m]
    % freq is the frequency of the circle path [Hz]
    % t is the current time in [s]

    % if the time input is a vector, calculate for each time value
    x = zeros(length(t),1);
    y = zeros(length(t),1);
    z = zeros(length(t),1);
    for i=1:1:length(t)
        % growing radius, lowering height
        radius = t(i)/100;
        height = z0-t(i)/10;

        if(radius>=r)
           radius = r;
        end

        if(height<z1)
           height = z1;
        end

        x(i) = cx + radius*cos(t(i)/2);
        y(i) = cy + radius*sin(t(i)/2);
        z(i) = height;
    end
end

function [x,y,z] = helix(cx, cy, r, z0, z1, freq, t_span, t)
    % cx, cy is the x,y center of the helix
    % r is radius of helix in [m]
    % z0 is start height [m]
    % z1 is end height in [m]
    % freq is the frequency of the helix path [Hz]
    % t_span is the time span of the helix trajectory in [s]
    % t is the current time in [s]

    x = cx + r*cos(2*pi*freq*t);
    y = cy + r*sin(2*pi*freq*t);
    z = z0 + z1*(t/t_span);
end


% ====================
% Plotting functions
% ====================
function basicPlot(t,x)
    global B H anchorPoints
    global x_camError y_camError z_camError
    global la_error lb_error lc_error ld_error

    [xr, yr, zr] = spiral(B/2, B/2, B/3, B, 0.1, t);
    cableLengths = positionToCableLengths([xr, yr, zr], anchorPoints);
    la_r = cableLengths(:,1);
    lb_r = cableLengths(:,2);
    lc_r = cableLengths(:,3);
    ld_r = cableLengths(:,4);

    % Plot mass x,y,z position
    figure;
    subplot(3,2,1);
    plot(t,xr,'r', t,x(:,1)+x_camError,'g', t,x(:,1),'k');
    title('Mass position')
    ylabel('X [m]')
    grid on;
    subplot(3,2,3);
    plot(t,yr,'r', t,x(:,2)+y_camError,'g', t,x(:,2),'k');
    ylabel('Y [m]')
    grid on;
    subplot(3,2,5);
    plot(t,zr,'r', t,x(:,3)+z_camError,'g', t,x(:,3),'k');
    ylabel('Z [m]')
    xlabel('Time [s]')
    legend('Reference', 'Measured by camera', 'True', 'location', 'northwest')
    grid on;
    % Plot x, y, z mass position error
    subplot(3,2,2);
    plot(t,xr-x(:,1)-x_camError,'g', t,xr-x(:,1),'k');
    title('Mass position error')
    ylabel('X error [m]')
    grid on;
    subplot(3,2,4);
    plot(t,yr-x(:,2)-y_camError,'g', t,yr-x(:,2),'k');
    ylabel('Y error [m]')
    grid on;
    subplot(3,2,6);
    plot(t,zr-x(:,3)-z_camError,'g', t,zr-x(:,3),'k');
    grid on;
    ylabel('Z error [m]')
    xlabel('Time [s]')
    legend('Measured by camera', 'True', 'location', 'northwest')
    set(gcf,'position',[200,200,1000,800]);

    % Plot cable lengths
    figure;
    subplot(4,2,1);
    plot(t,la_r,'r', t,x(:,7),'g', t,x(:,7)+la_error,'k');
    title('Cable lengths')
    ylabel('La [m]')
    grid on;
    subplot(4,2,3);
    plot(t,lb_r,'r', t,x(:,8),'g', t,x(:,8)+lb_error,'k');
    ylabel('Lb [m]')
    grid on;
    subplot(4,2,5);
    plot(t,lc_r,'r', t,x(:,9),'g', t,x(:,9)+lc_error,'k');
    ylabel('Lc [m]')
    grid on;
    subplot(4,2,7);
    plot(t,ld_r,'r', t,x(:,10),'g', t,x(:,10)+ld_error,'k');
    ylabel('Ld [m]')
    xlabel('Time [s]')
    legend('Reference', 'Measured by encoder', 'True', 'location', 'northwest')
    grid on;
    % Plot cable length errors
    subplot(4,2,2);
    title('Cable length error')
    plot(t,la_r-x(:,7),'g', t,la_r-x(:,7)-la_error,'k');
    ylabel('La_e [m]')
    grid on;
    subplot(4,2,4);
    plot(t,lb_r-x(:,8),'g', t,lb_r-x(:,8)-lb_error,'k');
    ylabel('Lb_e [m]')
    grid on;
    subplot(4,2,6);
    plot(t,lc_r-x(:,9),'g', t,lc_r-x(:,9)-lc_error,'k');
    ylabel('Lc_e [m]')
    grid on;
    subplot(4,2,8);
    plot(t,ld_r-x(:,10),'g', t,ld_r-x(:,10)-ld_error,'k');
    ylabel('Ld_e [m]')
    xlabel('Time [s]')
    grid on;
    set(gcf,'position',[300,100,1000,800]);
    legend('Measured by encoder', 'True', 'location', 'northwest')
end

function [] = plotCableTensions(time,states)
    global B H anchorPoints
    global da db dc dd
    global la_error lb_error lc_error ld_error

    nTimes = length(time);
    forces = NaN(nTimes,3);
    tensions = NaN(nTimes,4);
    dWork = NaN(nTimes,4);

    dWork(1,:)= [0,0,0,0];
    Work = 0;
    for i=1:length(time)
        t = time(i);
        x = states(i,:);

        % True unstretched cable lengths
        la = x(7) + la_error;
        lb = x(8) + lb_error;
        lc = x(9) + lc_error;
        ld = x(10) + ld_error;

        % Cable lengths calculated from true mass position
        trueAnchorPoints = anchorPoints+[da; db; dc; dd];
        cableLengths = positionToCableLengths([x(1),x(2),x(3)], trueAnchorPoints);
        la_m = cableLengths(1);
        lb_m = cableLengths(2);
        lc_m = cableLengths(3);
        ld_m = cableLengths(4);

        % Calculate the cable tensions based on cable stretch amount
        tensionFunctionSelect = 3;
        [Ta,~] = tensionFunction(tensionFunctionSelect, la_m, la, []);
        [Tb,~] = tensionFunction(tensionFunctionSelect, lb_m, lb, []);
        [Tc,~] = tensionFunction(tensionFunctionSelect, lc_m, lc, []);
        [Td,~] = tensionFunction(tensionFunctionSelect, ld_m, ld, []);

        % Resolve cable tensions into Cartesian components
        TaC = Ta*[-x(1), -x(2), H-x(3)]/la_m;
        TbC = Tb*[B-x(1), -x(2), H-x(3)]/lb_m;
        TcC = Tc*[B-x(1), B-x(2), H-x(3)]/lc_m;
        TdC = Td*[-x(1), B-x(2), H-x(3)]/ld_m;
        Tx = TaC(1) + TbC(1) + TcC(1) + TdC(1);
        Ty = TaC(2) + TbC(2) + TcC(2) + TdC(2);
        Tz = TaC(3) + TbC(3) + TcC(3) + TdC(3);

        tensions(i,:) = [Ta Tb Tc Td];
        forces(i,:) = [Tx Ty Tz];

        % Calculate motor energy required
        if i>1
            dT = t-time(i-1);
            dWork(i,:) = tensions(i-1,:).^2*dT;
            Work = Work + sum(dWork(i,:));
        end
    end

    disp('Total work [J]:')
    format long g
    disp(Work)

    figure()
    subplot(1,3,1)
    plot(time,forces(:,1),'.k')
    title('X')
    xlabel('Time [s]')
    ylabel('Net force [N]')
    grid on
    subplot(1,3,2)
    plot(time,forces(:,2),'.k')
    title('Y')
    xlabel('Time [s]')
    grid on
    subplot(1,3,3)
    plot(time,forces(:,3),'.k')
    title('Z')
    xlabel('Time [s]')
    grid on

    figure()
    subplot(1,4,1)
    plot(time,tensions(:,1),'.k')
    title('Cable a')
    xlabel('Time [s]')
    ylabel('Tension [N]')
    grid on
    subplot(1,4,2)
    plot(time,tensions(:,2),'.k')
    title('Cable b')
    xlabel('Time [s]')
    grid on
    subplot(1,4,3)
    plot(time,tensions(:,3),'.k')
    title('Cable c')
    xlabel('Time [s]')
    grid on
    subplot(1,4,4)
    plot(time,tensions(:,4),'.k')
    title('Cable d')
    xlabel('Time [s]')
    grid on
end
```

Table A17. *cableRobotSim_disturbancePrediction.m* is a script that runs simulations of a four-cable CSR to demonstrate the anchor-point prediction algorithm in Table A2.

```
% Disturbance prediction simulation
clear all;

% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Square shape with side lengths B)
% ====================

%         Z
%         |
%         |
%         |  d_________________c
%         | /|                /|
%         a/_|_______________b/ |
%         |  |              |  |
%         |  |  Y           |  |  H
%         |  | /            |  |
%         |  |/             |  |
%         |  |______________|__|
%         | /               | /  B
%         |/________________|/___________ X
%                 B


% ====================
% Initialize variables
% ====================

% Anchor point dimensions (see physical configuration above)
global B H anchorPoints
B = 0.75;               % length of each side [m]
H = 0.795;              % height of support [m]
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];    % anchor points for 4 cables

% Mass dynamics
global m c g
m = 1;                  % mass [kg]
c = 2;                  % viscous damping [Ns/m]
g = 9.81;               % gravity  [m/s^2]

% Cable length servo dynamics
global wn zeta
wn = 50;                % natural frequency [rad/s] of second-order length response
zeta = 0.9;             % damping factor of second-order length response


% ====================
% Disturbances
% ====================

% Anchor point position disturbances
% distubance defined as actual minus ideal
% Format: [x,y,z] position error from ideal anchor point
global da db dc dd;                     % actual disturbances [m]
global da_est db_est dc_est dd_est;     % estimated disturbances [m]
% Create random [x, y, z] disturbance for each cable anchor point
maxDist = 0.05;         % maximum disturbance amount [m]
da = round(2*(rand(1,3)-0.5)*maxDist,4);
db = round(2*(rand(1,3)-0.5)*maxDist,4);
dc = round(2*(rand(1,3)-0.5)*maxDist,4);
dd = round(2*(rand(1,3)-0.5)*maxDist,4);
% Initial estimate of disturbances is zero
```

```
da_est = [0,0,0];
db_est = [0,0,0];
dc_est = [0,0,0];
dd_est = [0,0,0];


% ====================
% Simulate and plot
% ====================

% States
% x = [x,y,z, vx,vy,vz, la,lb,lc,ld, la_dot,lb_dot,lc_dot,ld_dot];
%   = [position, velocity, cable lengths, cable length time derivatives]
x0 = 7*B/8;
y0 = B/2;
z0 = 0.1;
cableLengths = positionToCableLengths([x0,y0,z0], anchorPoints);
la0 = cableLengths(1);
lb0 = cableLengths(2);
lc0 = cableLengths(3);
ld0 = cableLengths(4);
X0 = [x0,y0,z0, 0,0,0, la0,lb0,lc0,ld0, 0,0,0,0];

% Run first simulation (assume zero anchor point disturbances)
t_f = 120;
t = 0:0.01:t_f;
[t,x] = ode45(@dynamics, t, X0);

%% Plot first simulation

% Clip simulation
tstart = 3;
ind = find(t>=tstart,1,'first');
t = t(ind:end);
x = x(ind:end, :);

basicPlot(t,x);
positions = x(:,1:3);
cableLengths = x(:,7:10);
trueAnchorPoints = anchorPoints+[da; db; dc; dd];
plotCables = false;
animatedPlot(t, positions, cableLengths, trueAnchorPoints, plotCables);
% Add reference curves to the animated plot
xPoints = [7*B/8 7*B/8 7*B/8 7*B/8 6*B/8 6*B/8 B/2 B/2 2*B/8 2*B/8 B/8 B/8 B/8 B/8 B/8
B/8 2*B/8 2*B/8 B/2 B/2 6*B/8 6*B/8 7*B/8 7*B/8 7*B/8];
yPoints = [B/2 B/2 6*B/8 6*B/8 7*B/8 7*B/8 7*B/8 7*B/8 7*B/8 7*B/8 6*B/8 6*B/8 B/2 B/2
2*B/8 2*B/8 B/8 B/8 B/8 B/8 B/8 B/8 2*B/8 2*B/8 B/2];
zPoints = 0.1*ones(1,25);
timeStamps = 0:5:120;
[xr,yr,zr] = pointsPath(xPoints, yPoints, zPoints, timeStamps, t);
stem3(anchorPoints(:,1),anchorPoints(:,2),anchorPoints(:,3),'r')
plot3(xr,yr,zr,'g');
% Add some fake points for custom legend
h = zeros(3, 1);
h(1) = plot(NaN,NaN,'or');
h(2) = plot(NaN,NaN,'.r');
h(3) = plot(NaN,NaN,'-g');
% legend(h,'Ideal anchor points','Actual anchor points','Reference
path','Location','southoutside');

%% Inject random noise to state measurements

% Maximum noise amounts [m]
maxLengthNoise = 0.002;
maxPositionNoise = 0.002;

numPoints = length(t);
```

```
z = zeros(numPoints,1);
% Generate random noise vectors for each cable length
laNoise = 2*(rand(numPoints,1)-0.5)*maxLengthNoise;
lbNoise = 2*(rand(numPoints,1)-0.5)*maxLengthNoise;
lcNoise = 2*(rand(numPoints,1)-0.5)*maxLengthNoise;
ldNoise = 2*(rand(numPoints,1)-0.5)*maxLengthNoise;
% Generate random noise vectors for each cartesian position
xNoise = 2*(rand(numPoints,1)-0.5)*maxPositionNoise;
yNoise = 2*(rand(numPoints,1)-0.5)*maxPositionNoise;
zNoise = 2*(rand(numPoints,1)-0.5)*maxPositionNoise;

% Create new state vector adding the random noise
x_noise = x + [xNoise yNoise zNoise z z z laNoise lbNoise lcNoise ldNoise z z z z];

% Moving average filter of the noisy state vector
x_noise_filtered = NaN(numPoints,14);
avgSamples = 100;
for i=avgSamples:numPoints
    avgSum = zeros(1,14);
    for j=1:avgSamples
        avgSum = avgSum + x_noise(i-j+1,:);
    end
    x_noise_filtered(i,:) = avgSum/avgSamples;
end


%% ====================
% Anchor-point disturbance prediction algorithms
% ====================

%1,4,5
method = 4;
% Uses data from the first simulation
positions = x(:,1:3);
cableLengths = x(:,7:10);
aEstimate = estimateDisturbances(t, positions, cableLengths, anchorPoints,
[64,74,84,94], method);
bEstimate = estimateDisturbances(t, positions, cableLengths, anchorPoints,
[94,104,114,4], method);
cEstimate = estimateDisturbances(t, positions, cableLengths, anchorPoints,
[4,14,24,34], method);
dEstimate = estimateDisturbances(t, positions, cableLengths, anchorPoints,
[34,44,54,64], method);
% Uses data from the first simulation, plus measurement noise
positions = x_noise(:,1:3);
cableLengths = x_noise(:,7:10);
aEstimateNoise = estimateDisturbances(t, positions, cableLengths, anchorPoints,
[64,74,84,94], method);
bEstimateNoise = estimateDisturbances(t, positions, cableLengths, anchorPoints,
[94,104,114,4], method);
cEstimateNoise = estimateDisturbances(t, positions, cableLengths, anchorPoints,
[4,14,24,34], method);
dEstimateNoise = estimateDisturbances(t, positions, cableLengths, anchorPoints,
[34,44,54,64], method);
% Plot estimated and actual anchor point positions
figure()
subplot(1,4,1)
plot(1:3,da,'or',1:3,aEstimate(1,:),'.k',1:3,aEstimateNoise(1,:),'xb')
xlim([0 4]);ylim([-1.5*maxDist 1.5*maxDist]);
ylabel('Disturbance amount [m]')
xticks([1 2 3]);xticklabels({'X','Y','Z'});grid on;
title('a')
subplot(1,4,2)
plot(1:3,db,'or',1:3,bEstimate(2,:),'.k',1:3,bEstimateNoise(2,:),'xb')
xlim([0 4]);ylim([-1.5*maxDist 1.5*maxDist]);
xticks([1 2 3]);xticklabels({'X','Y','Z'});grid on;
title('b')
```

```
subplot(1,4,3)
plot(1:3,dc,'or',1:3,cEstimate(3,:),'.k',1:3,cEstimateNoise(3,:),'xb')
xlim([0 4]);ylim([-1.5*maxDist 1.5*maxDist]);
xticks([1 2 3]);xticklabels({'X','Y','Z'});grid on;
title('c')
subplot(1,4,4)
plot(1:3,dd,'or',1:3,dEstimate(4,:),'.k',1:3,dEstimateNoise(4,:),'xb')
xlim([0 4]);ylim([-1.5*maxDist 1.5*maxDist]);
xticks([1 2 3]);xticklabels({'X','Y','Z'});grid on;
title('d')
legend({'Actual','Estimate','Estimate with noise'},'Location','South')

% Save result
da_est = aEstimateNoise(1,:);
db_est = bEstimateNoise(2,:);
dc_est = cEstimateNoise(3,:);
dd_est = dEstimateNoise(4,:);


%% ====================
% Repeat simulation using estimated disturbances
% ====================

X02 = X0;
t2 = 0:0.01:t_f;
[t2,x2] = ode45(@dynamics, t2, X02);


%% ====================
% Plot results (first and second simulation, together)
% ====================

% Clip simulation
tstart = 3;
ind = find(t2>=tstart,1,'first');
t2 = t2(ind:end);
x2 = x2(ind:end, :);

basicPlot(t2,x2);
basicPlot(t2,x2);

trueAnchorPoints = anchorPoints+[da; db; dc; dd];
estimatedAnchorPoints = anchorPoints+[da_est; db_est; dc_est; dd_est];
plotCables = false;

disp("3D plot 1: intial simulation, with unknown disturbances")
positions = x(:,1:3);
cableLengths = x(:,7:10);
animatedPlot(t, positions, cableLengths, trueAnchorPoints, plotCables);
% Add reference curves to the animated plot
xPoints = [7*B/8 7*B/8 7*B/8 7*B/8 6*B/8 6*B/8 B/2 B/2 2*B/8 2*B/8 B/8 B/8 B/8 B/8 B/8
B/8 2*B/8 2*B/8 B/2 B/2 6*B/8 6*B/8 7*B/8 7*B/8 7*B/8];
yPoints = [B/2 B/2 6*B/8 6*B/8 7*B/8 7*B/8 7*B/8 7*B/8 7*B/8 7*B/8 6*B/8 6*B/8 B/2 B/2
2*B/8 2*B/8 B/8 B/8 B/8 B/8 B/8 B/8 2*B/8 2*B/8 B/2];
zPoints = 0.1*ones(1,25);
timeStamps = 0:5:120;
[xr,yr,zr] = pointsPath(xPoints, yPoints, zPoints, timeStamps, t);
stem3(anchorPoints(:,1),anchorPoints(:,2),anchorPoints(:,3),'r')
plot3(xr,yr,zr,'g');
% Add some fake points for custom legend
h = zeros(3, 1);
h(1) = plot(NaN,NaN,'or');
h(2) = plot(NaN,NaN,'.r');
h(3) = plot(NaN,NaN,'-g');
legend(h,'Ideal anchor points','Actual anchor points','Reference
path','Location','southoutside');
```

```
disp("3D plot 2: final simulation, with estimated disturbances")
positions2 = x2(:,1:3);
cableLengths2 = x2(:,7:10);
animatedPlot(t2, positions2, cableLengths2, trueAnchorPoints, plotCables);
% Add reference curves to the animated plot
[xr,yr,zr] = pointsPath(xPoints, yPoints, zPoints, timeStamps, t);
stem3(estimatedAnchorPoints(:,1),estimatedAnchorPoints(:,2),estimatedAnchorPoints(:,3)
,'r')
plot3(xr,yr,zr,'g');
% Add some fake points for custom legend
h = zeros(3, 1);
h(1) = plot(NaN,NaN,'or');
h(2) = plot(NaN,NaN,'.r');
h(3) = plot(NaN,NaN,'-g');
legend(h,'Estimated anchor points','Actual anchor points','Reference
path','Location','southoutside');


% ====================
% Robot model
% ====================

function dx = dynamics(t,x)
    % Access global variables
    global B H anchorPoints
    global m c g wn zeta
    global da db dc dd;
    global da_est db_est dc_est dd_est;

    % True and estimated anchor point locations
    trueAnchorPoints = anchorPoints + [da; db; dc; dd];
    estimatedAnchorPoints = anchorPoints + [da_est; db_est; dc_est; dd_est];

    % True unstretched cable lengths
    la = x(7);
    lb = x(8);
    lc = x(9);
    ld = x(10);

    % Cable lengths calculated from true mass position
    cableLengths = positionToCableLengths([x(1),x(2),x(3)], trueAnchorPoints);
    la_m = cableLengths(1);
    lb_m = cableLengths(2);
    lc_m = cableLengths(3);
    ld_m = cableLengths(4);

    % Calculate the cable tensions based on cable stretch amount
    tensionFunctionSelect = 3;
    [Ta,~] = tensionFunction(tensionFunctionSelect, la_m, la, []);
    [Tb,~] = tensionFunction(tensionFunctionSelect, lb_m, lb, []);
    [Tc,~] = tensionFunction(tensionFunctionSelect, lc_m, lc, []);
    [Td,~] = tensionFunction(tensionFunctionSelect, ld_m, ld, []);

    % Resolve cable tensions into Cartesian components
    TaC = Ta*[-x(1), -x(2), H-x(3)]/la_m;
    TbC = Tb*[B-x(1), -x(2), H-x(3)]/lb_m;
    TcC = Tc*[B-x(1), B-x(2), H-x(3)]/lc_m;
    TdC = Td*[-x(1), B-x(2), H-x(3)]/ld_m;
    Tx = TaC(1) + TbC(1) + TcC(1) + TdC(1);
    Ty = TaC(2) + TbC(2) + TcC(2) + TdC(2);
    Tz = TaC(3) + TbC(3) + TcC(3) + TdC(3);

    % Get reference cartesian position
%      xPoints =    [7*B/8  7*B/8    B/2    B/8 B/8 B/8 B/2 7*B/8 7*B/8];
%      yPoints =    [  B/2  7*B/8  7*B/8 7*B/8 B/2 B/8 B/8    B/8    B/2];
%      zPoints =    [  0.1    0.2    0.1    0.2 0.1 0.2 0.1    0.2    0.1];
%      timeStamps = [    0     10     20     30  40  50  60     70     80];
%     xPoints =    [ 7*B/8 7*B/8 B/8   B/8 7*B/8 B/8 7*B/8 7*B/8 B/8   B/8];
%     yPoints =    [ B/8   7*B/8 7*B/8 B/8 B/8   B/8 B/8   7*B/8 7*B/8 B/8];
%     zPoints =    [ 0.1   0.1   0.1   0.1 0.1   0.1 0.1   0.1   0.1   0.1];
%     timeStamps = [ 0     10    20    30  40    50  60    70    80    90];
    xPoints = [7*B/8 7*B/8 7*B/8 7*B/8 6*B/8 6*B/8 B/2 B/2 2*B/8 2*B/8 B/8 B/8 B/8 B/8
B/8 B/8 2*B/8 2*B/8 B/2 B/2 6*B/8 6*B/8 7*B/8 7*B/8 7*B/8];
    yPoints = [B/2 B/2 6*B/8 6*B/8 7*B/8 7*B/8 7*B/8 7*B/8 7*B/8 7*B/8 6*B/8 6*B/8 B/2
B/2 2*B/8 2*B/8 B/8 B/8 B/8 B/8 B/8 B/8 2*B/8 2*B/8 B/2];
    zPoints = 0.1*ones(1,25);
    timeStamps = 0:5:120;
    [xr,yr,zr] = pointsPath(xPoints, yPoints, zPoints, timeStamps, t);

    % Convert reference position to reference cable lengths
    cableLengths = positionToCableLengths([xr,yr,zr], estimatedAnchorPoints);
    la_r = cableLengths(1);
    lb_r = cableLengths(2);
    lc_r = cableLengths(3);
    ld_r = cableLengths(4);

    % ------------
    % Control laws
    % ------------

    u = [la_r lb_r lc_r ld_r];

    % ------------
    % Calculate state derrivatives
    % ------------

    dx = zeros(14,1);

    % Mass dynamics
    dx(1)  = x(4);
    dx(2)  = x(5);
    dx(3)  = x(6);
    dx(4)  = (Tx - c*x(4))/m;
    dx(5)  = (Ty - c*x(5))/m;
    dx(6)  = (Tz - c*x(6))/m - g;

    % Cable length dynamics (second order response)
    dx(7)  = x(11);
    dx(8)  = x(12);
    dx(9)  = x(13);
    dx(10)  = x(14);
    dx(11)  = u(1)*wn^2 - 2*wn*zeta*x(11) - wn^2*x(7);
    dx(12)  = u(2)*wn^2 - 2*wn*zeta*x(12) - wn^2*x(8);
    dx(13)  = u(3)*wn^2 - 2*wn*zeta*x(13) - wn^2*x(9);
    dx(14)  = u(4)*wn^2 - 2*wn*zeta*x(14) - wn^2*x(10);
end


% ====================
% Plotting functions
% ====================

function basicPlot(t,x)
    global B H anchorPoints
    global da_est db_est dc_est dd_est;

    % Anchorpoint estimates
    estimatedAnchorPoints = anchorPoints + [da_est; db_est; dc_est; dd_est];

    % Get reference cartesian position
%     xPoints =    [7*B/8  7*B/8    B/2   B/8 B/8 B/8 B/2 7*B/8 7*B/8];
%     yPoints =    [  B/2  7*B/8  7*B/8 7*B/8 B/2 B/8 B/8   B/8   B/2];
%     zPoints =    [  0.1    0.2    0.1   0.2 0.1 0.2 0.1   0.2   0.1];
%     timeStamps = [    0     10     20    30  40  50  60    70    80];
%     xPoints =    [ 7*B/8 7*B/8 B/8   B/8 7*B/8 B/8 7*B/8 7*B/8 B/8   B/8];
%     yPoints =    [ B/8   7*B/8 7*B/8 B/8 B/8   B/8 B/8   7*B/8 7*B/8 B/8];
%     zPoints =    [ 0.1   0.1   0.1   0.1 0.1   0.1 0.1   0.1   0.1   0.1];
%     timeStamps = [ 0     10    20    30  40    50  60    70    80    90];
    xPoints = [7*B/8 7*B/8 7*B/8 7*B/8 6*B/8 6*B/8 B/2 B/2 2*B/8 2*B/8 B/8 B/8 B/8 B/8
B/8 B/8 2*B/8 2*B/8 B/2 B/2 6*B/8 6*B/8 7*B/8 7*B/8 7*B/8];
    yPoints = [B/2 B/2 6*B/8 6*B/8 7*B/8 7*B/8 7*B/8 7*B/8 7*B/8 7*B/8 6*B/8 6*B/8 B/2
B/2 2*B/8 2*B/8 B/8 B/8 B/8 B/8 B/8 B/8 2*B/8 2*B/8 B/2];
    zPoints = 0.1*ones(1,25);
    timeStamps = 0:5:120;
    [xr,yr,zr] = pointsPath(xPoints, yPoints, zPoints, timeStamps, t);
    % Convert to reference cable lengths
    cableLengths = positionToCableLengths([xr, yr, zr], estimatedAnchorPoints);
    la_r = cableLengths(:,1);
    lb_r = cableLengths(:,2);
    lc_r = cableLengths(:,3);
    ld_r = cableLengths(:,4);

    % Plot mass x,y,z position
    figure;
    subplot(3,2,1);
    plot(t,xr,'r', t,x(:,1),'k');
    title('Mass position')
    ylabel('X [m]')
    grid on;
    subplot(3,2,3);
    plot(t,yr,'r', t,x(:,2),'k');
    ylabel('Y [m]')
    grid on;
    subplot(3,2,5);
    plot(t,zr,'r', t,x(:,3),'k');
    ylabel('Z [m]')
    xlabel('Time [s]')
    legend('Reference', 'True', 'location', 'northwest')
    grid on;
    % Plot x, y, z mass position error
    subplot(3,2,2);
    plot(t,xr-x(:,1),'k');
    title('Mass position error')
    ylabel('X error [m]')
    grid on;
    subplot(3,2,4);
    plot(t,yr-x(:,2),'k');
    ylabel('Y error [m]')
    grid on;
    subplot(3,2,6);
    plot(t,zr-x(:,3),'k');
    grid on;
    ylabel('Z error [m]')
    xlabel('Time [s]')
    set(gcf,'position',[200,200,1000,800]);

    % Plot cable lengths
    figure;
    subplot(4,2,1);
    plot(t,la_r,'r', t,x(:,7),'k');
    title('Cable lengths')
    ylabel('La [m]')
    grid on;
    subplot(4,2,3);
    plot(t,lb_r,'r', t,x(:,8),'k');
    ylabel('Lb [m]')
    grid on;
    subplot(4,2,5);
    plot(t,lc_r,'r', t,x(:,9),'k');
    ylabel('Lc [m]')
    xlabel('Time [s]')
    grid on;
    subplot(4,2,7);
    plot(t,ld_r,'r', t,x(:,10),'k');
    ylabel('Ld [m]')
    xlabel('Time [s]')
    legend('Reference', 'True', 'location', 'northwest')
    grid on;
    % Plot cable length errors
    subplot(4,2,2);
    title('Cable length error')
    plot(t,la_r-x(:,7),'k');
    ylabel('La_e [m]')
    grid on;
    subplot(4,2,4);
    plot(t,lb_r-x(:,8),'k');
    ylabel('Lb_e [m]')
    grid on;
    subplot(4,2,6);
    plot(t,lc_r-x(:,9),'k');
    ylabel('Lc_e [m]')
    xlabel('Time [s]')
    grid on;
    subplot(4,2,8);
    plot(t,ld_r-x(:,10),'k');
    ylabel('Ld_e [m]')
    xlabel('Time [s]')
    grid on;
    set(gcf,'position',[300,100,1000,800]);
end
```

Table A18. *cableRobotSim_tension.m* is a script that runs simulations of a four-cable CSR using cable-tension-based control.

```
% Cable Robot Simulation, tension control
clear all;

% ====================
% Physical configuration of the cable-suspended robot (cables not shown)
%   (Square shape with side lengths B)
% ====================

%            d_________________c
%           /|                /|
%         a/_|______________b/ |
%         |  |              |  |
%         |  |              |  |  H
%         |  |              |  |
%         |  |              |  |
%         |  |______________|__|
%         | /               | /  B
%         |/________________|/
%                 B


% ====================
% Define some variables
% ====================
global anchorPoints

% Anchor point dimensions (see physical configuration above)
B = 10.0;       % length of each side [m]
H = 10.0;       % height of cable anchor points [m]
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];    % anchor points for 4 cables
```

```
% End effector dynamics
global m g c;
m = 10;         % mass [kg]
g = 9.81;       % gravity acceleration [m/s^2]
c = 10;         % coulomb damping for mass movement [Ns/m]


% ====================
% Simulate and plot
% ====================

% States
% x = [x,y,z, vx,vy,vz, x_e,y_e,z_e]
%   = [position, velocity, position error]
x0 = 8;
y0 = 5;
z0 = 2;
X0 = [x0,y0,z0,0,0,0,0,0,0];

% Run simulation
tf = 60;            % simulation end time [s]
t = 0:0.01:tf;
[t,x] = ode45(@dynamics,t,X0);

%% Plot results
basicPlot(t, x);
positions = x(:,1:3);
plotCables = false;
animatedPlot(t, positions, [], anchorPoints, plotCables)
% Add reference curves to the animated plot
f = 1/30;
xr = 5 + 3*cos(2*pi*f*t);
yr = 5 + 3*sin(2*pi*f*t);
zr = 2 + (1/15)*t;
plot3(xr,yr,zr,'--r');
% Add some fake points for custom legend
h = plot(NaN,NaN,'--r');
legend(h,'Reference path','Location','southoutside');

%% Plot cable tensions
tic
plotCableTensions(t, x);
toc


%% ====================
% Model
% ====================
function [dx, cartesianForces] = dynamics(t,x)
    % Select controller
    controller = 1;

    % Retrieve global variables defined above
    global m g c
    MM = m*eye(3);      % mass matrix [kg]
    CC = c*eye(3);      % damping matrix [N*s/m]
    gg = [0;0;m*g];     % gravity matrix [N]

    % Disturbances
    F_dist = 0*sin(t*2*pi*0.25);        % Force disturbance [N]
    m_dist = 0;                         % mass disturbance [kg]
    c_dist = 0;                     % damping disturbance [N*s/m]
    % Maximum allowable disturbances
    mbar = 5;           % [kg]
    cbar = 10;          % [N*s/m]
    gbar = mbar*g;      % [N]
    % Set reference trajectory
    f = 1/30;
    xr = 5 + 3*cos(2*pi*f*t);
    yr = 5 + 3*sin(2*pi*f*t);
    zr = 2 + (1/15)*t;
    % First time derivative of reference trajectory
    xr_d = -3*2*pi*f*sin(2*pi*f*t);
    yr_d = 3*2*pi*f*cos(2*pi*f*t);
    zr_d = 1/15;
    % Second time derivative of reference trajectory
    xr_dd = -3*(2*pi*f)^2*cos(2*pi*f*t);
    yr_dd = -3*(2*pi*f)^2*sin(2*pi*f*t);
    zr_dd = 0;

    % Calculate state errors
    xe = xr - x(1);
    ye = yr - x(2);
    ze = zr - x(3);

    % Calculate state error time derivatives
    xe_d = xr_d - x(4);
    ye_d = yr_d - x(5);
    ze_d = zr_d - x(6);


    % ------------
    % Control laws
    % ------------

    % Default is no control
    cartesianForces = zeros(1,3);
    switch controller
        case 1      % Feedback linearization (computed torque)
            Kp = 15;
            Kd = 2;
            q_d = [x(4);x(5);x(6)];
            qr_dd = [xr_dd;yr_dd;zr_dd];
            e = [xe;ye;ze];
            e_d = [xe_d;ye_d;ze_d];
            cartesianForces = MM*(qr_dd + Kd*e_d + Kp*e) + CC*q_d + gg;

        case 2      % Sliding-mode control
            % Parameters
            Sp = 100;
            Sd = 20;
            k0 = 5;    % any positive constant
            alpha = 0.5;
            sat = @(x) min(1, max(-1,x/alpha));

            % Define the sliding surface
            e = [xe;ye;ze];
            e_d = [xe_d;ye_d;ze_d];
            e_int = [x(7);x(8);x(9)];
            s = e_d + Sd*e + Sp*e_int;

            % Calculate switching constant
            qr_dd = [xr_dd;yr_dd;zr_dd];
            q_d = [x(4);x(5);x(6)];
            K = mbar*abs(qr_dd+Sd*e_d+Sp*e) + cbar*abs(q_d) + [0;0;gbar] + k0;

            % Control law
            cartesianForces = MM*qr_dd + CC*q_d + gg + MM*Sd*e_d + MM*Sp*e +
K.*sign(s);
    end
    % ------------
    % Caclulate state derrivatives
    % ------------

    % Initialize time derivative vector
    dx = zeros(9,1);
    Fx = cartesianForces(1);
    Fy = cartesianForces(2);
    Fz = cartesianForces(3);

    % Mass dynamics
    dx(1) = x(4);
    dx(2) = x(5);
    dx(3) = x(6);
    % From Newton's law: mass*accel = T - c*vel - mass*g;
    dx(4) = ((Fx+F_dist) - (c+c_dist)*x(4))/(m+m_dist);
    dx(5) = ((Fy+F_dist) - (c+c_dist)*x(5))/(m+m_dist);
    dx(6) = ((Fz+F_dist) - (c+c_dist)*x(6))/(m+m_dist) - g;

    % Error integral states
    dx(7) = xe;
    dx(8) = ye;
    dx(9) = ze;
end


% ====================
% Plotting functions
% ====================

function basicPlot(t, x)

    % Generate reference trajectory
    f = 1/30;
    xr = 5 + 3*cos(2*pi*f*t);
    yr = 5 + 3*sin(2*pi*f*t);
    zr = 2 + (1/15)*t;

    % Position
    figure;
    subplot(3,2,1);
    plot(t,xr,'--r', t,x(:,1),'k');
    ylabel('X [m]')
    grid on;
    subplot(3,2,3);
    plot(t,yr,'--r', t,x(:,2),'k');
    ylabel('Y [m]')
    grid on;
    subplot(3,2,5);
    plot(t,zr,'--r', t,x(:,3),'k');
    ylabel('Z [m]')
    xlabel('Time [s]')
    legend('Reference', 'Simulated', 'location', 'northwest')
    grid on;
    % Position error
    maxErrorLine = 0.01;
    subplot(3,2,2);
    plot(t,xr-x(:,1),'k', t,maxErrorLine*ones(length(t),1),'r', t,-
maxErrorLine*ones(length(t),1),'r');
    ylabel('X error [m]')
    grid on;
    subplot(3,2,4);
    plot(t,yr-x(:,2),'k', t,maxErrorLine*ones(length(t),1),'r', t,-
maxErrorLine*ones(length(t),1),'r');
    ylabel('Y error [m]')
    grid on;
    subplot(3,2,6);
    plot(t,zr-x(:,3),'k', t,maxErrorLine*ones(length(t),1),'r', t,-
maxErrorLine*ones(length(t),1),'r');
    grid on;
    ylabel('Z error [m]')
    xlabel('Time [s]')

end

function [] = plotCableTensions(time,states)
    global anchorPoints

    nTimes = length(time);
    forces = NaN(nTimes,3);
    tensions = NaN(nTimes,4);

    sum1 = 0;
    sum2 = 0;
    for i=1:length(time)
        t = time(i);
        x = states(i,:);
        [~, cartesianForces] = dynamics(t,x);
        forces(i,:) = cartesianForces;
        % Convert cartesian forces to cable tensions
        method = 4;
        position = [x(1) x(2) x(3)];
        [cableTensions, ~] =
positionToTension(position,anchorPoints,cartesianForces,method);
        tensions(i,:) = cableTensions;

        if i>1
            dT = t-time(i-1);
            sum1 = sum1 + sum(tensions(i-1,:))*dT;
            sum2 = sum2 + sum(tensions(i-1,:).^2)*dT;
        end
    end

    disp('Tension integral:')
    format long g
    disp(sum1)
    disp('Tension squared integral:')
    disp(sum2)

    figure()
    subplot(1,3,1)
    plot(time,forces(:,1),'.k')
    title('X')
    xlabel('Time [s]')
    ylabel('Net force [N]')
    grid on
    subplot(1,3,2)
    plot(time,forces(:,2),'.k')
    title('Y')
    xlabel('Time [s]')
    grid on
    subplot(1,3,3)
    plot(time,forces(:,3),'.k')
    title('Z')
    xlabel('Time [s]')
    grid on

    figure()
    subplot(1,4,1)
    plot(time,tensions(:,1),'.k')
    title('Cable a')
    xlabel('Time [s]')
    ylabel('Tension [N]')
    grid on
    subplot(1,4,2)
    plot(time,tensions(:,2),'.k')
    title('Cable b')
    xlabel('Time [s]')
    grid on
    subplot(1,4,3)
    plot(time,tensions(:,3),'.k')
    title('Cable c')
    xlabel('Time [s]')
    grid on
    subplot(1,4,4)
    plot(time,tensions(:,4),'.k')
    title('Cable d')
    xlabel('Time [s]')
    grid on
end
```

Table A19. *dcMotorServoSim.m* is a script that runs simulations for a closed-loop DC servo motor. This script also plots some controller-design analysis plots.

```
% DC motor servo control simulation
clear all;

% ====================
% Physical configuration:
%   A voltage-controlledDC motor with a point mass directly attached using a string
% ====================

%
%       voltageSource ----> DC motor -----> mass vertical position
%                           ^                        |
%                           |                        |
%                           ------ load torque -----
%

% ====================
% Define some variables
% ====================

m = 0.5;    % weight mass [kg]
g = 9.81;   % gravity acceleration [m/s^2]

% DC motor characteristics (see Maxon Amax26 110950 datasheet)
Kt = 0.0202;            % torque constant [Nm/A]
Ke = 60/(2*pi*472);     % back emf constant [Vs/rad]
B = 0.000001;           % linear damping [Ns/rad]
R = 8;                  % resistance [ohm]
J = 12.4/(1000*100*100);% rotor inertia [kg*m^2]
L = 0.00086;            % inductance [H]
r = 0.01;               % motor wheel radius [m]

 % ====================
% Transfer functions
% ====================

% Laplace variable
s = tf([1 0],1);

% Open-loop
%   Motor voltage [V] to motor speed [rad/s]
omega_V = Kt/(L*J*s^2 + (J*R+B*L)*s + B*R+Kt*Ke);
%   Load tension to [V] to motor speed [rad/s]
omega_T = -r*(L*s+R)/(L*J*s^2 + (J*R+B*L)*s + B*R+Kt*Ke);
```

```
%   Motor voltage [V] to mass position [m]
Y_V = Kt*r/((L*J+L*m*r^2)*s^3 + (J*R+B*L+m*r^2*R)*s^2 + (B*R+Kt*Ke)*s);
%   Gravity acceleration [m/s^2] to mass position [m]
Y_G = -m*r^2*(L*s+R)/((L*J+L*m*r^2)*s^3 + (J*R+B*L+m*r^2*R)*s^2 + (B*R+Kt*Ke)*s);

% Closed-loop position controller (PID)
P = 500;
I = 500;
D = 50;
Ts = 0.01;  % digital sampling period [s]
% Add extra pole at -2/Ts
% (Ref: Digital Control System Analysis and Design by Phillips, Section 8.10)
s = tf([1 0], 1);
C = P + I/s + D*s/(1 + (Ts/2)*s);

% Closed loop system
%   Reference position [rad] to mass vertical position [m]
Y_r = (C*Kt*r)/((L*J+L*m*r^2)*s^3 + (J*R+B*L+R*m*r^2)*s^2 + (B*R+Kt*Ke)*s + C*Kt*r);
%   Gravity acceleration to motor position [rad]
Y_g = -m*r^2*(L*s+R)/((L*J+L*m*r^2)*s^3 + (J*R+B*L+R*m*r^2)*s^2 + (B*R+Kt*Ke)*s +
C*Kt*r);

%   Mass vertical position [m] to tension [N]
%       Added two far-left poles to fascillitate simulation (otherwise a
%       non-causal transfer function)
T_y = m*s^2/(s+1000)^2;
%   Gravity acceleration to tension [N]
T_g = tf(m,1);


%% ====================
% Some analysis plots
% ====================

% Bode plot for stablity margins
%   Loop transfer function with simple PID
C_simple = P + I/s + D*s;
L = series(C_simple,Y_V);
figure()
margin(L);
grid on
%   Loop transfer function with controller created above
L = series(C,Y_V);
figure()
margin(L);
grid on


%% ====================
% Run and plot simulations
% ====================

% Closed-loop simulation
%   -Constant mass with gravity system
%   -Step reference input
t = 0:0.0001:6;             % time vector for simulation
r = 1*ones(length(t),1);    % reference position
G = g*ones(length(t),1);    % gravity
y_r = lsim(Y_r, r, t);      % position response due to reference input
y_g = lsim(Y_g, G, t);      % position response due to gravity
y = y_r + y_g;              % total position response
e = r-y;                    % position error
v = lsim(C, e, t);          % controller output (motor voltage)
t_y = lsim(T_y, y, t);      % tension response due to mass position
t_g = lsim(T_g, G, t);      % tension response due to gravity
T = t_y + t_g;              % total tension response
figure()
```

```
% Plot position response
subplot(2,2,1)
plot(t, y, 'k', t, r, ':k')
xlabel('Time [s]')
ylabel('Mass position [m]')
grid on
title('Step response')
% Plot position error
subplot(2,2,2)
plot(t, e, 'k')
xlabel('Time [s]')
ylabel('Position error [m]')
grid on
% Plot controller output (motor voltage)
subplot(2,2,3)
plot(t, v, 'k')
xlabel('Time [s]')
ylabel('Motor voltage [V]')
grid on
% Plot tension response
subplot(2,2,4)
plot(t, T, 'k')
xlabel('Time [s]')
ylabel('Tension [N]')
grid on

% satVal = 100;
% vSat = v;
% vSat(vSat>satVal) = satVal;
% vSat(vSat<-satVal) = -satVal;
% subplot(1,4,3)
% plot(t,vSat,'r')
% y_vSat = lsim(Y_V, vSat, t);
% y_gSat = lsim(Y_G, G, t);
% ySat = y_vSat + y_gSat;
% subplot(1,4,1)
% plot(t,ySat,'r')


%% Closed-loop simulation
%   -Constant mass with gravity system
%   -Ramp reference input
t = 0:0.0001:6;             % time vector for simulation
r = 0.1*t';                 % reference position
G = g*ones(length(t),1);    % gravity
y_r = lsim(Y_r, r, t);      % position response due to reference input
y_g = lsim(Y_g, G, t);      % position response due to gravity
y = y_r + y_g;              % total position response
e = r-y;                    % position error
v = lsim(C, e, t);          % controller output (motor voltage)
t_y = lsim(T_y, y, t);      % tension response due to mass position
t_g = lsim(T_g, G, t);      % tension response due to gravity
T = t_y + t_g;              % total tension response
figure()
% Plot position response
subplot(2,2,1)
plot(t, y, 'k', t, r, ':k')
xlabel('Time [s]')
ylabel('Mass position [m]')
grid on
title('Ramp response')
% Plot position error
subplot(2,2,2)
plot(t, e, 'k')
xlabel('Time [s]')
ylabel('Position error [m]')
grid on
```

```
% Plot controller output (motor voltage)
subplot(2,2,3)
plot(t, v, 'k')
xlabel('Time [s]')
ylabel('Motor voltage [V]')
grid on
% Plot tension response
subplot(2,2,4)
plot(t, T, 'k')
xlabel('Time [s]')
ylabel('Tension [N]')
grid on

%% Closed-loop simulation
%   -Constant mass with gravity system
%   -Sine reference input
t = 0:0.0001:10;              % time vector for simulation
r = sin(2*pi*0.5*t)';        % reference position
G = g*ones(length(t),1);     % gravity
y_r = lsim(Y_r, r, t);       % position response due to reference input
y_g = lsim(Y_g, G, t);       % position response due to gravity
y = y_r + y_g;               % total position response
e = r-y;                     % position error
v = lsim(C, e, t);           % controller output (motor voltage)
t_y = lsim(T_y, y, t);       % tension response due to mass position
t_g = lsim(T_g, G, t);       % tension response due to gravity
T = t_y + t_g;               % total tension response
figure()
% Plot position response
subplot(2,2,1)
plot(t, y, 'k', t, r, ':k')
xlabel('Time [s]')
ylabel('Mass position [m]')
grid on
title('Sine response')
% Plot position error
subplot(2,2,2)
plot(t, e, 'k')
xlabel('Time [s]')
ylabel('Position error [m]')
grid on
% Plot controller output (motor voltage)
subplot(2,2,3)
plot(t, v, 'k')
xlabel('Time [s]')
ylabel('Motor voltage [V]')
grid on
% Plot tension response
subplot(2,2,4)
plot(t, T, 'k')
xlabel('Time [s]')
ylabel('Tension [N]')
grid on
```

Table A20. *animatedPlot.m* is a function that creates an animated plot from simulation or experimental data for a CSR of any valid configuration.

```
function animatedPlot(t, positions, cableLengths, anchorPoints, plotCables)
% Plots positions of a cable suspended robot. If multiple time steps are
% given, gives an animation.
%
%   t is a time vector in [s], one time for each end effector position.
%   positions is a matrix with the cartesian position of the end effector at
%       each time step [m]. The three columns of the matrix are the X,Y,Z
%       coordinates of the effector.
%   cableLengths is a matrix with the cable lengths at each time instant [m].
%       One column is required for each cable, and one row for each time.
%   anchorPoints is a matrix with one cartesian position for each cable
%       anchor location [m]. The three columns of the matrix are the X,Y,Z
%       coordinates of anchor points. One row is required for each cable.
%   plotCables is a boolean that selects whether to plot cables in the
%       animation. If no cables are plotted, just the location of the end
%       effector is marked.

    % Get the number of times/positions and cables
    numTimes = length(t);
    [numCables, ~] = size(anchorPoints);

    % If cable lengths not provided, set plotCables to false
    if isempty(cableLengths)
        plotCables = false;
    end

    % Create a figure
    figure();
    subplot(1,1,1);
    hold on;
    % Some plot formatting
    grid on;
    axis square;
    view(-30,15)
    xlabel('X [m]')
    ylabel('Y [m]')
    zlabel('Z [m]')
    % Plot cable anchor points, one at a time
    for c=1:numCables
        % Plot amchor points as filled red circle with line to ground
        stem3(anchorPoints(c,1),anchorPoints(c,2),anchorPoints(c,3),'r','filled')
    end

    % Decrease number of points plotted to avoid slow animation
    n = 1;
    if numTimes>1000
        n = floor(numTimes/1000);
    end

    % Plot end effector positions (and cables, if requested) one time step at a time
    for p=1:n:numTimes
        % If cable plotting is on, clear current plot to make space for
        % updated cables.
        if (plotCables == true)
            % Clear current plot (clear axes)
            cla

            % Plot cable anchor points again, since plot was cleared
            for c=1:numCables
                % Plot amchor points as filled red circle with line to ground

stem3(anchorPoints(c,1),anchorPoints(c,2),anchorPoints(c,3),'r','filled')
            end
        end

        % Display the time in the figure title
        title(sprintf('%0.3f [s]',t(p)))

        % Plot current end effector position as a black point
        plot3(positions(p,1),positions(p,2),positions(p,3),'.k');

        % Plot cables, if requested
        if (plotCables == true)
```

```
            % Calculate cable lengths based on mass position
            [idealCableLengths] = positionToCableLengths(positions(p,:),
anchorPoints);

            % For each cable, determine if there is slack and plot
            for c=1:numCables
                actualLength = cableLengths(p,c);
                idealLength = idealCableLengths(c);
                if (actualLength>idealLength)
                    % Create a curve that illustrates the cable slack
                    a = 0.8*sqrt(actualLength^2 - idealLength^2);
                    v = linspace(0,idealLength);
                    dz = a - a.*(2/idealLength).^2.*(v-idealLength/2).^2;
                    cable = [ linspace(positions(p,1),anchorPoints(c,1))' ...
                              linspace(positions(p,2),anchorPoints(c,2))' ...
                              linspace(positions(p,3),anchorPoints(c,3))'-dz'];
                elseif (actualLength/idealLength)<0.95
                    vec = [positions(p,1)-anchorPoints(c,1) positions(p,2)-
anchorPoints(c,2) positions(p,3)-anchorPoints(c,3)];
                    dir = vec/norm(vec);
                    cable = [anchorPoints(c,1) anchorPoints(c,2) anchorPoints(c,3);
anchorPoints(c,:)+actualLength*dir];
                else
                    % There is no slack, so cable is a straight line from
                    % effector position to anchor point
                    cable = [ positions(p,1)    positions(p,2)    positions(p,3);
                              anchorPoints(c,1) anchorPoints(c,2) anchorPoints(c,3)];
                end
                % Plot the cable as a solid black line
                plot3(cable(:,1),cable(:,2),cable(:,3),'-k');
            end
        end

        % Plot the points now
        drawnow
    end
end
```

Table A21. *cableLengthsToPosition.m* is a function that calculates the most likely position of the CSR end effector given the cable lengths. In other words, this function solves the forward kinematic problem for the CSR.

```
function [position,activeCables,redundantCables] =
cableLengthsToPosition(cableLengths, anchorPoints)
% Calculates the cartesian position of a cable suspended robot end
% effector given the cable lengths and cable anchor positions. This routine
% attempts to deal with redundant cable lengths and/or invalid cables.
%
%   cableLengths: cable lengths in [m]
%   anchorpoints: Cartesian positions of the cable anchor points. Each cable
%       can be seen as originiating from its respective anchor point.
%       Each row of this argument is a cartesian position vector. One row
%       is required for each cable, in the same order as cableLengths.
%   position: the calculated cartesian position of the end effector mass based
%       on the cable lengths given. The position is in the unit of [m].
%   activeCables: list of the three cables that uniquely define the
%       calculated position. The cable numbers are relative to the order of
%       the cables in cableLengths.
%   redundant cables: list of the cables that are long enough to reach the
%       calculated position, but are redundant (they could be removed
%       without affecting the effector position).

    % Get number of cables
    numCables = length(cableLengths);

    % If only one cable is given, solution is immediately known
    if numCables ==1
        disp('Only one cable given.')
        x = anchorPoints(1);
        y = anchorPoints(2);
        z = anchorPoints(3)-cableLengths;
        return
    end

    % If two or more cables are given:
    % ======================================================================
    % Step 1: Create a connection matrix (an adjacency matrix with 1's on
    %         the diagonal)
    % ======================================================================

    % Connection matrix describes the possible connections between cables.
    % Cables i and j can connect with each other if there is a 1 in the
    % matrix location (i,j) or (j,i). This matrix is symmetric.
    connectionMatrix = eye(numCables);
    % Fill the matrix by checking every cable pair for a possible connection
    for i=1:numCables-1
        for j = i+1:numCables
            % Calculate distance between the two cable anchor points
            distanceBetweenAnchorPoints = norm(anchorPoints(i,:)-anchorPoints(j,:));
            % If distance is less than sum of cable lengths, a connection is possible
            if distanceBetweenAnchorPoints <= cableLengths(i)+cableLengths(j)
                connectionMatrix(i,j) = 1;
                connectionMatrix(j,i) = 1;
            end
        end
    end

    % ======================================================================
    % Step 2: Review the connection matrix
    % ======================================================================

    % No cables can connect to each other
    % (i.e. the connection matrix is the identity matrix)
    if sum(connectionMatrix, 'all') == numCables
        disp('No solution exists. The cables are all too short to connect to each
other.')
        x = NaN;
        y = NaN;
        z = NaN;
        return
    end

    % Find the largest group of cables that can connect to each other.
    [~,groupSizes,cableOrder] = findCliques(connectionMatrix, true);

    % If no group of three or more connecting cables exists
    if max(groupSizes)<3
        disp('No solution exists because less than 3 cables can connect with each
other.')
        x = NaN;
        y = NaN;
        z = NaN;
        return
    end

    % Extract the list of valid cables
    [maxGroupSize, maxGroupIndex] = max(groupSizes);
    if maxGroupIndex==1
        connectingCables = cableOrder(1:maxGroupSize);
    else
```

```
        groupStartIndex = sum(groupSizes(1:maxGroupIndex-1))+1;
        connectingCables = cableOrder(groupStartIndex:groupStartIndex+maxGroupSize-1);
    end

    % Generate a list of cable combinations
    cableCombinations = nchoosek(connectingCables,3);
    [numCombinations,~] = size(cableCombinations);

    % Test cable combinations in groups of three
    % Each combination is tested for a possible solution
    for c=1:numCombinations
        % Get cable lengths for current combination
        l1 = cableLengths(cableCombinations(c,1));
        l2 = cableLengths(cableCombinations(c,2));
        l3 = cableLengths(cableCombinations(c,3));
        % Get cable anchor points for current combination
        x1 = anchorPoints(cableCombinations(c,1),1);
        y1 = anchorPoints(cableCombinations(c,1),2);
        z1 = anchorPoints(cableCombinations(c,1),3);
        x2 = anchorPoints(cableCombinations(c,2),1);
        y2 = anchorPoints(cableCombinations(c,2),2);
        z2 = anchorPoints(cableCombinations(c,2),3);
        x3 = anchorPoints(cableCombinations(c,3),1);
        y3 = anchorPoints(cableCombinations(c,3),2);
        z3 = anchorPoints(cableCombinations(c,3),3);

        % Calculate possible solution for end effector position
        x0 = [0,0,0];
        fun = @(pos) [ l1^2-((pos(1)-x1)^2+(pos(2)-y1)^2+(pos(3)-z1)^2);
                       l2^2-((pos(1)-x2)^2+(pos(2)-y2)^2+(pos(3)-z2)^2);
                       l3^2-((pos(1)-x3)^2+(pos(2)-y3)^2+(pos(3)-z3)^2)  ];
        options = optimset('Display','off');
        [candidatePosition,~] = fsolve(fun,x0,options);

        % Test the solution
        %   First, check that the solution is in the allowable area under
        %   the triangle created by the three cables. This ensures the
        %   three cables are in tension.
        %   Height condition:
        maxHeight = min([z1 z2 z3]);
        if candidatePosition(3)>maxHeight
           disp('The candidate position is too high.')
           continue
        end
        %   Ground condition:
        if candidatePosition(3)<0
           disp('The candidate position is below the ground.')
           continue
        end
        %   Convex hull condition:
        a1 = [y2-y1 -(x2-x1)];
        c1 = x1*(y2-y1)-y1*(x2-x1);
        test1 = a1*candidatePosition(1:2)'<=c1;
        a2 = [y3-y2 -(x3-x2)];
        c2 = x2*(y3-y2)-y2*(x3-x2);
        test2 = a2*candidatePosition(1:2)'<=c2;
        a3 = [y1-y3 -(x1-x3)];
        c3 = x3*(y1-y3)-y3*(x1-x3);
        test3 = a3*candidatePosition(1:2)'<=c3;
        if test1+test2+test3<3
            disp('The candidate position is not within the working area of the three
active cables.')
            continue
        end
        %   Second, check if this position is possible considering the
        %   remaining cables (they should all have zero or positive slack)
        cableLengthsFromCandidatePosition = positionToCableLengths(candidatePosition,
anchorPoints);
        for i=1:length(connectingCables)
            actualCableLength = cableLengths(connectingCables(i));
            requiredCableLength =
cableLengthsFromCandidatePosition(connectingCables(i));
            % Check if the required cable length is smaller than the
            % required length to support the candidate effector position
            if actualCableLength-requiredCableLength<-0.001
                aCableIsTooShort = true;
                break
            else
                aCableIsTooShort = false;
            end
        end
        if aCableIsTooShort==true
            disp('The candidate position is not valid because one or more cable(s) is
not long enough to reach the position.')
            disp('A different position will be attempted.')
            continue
        end

        % If all tests passed, the calculated point is a valid solution
        position = candidatePosition;
        activeCables = cableCombinations(c,:);
        disp('A valid position was found!')

        % Make a list of the redundant cables
        i=1;
        redundantCables = connectingCables';
        while i<=length(redundantCables)
            for j=1:3
                if redundantCables(i)==cableCombinations(c,j)
                    redundantCables(i) = [];
                    i=i-1;
                    break
                end
            end
            i=i+1;
        end
        return
    end

    % All combinations of three cables were tested and no valid solution
    % was found.
    disp('No solution found.')
    position = NaN;
    activeCables = NaN;
    redundantCables = NaN;
end
```

Table A22. *pointsPath.m* is a function creates a reference trajectory from a given set of waypoints and time stamps for the same points. The trajectory simply connects the points using straight line segments.

```
function [xr,yr,zr] = pointsPath(xCoordinates, yCoordinates, zCoordinates, timeStamps,
currentTime)
% Used to create a cartesian path from waypoints and time stamps
%
% xCoordinates is a list of x positions in meters
% yCoordinates is a list of y positions in meters
% zCoordinates is a list of z positions in meters
% timeStamps is a list of times in seconds
% currentTime is the current time in seconds
%
% Function returns the x,y,z position corresponding to the current time.
% Lists of cooridnates and time stamps should be the same length.
% Optional: instead of time stamps, give the movement speed in [m/s]

    % Get the number of times to sample
    numTimeSamples = length(currentTime);
    % Get the number of points in the path
    numPoints = length(xCoordinates);
    % Calculate the number of segments in the path
    numSegments = numPoints-1;

    % If time stamps not given, calculate them using a constant movement speed
    if(length(timeStamps)==1 && numPoints>1)
        speed = timeStamps; % Movement speed in [m/s]
        timeStamps = NaN(numPoints,1);  % Create array of time stamps
        timeStamps(1) = 0;  % First time stamp is zero seconds
        for(i=1:numSegments)
            % Changes in coordinates for current segment
            dx = xCoordinates(i+1)-xCoordinates(i);
            dy = yCoordinates(i+1)-yCoordinates(i);
            dz = zCoordinates(i+1)-zCoordinates(i);
            distance = norm([dx,dy,dz],2);
            timeStamps(i+1) = timeStamps(i)+distance/speed;
        end
    end

    % Initialize position arrays to be returned
    xr = NaN(numTimeSamples,1);
    yr = NaN(numTimeSamples,1);
    zr = NaN(numTimeSamples,1);

    % Iterate through time samples
    for(i=1:numTimeSamples)
        % Calculate the current segment number
        segment = find(currentTime(i)>=timeStamps,1,'last');

        if(segment==numPoints)
            % If the current time is beyond the final timestamp, stay at final
position
            x = xCoordinates(end);
            y = yCoordinates(end);
            z = zCoordinates(end);
        else
            % Calculate the current segment start and end times
            segmentStartTime = timeStamps(segment);
            segmentEndTime = timeStamps(segment+1);

            % Calculate current segment progress
            segmentProgress = (currentTime(i)-segmentStartTime)/(segmentEndTime-
segmentStartTime);

            % Calculate start and end coordinates of current segment
            xStart = xCoordinates(segment);
            yStart = yCoordinates(segment);
            zStart = zCoordinates(segment);
            xEnd = xCoordinates(segment+1);
            yEnd = yCoordinates(segment+1);
            zEnd = zCoordinates(segment+1);

            % Calculate current position
            x = xStart + segmentProgress*(xEnd-xStart);
            y = yStart + segmentProgress*(yEnd-yStart);
            z = zStart + segmentProgress*(zEnd-zStart);
        end
        % Return reference position
        xr(i) = x;
        yr(i) = y;
        zr(i) = z;
    end
end
```

Table A23. *positionToCableLengths.m* is a function that calculates the ideal cable lengths needed to produce a given end-effector position for a CSR. In other words, this function solves the inverse kinematics problem.

```
function [cableLengths] = positionToCableLengths(position, anchorPoints)
% Calculates the cable lengths of a cable-suspended robot given a
% cartesian position. Cables are assumed inextensible and taught.
%
%   position: Cartesian position of the end effector in [m]
%   anchorpoints: Cartesian positions of the cable anchor points. Each cable
%       can be seen as originiating from its respective position.

    % Get the number of cables (number of rows in anchorPoints)
    [numCables, ~] = size(anchorPoints);
    % Get the number of positions to calculate (number of rows in position)
    [numPositions, ~] = size(position);

    % Declare the matrix to store and return results
    cableLengths = NaN(numPositions,numCables);

    % The maximum allowable height of the mass is the lowest cable anchor point.
    maxPossibleHeight = min(anchorPoints(:,3));
    % The point must lie in the convex hull of the 2D projection of the anchor
    % points.
    convexHullIndices = convhull(anchorPoints(:,1:2));
    convexHullPoints = anchorPoints(convexHullIndices,:);

    % Loop through positions
    for pos = 1:numPositions
        % First, check if the position is too high
        if position(pos,3)>=maxPossibleHeight
            fprintf('Position %i is above the maximum height.\n', pos);
%             cableLengths(pos,:) = NaN;
%             continue
        end

        % Second, check if the position is too low
        if position(pos,3)<0
            fprintf('Position %i is below ground level.\n', pos);
%             cableLengths(pos,:) = NaN;
%             continue
        end

        % Third, check if the position is within the allowable 2D region
        for point=1:length(convexHullIndices)-1
            % Get point coordinates
            x1 = convexHullPoints(point,1);
            y1 = convexHullPoints(point,2);
            x2 = convexHullPoints(point+1,1);
            y2 = convexHullPoints(point+1,2);

            % Create line from current point to next point
            a = [y2-y1 -(x2-x1)];
            c = x1*(y2-y1)-y1*(x2-x1);

            % Check if position is to the left of the line
            if a*position(pos,1:2)'<=c
                onLeft = true;
            else
                onLeft = false;
            end

            % Check if mass point is to the left of the line
            if ~onLeft
                fprintf('Position %i is not within the allowable space.\n', pos);
%                 cableLengths(pos,:) = NaN;
                break
                continue
            end
        end

        % If above tests passed, calculate length of each cable
        for cable = 1:numCables
            % Ideal cable length is Eucledian distance between point and
            % cable anchor location.
            dx = position(pos,1)-anchorPoints(cable,1);
            dy = position(pos,2)-anchorPoints(cable,2);
            dz = position(pos,3)-anchorPoints(cable,3);
            cableLengths(pos,cable) = norm([dx,dy,dz],2);
        end
    end
end
```

Table A24. *positionToMaxXYForce.m* is a function that calculates the maximum-allowed *x-y* force for a CSR at a given position in order to maintain positive cable tensions.

```
function [Fxy, cableTensions] = positionToMaxXYForce(anchorPoints, position, Fz)
    % Create cable direction matrix
    [numCables, ~] = size(anchorPoints);
    cableDirections = NaN(3,numCables);
    for c=1:numCables
        dx = anchorPoints(c,1)-position(1);
        dy = anchorPoints(c,2)-position(2);
        dz = anchorPoints(c,3)-position(3);
        magnitude = norm([dx dy dz]);
        cableDirections(:,c) = [dx;dy;dz]/magnitude;
    end

    % Setup optimization problem
    x0 = zeros(1,numCables);
    lb = zeros(1,numCables);
    fun = @(x) -norm(cableDirections(1:2,:)*x',2);
    options = optimoptions('fmincon','Display','off');
    [cableTensions, fval] =
fmincon(fun,x0,[],[],cableDirections(3,:),Fz,lb,[],[],options);
    Fxy = -fval;
end
```

Table A25. *positionToMinZForce.m* is a function that calculates the minimum-allowed *z* force for a CSR at a given position in order to maintain positive cable tensions.

```
function [Fz, cableTensions] = positionToMinZForce(anchorPoints, position, Fx, Fy)
    % Create cable direction matrix
    [numCables, ~] = size(anchorPoints);
    cableDirections = NaN(3,numCables);
    for c=1:numCables
```

```
        dx = anchorPoints(c,1)-position(1);
        dy = anchorPoints(c,2)-position(2);
        dz = anchorPoints(c,3)-position(3);
        magnitude = norm([dx dy dz]);
        cableDirections(:,c) = [dx;dy;dz]/magnitude;
    end

    % Setup optimization problem
    x0 = zeros(1,numCables);
    lb = zeros(1,numCables);
    fun = @(x) cableDirections(3,:)*x';
    options = optimoptions('fmincon','Display','off');
    [cableTensions, Fz] =
fmincon(fun,x0,[],[],cableDirections(1:2,:),[Fx;Fy],lb,[],[],options);
end
```

Table A26. *positionToStiffness.m* is a function that calculates the CSR directional stiffness values at a given position of the end effector, assuming that the cables are taut.

```
function [stiffness] = positionToStiffness(position, anchorPoints, mass, method)
% Calculates the stiffness for a given end effector position.
%
%   position: Cartesian position of the end effector in [m]
%   anchorpoints: Cartesian positions of the cable anchor points. Each cable
%       can be seen as originiating from its respective position. One row
%       is required for each cable. Each row has the X,Y,Z location of the
%       cable anchor point.
%   mass: mass of end effector in [kg]
%   method: different methods of solving. See switch case statements below for details
%   stiffness: calculated cable stiffness in [N/m]

    % Get the number of cables (number of rows in anchorPoints)
    [numCables, ~] = size(anchorPoints);

    % The maximum allowable height of the mass is the lowest cable anchor point.
    maxPossibleHeight = min(anchorPoints(:,3));
    % Check if the position is too high
    if position(3)>=maxPossibleHeight
        disp('Position %i is above the maximum height.');
        stiffness = NaN(1,numCables);
        return
    end
    % Check if the position is too low
    if position(3)<0
        disp('Position is below ground level.');
        stiffness = NaN(1,numCables);
        return
    end

    % The point must lie in the convex hull of the 2D projection of the anchor
    % points.
    convexHullIndices = convhull(anchorPoints(:,1:2));
    convexHullPoints = anchorPoints(convexHullIndices,:);
    % Check if the position is within the allowable 2D region
    for point=1:length(convexHullIndices)-1
        % Get point coordinates
        x1 = convexHullPoints(point,1);
        y1 = convexHullPoints(point,2);
        x2 = convexHullPoints(point+1,1);
        y2 = convexHullPoints(point+1,2);

        % Create line from current point to next point
        a = [y2-y1 -(x2-x1)];
        c = x1*(y2-y1)-y1*(x2-x1);

        % Check if position is to the left of the line
        if a*position(1:2)'<=c
            onLeft = true;
        else
            onLeft = false;
        end

        % Check if mass point is to the left of the line
        if ~onLeft
            disp('Position is not within the allowable space.');
            stiffness = NaN(1,numCables);
            return
        end
    end

    % If above tests passed, calculate the stiffness
    % Use whatever solving method was requested
    stiffness = zeros(1,6);
    g = 9.81;   % gravity acceleration [m/s^2]
    switch method
        % Approximate method
        case 1
            % Calculate the cable tensions
            d = 0.001;  % [m]
            cartesianForces = [0;0;mass*g];
            [cableTensions,~] = positionToTension(position, anchorPoints,
cartesianForces, 2);
            x = position(1);
            y = position(2);
            z = position(3);
            for c=1:numCables
                xa = anchorPoints(c,1);
                ya = anchorPoints(c,2);
                za = anchorPoints(c,3);
                % Calculate the original cable length
                l = norm(anchorPoints(c,:)-position);
                % Calculate the cable length after a small positive disturbance
                ldx = norm(anchorPoints(c,:)-(position+[d,0,0]));
                ldy = norm(anchorPoints(c,:)-(position+[0,d,0]));
                ldz = norm(anchorPoints(c,:)-(position+[0,0,d]));
                % Calculate the unstretched cable length and linear stiffness
                functionSelect = 3;
                [l0,k] = tensionFunction(functionSelect, l, [], cableTensions(c));
                % x stiffness
                stiffness(1) = stiffness(1) + k*( 1-(l0/ldx) + l0*(xa-x)*(1/ldx-1/l)/d
);
                % y stiffness
                stiffness(2) = stiffness(2) + k*( 1-(l0/ldy) + l0*(ya-y)*(1/ldy-1/l)/d
);
                % z stiffness
                stiffness(3) = stiffness(3) + k*( 1-(l0/ldz) + l0*(za-z)*(1/ldz-1/l)/d
);
            end

        % Exact method
        case 2
            % Calculate the cable tensions
            cartesianForces = [0;0;mass*g];
            [cableTensions,~] = positionToTension(position, anchorPoints,
cartesianForces, 2);
            x = position(1);
            y = position(2);
            z = position(3);
            for c=1:numCables
                xa = anchorPoints(c,1);
                ya = anchorPoints(c,2);
                za = anchorPoints(c,3);
                % Calculate the original cable length
                l = norm(anchorPoints(c,:)-position);
                % Calculate the unstretched cable length and linear stiffness
                functionSelect = 3;
                [l0,k] = tensionFunction(functionSelect, l, [], cableTensions(c));
                % x stiffness
                stiffness(1) = stiffness(1) + k*( 1-(l0/l) + l0*(xa-x)^2/l^3 );
                % y stiffness
                stiffness(2) = stiffness(2) + k*( 1-(l0/l) + l0*(ya-y)^2/l^3 );
                % z stiffness
                stiffness(3) = stiffness(3) + k*( 1-(l0/l) + l0*(za-z)^2/l^3 );
            end
    end

    % Check calculated stiffness for negative values
    if ~isnan(find(stiffness<0,1))
        disp('The calculated stiffness is negative!')
    end
end
```

Table A27. *positionToTension.m* is a function that calculates the CSR cable tensions at a given position of the end effector, assuming that the cables are taut.

```
function [cableTensions, tensionDirections] = positionToTension(position,
anchorPoints, cartesianForces, method)
% Calculates the cable tensions for a given end effector position. The
% tension is based purely on a static equilibrium analysis.
%
%   position: Cartesian position of the end effector in [m]
%   anchorpoints: Cartesian positions of the cable anchor points. Each cable
%       can be seen as originiating from its respective position. One row
%       is required for each cable. Each row has the X,Y,Z location of the
%       cable anchor point.
%   cartesianForces: desired sum of Cartesian forces (a 3 by 1 vector) [N]
%   method: different methods of solving for the tensions. See switch case
%       statements below for details.
%   cableTensions: cable tensions in [N], one for each active cable
%   tensionDirections: cartesian direction vectors for tensions [m]

    % Get the number of cables (number of rows in anchorPoints)
    [numCables, ~] = size(anchorPoints);

    % The maximum allowable height of the mass is the lowest cable anchor point.
    maxPossibleHeight = min(anchorPoints(:,3));
    % Check if the position is too high
    if position(3)>=maxPossibleHeight
        disp('Position is above the maximum height.');
        cableTensions = NaN(1,numCables);
        tensionDirections = NaN(numCables,3);
        return
    end
    % Check if the position is too low
    if position(3)<0
        disp('Position is below ground level.');
        cableTensions = NaN(1,numCables);
        tensionDirections = NaN(numCables,3);
        return
    end

    % The point must lie in the convex hull of the 2D projection of the anchor
    % points.
    convexHullIndices = convhull(anchorPoints(:,1:2));
    convexHullPoints = anchorPoints(convexHullIndices,:);
    % Check if the position is within the allowable 2D region
    for point=1:length(convexHullIndices)-1
        % Get point coordinates
        x1 = convexHullPoints(point,1);
        y1 = convexHullPoints(point,2);
        x2 = convexHullPoints(point+1,1);
        y2 = convexHullPoints(point+1,2);

        % Create line from current point to next point
        a = [y2-y1 -(x2-x1)];
        c = x1*(y2-y1)-y1*(x2-x1);

        % Check if position is to the left of the line
        if a*position(1:2)'<=c
            onLeft = true;
        else
            onLeft = false;
        end

        % Check if mass point is to the left of the line
        if ~onLeft
            disp('Position is not within the allowable space.');
            cableTensions = NaN(1,numCables);
            tensionDirections = NaN(numCables,3);
            return
        end
    end

    % If above tests passed, calculate the cable tensions

    % Default to method 0 if there are only three cables provided
    if numCables==3
        method = 0;
        disp('Only 3 cables given. Exact solving method will be used.')
    end

    % Use whatever solving method was requested
    switch method
        % Exact inverse method, only three cables were given
        case 0
            % Create direction vector matrix for cables (one column for each cable)
            cableDirections = NaN(3,3);
            for c=1:3
                dx = anchorPoints(c,1)-position(1);
                dy = anchorPoints(c,2)-position(2);
                dz = anchorPoints(c,3)-position(3);
                magnitude = norm([dx dy dz]);
                cableDirections(:,c) = [dx;dy;dz]/magnitude;
            end
            % Solve for the tensions using a the common matrix inverse method
            Fsum = cartesianForces;
            cableTensions = (cableDirections\Fsum)';
            tensionDirections = cableDirections';

        % Exact inverse method, selecting and using only three cables
        %   There are different ways to choose three active cables
        case 1
            % Select three active cables
            % Generate a list of three-cable combinations
            cableCombinations = nchoosek(1:numCables,3);
            [numCombinations,~] = size(cableCombinations);
            for c=1:numCombinations
                % Get cable anchor points for current cable combination
                x1 = anchorPoints(cableCombinations(c,1),1);
                y1 = anchorPoints(cableCombinations(c,1),2);
                x2 = anchorPoints(cableCombinations(c,2),1);
                y2 = anchorPoints(cableCombinations(c,2),2);
                x3 = anchorPoints(cableCombinations(c,3),1);
                y3 = anchorPoints(cableCombinations(c,3),2);
                % Convex hull condition
                a1 = [y2-y1 -(x2-x1)];
                c1 = x1*(y2-y1)-y1*(x2-x1);
                test1 = a1*position(1:2)'<=c1;
                a2 = [y3-y2 -(x3-x2)];
                c2 = x2*(y3-y2)-y2*(x3-x2);
                test2 = a2*position(1:2)'<=c2;
                a3 = [y1-y3 -(x1-x3)];
                c3 = x3*(y1-y3)-y3*(x1-x3);
                test3 = a3*position(1:2)'<=c3;
                % If the point is within the convex hull of the current
                % combination, these cables are kept as the active cables.
                % There may be other valid combinations. The first valid
                % combination is used.
                if test1+test2+test3==3
                    activeCables = cableCombinations(c,:);
                    break
                end
            end
            % Use the three active cables to calculate a tension solution
            % Create direction vector matrix for cables (one column for each cable)
            cableTensions = zeros(1,numCables);
            cableDirections = zeros(3,numCables);
            for ac=1:3
                cableNo = activeCables(ac);
                dx = anchorPoints(cableNo,1)-position(1);
                dy = anchorPoints(cableNo,2)-position(2);
                dz = anchorPoints(cableNo,3)-position(3);
                magnitude = norm([dx dy dz]);
                cableDirections(:,cableNo) = [dx;dy;dz]/magnitude;
            end
            % Solve for the tensions using a the common matrix inverse method
            Fsum = cartesianForces;
            cableTensions(activeCables) = (cableDirections(:,activeCables)\Fsum)';
            tensionDirections = cableDirections';

        % Least squares method, with non-negative tension constraint
        case 2
            % Create direction vector matrix for cables (one column for each cable)
            cableDirections = NaN(3,numCables);
            for c=1:numCables
                dx = anchorPoints(c,1)-position(1);
                dy = anchorPoints(c,2)-position(2);
                dz = anchorPoints(c,3)-position(3);
                magnitude = norm([dx dy dz]);
                cableDirections(:,c) = [dx;dy;dz]/magnitude;
            end
            % Solve for the tensions using an optimization method
            Fsum = cartesianForces;   % Desired summation of forces in Cartesian
directions
            x0 = zeros(1,numCables);
            lb = zeros(1,numCables);
            fun = @(x) norm(Fsum-cableDirections*x');
            options = optimoptions('fmincon','Display','off');
            cableTensions = fmincon(fun,x0,[],[],[],[],lb,[],[],options);
            tensionDirections = cableDirections';

        % Minimize sum of tensions, with non-negative tension constraint
        case 3
            % Create direction vector matrix for cables (one column for each cable)
            cableDirections = NaN(3,numCables);
            for c=1:numCables
                dx = anchorPoints(c,1)-position(1);
                dy = anchorPoints(c,2)-position(2);
                dz = anchorPoints(c,3)-position(3);
                magnitude = norm([dx dy dz]);
                cableDirections(:,c) = [dx;dy;dz]/magnitude;
            end
            % Solve for the tensions using an optimization method
            Fsum = cartesianForces;   % Desired summation of forces in Cartesian
directions
            x0 = zeros(1,numCables);
            lb = zeros(1,numCables);
            fun = @(x) sum(x);
            options = optimoptions('fmincon','Display','off');
            cableTensions =
fmincon(fun,x0,[],[],cableDirections,Fsum,lb,[],[],options);
            tensionDirections = cableDirections';

        % Minimize sum of tensions^2, with non-negative tension constraint
        case 4
            % Create direction vector matrix for cables (one column for each cable)
            cableDirections = NaN(3,numCables);
            for c=1:numCables
                dx = anchorPoints(c,1)-position(1);
                dy = anchorPoints(c,2)-position(2);
                dz = anchorPoints(c,3)-position(3);
                magnitude = norm([dx dy dz]);
                cableDirections(:,c) = [dx;dy;dz]/magnitude;
            end
            % Solve for the tensions using an optimization method
            Fsum = cartesianForces;   % Desired summation of forces in Cartesian
directions
            x0 = zeros(1,numCables);
            lb = zeros(1,numCables);
            fun = @(x) norm(x)^2;
            options = optimoptions('fmincon','Display','off');
            cableTensions =
fmincon(fun,x0,[],[],cableDirections,Fsum,lb,[],[],options);
            tensionDirections = cableDirections';

        otherwise
            disp('Method number not valid.')
            cableTensions = NaN(numCables,1);
            tensionDirections = NaN(numCables,3);
            return
    end

    % Check calculated tensions for negative values
    if ~isnan(find(cableTensions<-0.000001,1))
        disp('At least one of the calculated tensions is negative!')
    end
end
```

Table A28. *tensionFunction.m* is a function that calculates the tension in a single cable given the cable length and stretch amount. Several cable models are tested.

```
function [output,stiffness] = tensionFunction(functionSelect, stretchedLength,
unstretchedLength, tension)
% This function implements a cable tension model and calculates related
% quantities. The first input argument selects the cable model that is used.
% At least two of the three remaining arguments must be given.
% This function calculates and returns the third quantity that is left out.
% If all three quantities are given, only the stiffness is calculated.
%
%   functionSelect: selects which function is used for calculations. See
%       the switch statements below for details.
%   stretchedLength: the instantaneous stretched length of the cable [m]
%   unstretchedLength: the instantaneous unstretched length of the cable [m]
%   tension: the cable tension [N]
%   output: whichever of the three quantities was not provided. NaN if all
%       three quantities are provided.
%   stiffness: the linear stiffness [N/m] of the cable at the unstretched cable
%       length provided.

    % Switch to select stiffness function
    %   Each case must specify a formula for the tension, stretched cable
    %   length, unstretched cable length, and stiffness.
    switch functionSelect
        % Pure linear stiffness based on cable stretch only
        % Negative tension not allowed
        case 1
            k = 2000;                           % linear stiffness coefficient [N/m]

            T = @(l,l0) max(0,k*(l-l0));        % tension [N]
            L = @(t,l0) l0+t/k;                 % stretched length [m]
            L0 = @(t,l) l-t/k;                  % unstretched length [m]
            K = @(l,l0) max(abs(l-l0)/(l-l0),0)*k;  % linear cable stiffness at the
unstretched length [N/m]

        % Stiffness based on cable length and stretch (Hooke's law)
        case 2
            E = 200e9;                          % Elastic modulus (Pa)
            d = 0.0001;                     % Cable diameter [m]
            A = pi*(d/2)^2;                     % Cross sectional area [m^2]

            T = @(l,l0) (E*A/l0)*(l-l0);        % tension [N]
            L = @(t,l0) l0+(t*l0)/(E*A);        % stretched length [m]
            L0 = @(t,l) E*A*l/(t+E*A);          % unstretched length [m]
            K = @(l,l0) (E*A/l0);               % linear cable stiffness at the
unstretched length [N/m]

        % Stiffness based on cable length and stretch (Hooke's law)
        % Negative tension not allowed
        case 3
            E = 200e9;                              % Elastic modulus (Pa)
            d = 0.0001;                             % Cable diameter [m]
            A = pi*(d/2)^2;                         % Cross sectional area [m^2]

            T = @(l,l0) max(0,(E*A/l0)*(l-l0));     % tension [N]
            L = @(t,l0) l0+(t*l0)/(E*A);            % stretched length [m]
            L0 = @(t,l) E*A*l/(t+E*A);              % unstretched length [m]
            K = @(l,l0) max(abs(l-l0)/(l-l0),0)*(E*A/l0);% linear cable stiffness at
the unstretched length [N/m]

        % Stiffness based on cable length and stretch (Hooke's law)
        % Negative tension not allowed
        % Small tension for small slack
        case 4
            E = 200e9;                              % Elastic modulus (Pa)
            d = 0.0001;                           % Cable diameter [m]
            A = pi*(d/2)^2;                         % Cross sectional area [m^2]
            shift = 0.02;                          % shift in tension function

            T = @(l,l0) max(0,(E*A/l0)*(l-l0+shift*l0));    % tension [N]
            L = @(t,l0) (1-shift)*l0+(t*l0)/(E*A);          % stretched length [m]
            L0 = @(t,l) E*A*l/(t+E*A*(1-shift));            % unstretched length [m]
            K = @(l,l0) max(abs(l-l0+shift*l0)/(l-l0+shift*l0),0)*(E*A/l0);% linear
cable stiffness at the unstretched length [N/m]
    end

    % Determine output
    %   The output is whatever variable is not given as an input argument
    %   The stiffenss is always output in addition to 'output'
    if isempty(stretchedLength)
        stretchedLength = L(tension,unstretchedLength);
        output = stretchedLength;
        stiffness = K(stretchedLength,unstretchedLength);

    elseif isempty(unstretchedLength)
        unstretchedLength = L0(tension, stretchedLength);
        output = unstretchedLength;
        stiffness = K(stretchedLength,unstretchedLength);

    elseif isempty(tension)
        tension = T(stretchedLength,unstretchedLength);
        output = tension;
        stiffness = K(stretchedLength,unstretchedLength);

    else
        output = NaN;
        stiffness = K(stretchedLength,unstretchedLength);
    end
end
```

APPENDIX B:

EXPERIMENT CODES

The tables in this appendix contain the Arduino and Python codes used in the experiment work of this dissertation. To run these codes, the files should be organized as shown in Figure 128. Note that there are also two MATLAB files included. These two files were used to plot the experimental results.

```
🗀 Arduino
    🗀 cableRobotController
        cableRobotController.ino
    🗀 nano_encoder
        nano_encoder.ino
    🗀 Sweep
        Sweep.ino

🗀 Python
    🗀 Results
        plotData.m
        pointsPath.m
    kinect_massTracker.py
```

Figure 128. Folder structure for the Arduino and Python experiment codes.

Table A29. *cableRobotController.ino* is an Arduino program that runs on the Arduino Mega 2560 and executes the main experiment routine.

```
// For use with kinect_massTracker.py (when using vision-based control)

// ====================================================================================
//      Include libraries
// ====================================================================================

#include <math.h>      // for math functions
#include <Wire.h>      // for i2c serial communication


// ====================================================================================
//      Initialize variables
// ==================================================================================

// DC motor pin setup (for L298N H-bridge controllers)
// motor a
#define a1 22
#define a2 24
#define aEnable 2
// motor b
#define b1 26
#define b2 28
#define bEnable 3
// motor c
#define c1 30
#define c2 32
#define cEnable 4
// motor d
#define d1 34
#define d2 36
#define dEnable 5

// Geometric constants for the experimental setup
// The following dimensions are for the eye hook through which the cables run
float b = 0.74;                                 // Length of each side of the cage [m]
float h = 0.79;                                 // Height of the cage [m]

// General controller variables
int pwmA, pwmB, pwmC, pwmD;                      // PWM value for each motor driver
int timeDelay;                                   // Time delay between each main loop
[ms] (actual time delay will be slightly longer)
float referencePosition[3];                      // Reference (or desired) cartesian
position (x,y,z) in [m]
float currentTime;                               // current time in [s]
long t0;                                         // zero time in [ms]
long controlTime[2];                             // Time at each controller iteration
in [ms] [0]=current, [1]=last iteration
long dt;                                                                          //
Time between controller iterations in [ms]
long iteration;                                  // Controller iteration count


// For cable-length based control:
long encoderCounts0[4];                          // Initial encoder counts, value sent
by Arduino Nanos, 4 cables
long encoderCounts[4];                           // Current encoder counts, value sent
by Arduino Nanos, 4 cables
// 61.83
float countToLength = 66.0/1000000;              // Convserion from encoder count to
cable length [m]
float la_i, lb_i, lc_i, ld_i;                    // Initial cable lengths in [m]
(calculated from initial encoder counts)
float la_r, lb_r, lc_r, ld_r;                    // Reference (or desired) cable lengths
in [m]
float la, lb, lc, ld;                            // Current cable lengths in [m]
(calculated from encoder counts)
float ki_c;                                      // Integral gain for camera-measured
cable-length error

// For vision-based control
float kinectPosition[3];                         // Cartesian position based on camera
vision (x,y,z) in [m]
float la_c, lb_c, lc_c, ld_c;                    // Cable lengths based on kinect camera
data [m]
float la_c_e[2], lb_c_e[2], lc_c_e[2], ld_c_e[2]; // Cable-length errors (reference minus
actual) [0]=current, [1]=last iteration
float la_c_e_i, lb_c_e_i, lc_c_e_i, ld_c_e_i;    // Cable-length error time integrals
```

```
byte xBytes[2];                                      // Data bytes sent by Kinect for x
position, 2 bytes
byte yBytes[2];                                      // Data bytes sent by Kinect for y
position, 2 bytes
byte zBytes[2];                                      // Data bytes sent by Kinect for z
position, 2 bytes

// For cable length PID control
float kp;                                          // Proportial gain
float ki;                                          // Integral gain
float kd;                                          // Derivative gain
float la_e[2], lb_e[2], lc_e[2], ld_e[2];             // Cable-length errors (reference
minus actual) [0]=current, [1]=last iteration
float la_e_d, lb_e_d, lc_e_d, ld_e_d;          // Cable-length error time derrivatives
float la_e_i, lb_e_i, lc_e_i, ld_e_i;             // Cable-length error time integrals

// Some other options
bool commDebug = false;
bool sendDataToKinect = true;

// Select reference path
// 1. Circle
// 2. T shape
// 3. Calibration path (square with chamfered corners)
// 4. Test path (rotated square with circle within the square)
// 5. Layers path (like Test path, but with height variation)
int path = 3;

// Select a controller to use:
//      1. PID controller based on cable lengths, as measured by encoder
//      2. PID controller based on cable lengths, as measured by encoder, add correction
from Kinect vision
int controller = 2;


// =====================================================================================
//      Setup function (runs once)
// =====================================================================================

void setup() {
  // Begin i2c serial communication (for communication with Arduino Nanos)
  Wire.begin();

  // Begin serial communication (for communication with Kinect Python script)
  Serial.begin(2000000);      // Baud rate

  // Setup motor control pins (digital output)
  // motor a
  pinMode(a1, OUTPUT);
  pinMode(a2, OUTPUT);
  // motor b
  pinMode(b1, OUTPUT);
  pinMode(b2, OUTPUT);
  // motor c
  pinMode(c1, OUTPUT);
  pinMode(c2, OUTPUT);
  // motor d
  pinMode(d1, OUTPUT);
  pinMode(d2, OUTPUT);

  // Set intial cable lengths after zeroing position [m]
  la_i = 0.55;
  lb_i = 0.56;
  lc_i = 0.56;
  ld_i = 0.55;

  // Set intial cable-length errors to zero
  la_e[1] = 0.0;
  lb_e[1] = 0.0;
  lc_e[1] = 0.0;
  ld_e[1] = 0.0;
  la_c_e[1] = 0.0;
  lb_c_e[1] = 0.0;
  lc_c_e[1] = 0.0;
  ld_c_e[1] = 0.0;

  // Set intial cable-length error integrals to zero
  la_e_i = 0.0;
  lb_e_i = 0.0;
  lc_e_i = 0.0;
  ld_e_i = 0.0;
  la_c_e_i = 0.0;
  lb_c_e_i = 0.0;
  lc_c_e_i = 0.0;
  ld_c_e_i = 0.0;

  // Move mass to starting position
  zeroPosition();

  // Get the initial encoder counts from the Arduino Nanos
  getEncoderCounts(encoderCounts0);

  // Reset zero time, controller time, and iteration count
  iteration = 0;
  t0 = millis();
  controlTime[1] = t0;

  // Deault controller time delay
  timeDelay = 10;
}


// ====================================================================================
//      Main loop (runs continuosly)
// ====================================================================================

void loop() {
  // Add delay between loop iterations
  delay(timeDelay);

  // Update controller iteration count
  iteration = iteration + 1;

  // Update current time
  currentTime = (millis()-t0)/1000.0;

  // Update reference position at current time/iteration
  switch (path){
    case 1: // Circle, total runtime about 70s
        circlePath();
        break;

    case 2:{  // Capital 'T' shape, total runtime about 100s
        float xPoints[] = {b/2, 0.5, 0.5, 0.4, 0.4, 0.5, 0.5, 0.6, 0.6, 0.1, 0.1, 0.2,
0.2, 0.3, 0.3, 0.2, 0.2, 0.5};
        float yPoints[] = {b/2, 0.1, 0.2, 0.2, 0.5, 0.5, 0.4, 0.4, 0.6, 0.6, 0.4, 0.4,
0.5, 0.5, 0.2, 0.2, 0.1, 0.1};
        float   zPoints[]   =   {0.10,0.10,0.10,0.10,0.10,   0.10,0.10,0.10,0.10,0.10,
0.10,0.10,0.10,0.10,0.10, 0.10,0.10,0.10};
        float times[] = {0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75,
80, 85};
        pointsPath(xPoints, yPoints, zPoints, times, (int)sizeof(xPoints)/4);
        break;
    }
    case 3:{  // Calibration shape (square with chamfered corners), total runtime about
135s
        float xPoints[] = {b/2, 7*b/8, 7*b/8, 7*b/8, 7*b/8, 6*b/8, 6*b/8, b/2, b/2,
2*b/8, 2*b/8, b/8, b/8, b/8, b/8, b/8, b/8, 2*b/8, 2*b/8, b/2, b/2, 6*b/8, 6*b/8, 7*b/8,
7*b/8, 7*b/8};
        float yPoints[] = {b/2, b/2, b/2, 6*b/8, 6*b/8, 7*b/8, 7*b/8, 7*b/8, 7*b/8,
7*b/8, 7*b/8, 6*b/8, 6*b/8, b/2, b/2, 2*b/8, 2*b/8, b/8, b/8, b/8, b/8, b/8, b/8, 2*b/8,
2*b/8, b/2};
        float zPoints[] = {0.1, 0.1, 0.1, 0.1, 0.1, 0.1,  0.1, 0.1, 0.1, 0.1, 0.1,  0.1,
0.1, 0.1, 0.1, 0.1,  0.1, 0.1, 0.1, 0.1, 0.1,  0.1, 0.1, 0.1, 0.1, 0.1};
        float times[] = {0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75,
80, 85, 90, 95, 100, 105, 110, 115, 120, 125};
        pointsPath(xPoints, yPoints, zPoints, times, (int)sizeof(xPoints)/4);
        break;
    }
    case 4:{  // Test path (rotated square plus circle), total runtime about 80s
        float xPoints[] = {b/2, 7*b/8, b/2, b/8, b/2, 7*b/8, b/2+b/6};
        float yPoints[] = {b/2, b/2, 7*b/8, b/2, b/8, b/2, b/2};
        float zPoints[] = {0.10,0.10,0.10,0.10,0.10,0.10,0.10};
        float times[] = {0, 5, 15, 25, 30, 35, 40};
        testPath(xPoints, yPoints, zPoints, times, (int)sizeof(xPoints)/4);
        break;
    }

    case 5:{  // Layers path (like test path but with change in height), total runtime
about 85s
        float xPoints[] = {b/2, 7*b/8, b/2, b/8,  b/8,  b/2, 7*b/8, b/2+b/6};
        float yPoints[] = {b/2, b/2, 7*b/8, b/2,  b/2,  b/8, b/2, b/2};
        float zPoints[] = {0.095,0.095,0.095,0.095, 0.105,  0.105,0.105,0.105};
        float times[] = {0, 5, 15, 25,  30,  35, 40, 45};
        testPath(xPoints, yPoints, zPoints, times, (int)sizeof(xPoints)/4);
        break;
    }
  }

  // Run controller iteration (each control case should update the motor pwm values)
  switch (controller){
    case  1:
      timeDelay = 10;
      kp = 10000.0;
      ki = 20000.0;
      kd = 300.0;
      controller_PID();
      break;

    case 2:
      // Delay cannot be much lower than 40ms due to camera framerate limitation
      timeDelay = 10;
//      kp = 2000.0;
//      ki = 4000.0;
//      kd = 100.0;
      kp = 10000.0;
      ki = 20000.0;
      kd = 300.0;
      if (currentTime>10){
        ki_c = 0.0;
      }
      else{
        ki_c = 0.0;
        la_c_e_i = 0.0;
        lb_c_e_i = 0.0;
        lc_c_e_i = 0.0;
        ld_c_e_i = 0.0;
      }
      controller_PID_camera();

      break;
  }

  // Write updated motor PWM outputs
  // pwm values are calculated in the controller function in the switch above
  motor(a1, a2, aEnable, pwmA);
  motor(b1, b2, bEnable, pwmB);
  motor(c1, c2, cEnable, pwmC);
  motor(d1, d2, dEnable, pwmD);
}


// ==================================================================================
//      Trajectory path functions
// ==================================================================================

// Circle path (starts small and spirals out to maximum radius, constant height)
void circlePath(){
  // Circular motion (radius of circle grows to max value and then stays at max radius)
  float radius = currentTime*0.008;
  // Height hoes from max to a minimum value and then stays there
  float height = 0.7 - currentTime*0.08;

  if(radius>0.2){
    radius = 0.2;
  }

  if(height<0.1){
    height = 0.1;
  }

  referencePosition[0] = b/2 + radius*cos(currentTime*0.2);
  referencePosition[1] = b/2 + radius*sin(currentTime*0.2);
  referencePosition[2] = height;
}

// Straight-lines path defined by x,y,z points and time stamps
void pointsPath(float *xCoordinates, float *yCoordinates, float *zCoordinates, float
*timeStamps, int numPoints){

  // Start time of first segment [s]
  float pathStartTime = 10.0;

  // Initial operation is to lower the mass to starting position
  float x = b/2.0;
  float y = b/2.0;
  float z = 0.65 - currentTime*0.08;
  // Height remains at 0.1 m after reaching it
  if(z<0.1){
    z = 0.1;
  }

  // Path starts at start time
  int segment;
  if(currentTime>=pathStartTime){
    // Find the segment number according to the current time
    float currentPathTime = currentTime-pathStartTime;
    for(int i=0;i<numPoints;i++){
      if(currentPathTime>=timeStamps[i]){
        segment = i;
      }
    }

    // If time is past last time stamp, stay at final position
    if(segment==numPoints-1){
      x = xCoordinates[numPoints-1];
      y = yCoordinates[numPoints-1];
      z = zCoordinates[numPoints-1];
    }
    else{
      // Current segment start and end times
      float segmentStartTime = timeStamps[segment];
      float segmentEndTime = timeStamps[segment+1];

      // Segment progress
      float   segmentProgress   =   (currentPathTime-segmentStartTime)/(segmentEndTime-
segmentStartTime);

      // Calcuate x,y,z according to segment progress
      x    =    xCoordinates[segment]    +    segmentProgress*(xCoordinates[segment+1]-
xCoordinates[segment]);
      y    =    yCoordinates[segment]    +    segmentProgress*(yCoordinates[segment+1]-
yCoordinates[segment]);
      z    =    zCoordinates[segment]    +    segmentProgress*(zCoordinates[segment+1]-
zCoordinates[segment]);
    }
  }

  // Store reference position in array
  referencePosition[0] = x;
  referencePosition[1] = y;
  referencePosition[2] = z;
}

// Identical to pointsPath, but with a never-ending circle at the end of the path.
//  Last  point  of  path  should  be  starting  position  of  circle  (x-y  cooridnates:
b/2+circleRadius, b/2)
void  testPath(float  *xCoordinates,  float  *yCoordinates,  float  *zCoordinates,  float
*timeStamps, int numPoints){
  // Start time of first segment [s]
  float pathStartTime = 10.0;

  // Radius of circle at the end of the path
  float radius = b/6.0;         // [m]
  float circlePeriod = 20.0;    // [s]

  // Initial operation is to lower the mass to starting position
  float x = b/2.0;
  float y = b/2.0;
  float z = 0.65 - currentTime*0.08;
  // Height remains at 0.1 m after reaching it
  if(z<0.1){
    z = 0.1;
  }

  // Path starts at start time
  int segment;
  if(currentTime>=pathStartTime){
    // Find the segment number according to the current time
    float currentPathTime = currentTime-pathStartTime;
    for(int i=0;i<numPoints;i++){
      if(currentPathTime>=timeStamps[i]){
        segment = i;
      }
    }

    // If time is past last time stamp, move in circle
    if(segment==numPoints-1){
      z = zCoordinates[numPoints-1];
      x      =      b/2.0      +      radius*cos((currentPathTime-timeStamps[numPoints-
1])*2*3.14159/circlePeriod);
      y      =      b/2.0      +      radius*sin((currentPathTime-timeStamps[numPoints-
1])*2*3.14159/circlePeriod);
    }
    else{
      // Current segment start and end times
      float segmentStartTime = timeStamps[segment];
      float segmentEndTime = timeStamps[segment+1];

      // Segment progress
      float   segmentProgress   =   (currentPathTime-segmentStartTime)/(segmentEndTime-
segmentStartTime);

      // Calcuate x,y,z according to segment progress
      x    =    xCoordinates[segment]    +    segmentProgress*(xCoordinates[segment+1]-
xCoordinates[segment]);
      y    =    yCoordinates[segment]    +    segmentProgress*(yCoordinates[segment+1]-
yCoordinates[segment]);
      z    =    zCoordinates[segment]    +    segmentProgress*(zCoordinates[segment+1]-
zCoordinates[segment]);
    }
  }

  // Store reference position in array
  referencePosition[0] = x;
  referencePosition[1] = y;
  referencePosition[2] = z;
}

// =======================================================================================
//      Controller functions
// =======================================================================================

// PID controller based on cable lengths
void controller_PID(){
  // Calculate time step in [ms] from last controller iteration to now
  controlTime[0] = millis();
  dt = controlTime[0] - controlTime[1];
  // Save current time for next iteration
  controlTime[1] = controlTime[0];


  // Get the reference (desired) cable lengths given the current reference position
  convertPositionToCableLengths(referencePosition, &la_r, &lb_r, &lc_r, &ld_r);

  // Update the current cable lengths (la, lb, lc, ld)
  getCableLengths();


  // Calculate current cable-length errors (reference minus current)
  la_e[0] = la_r - la;
  lb_e[0] = lb_r - lb;
  lc_e[0] = lc_r - lc;
  ld_e[0] = ld_r - ld;

  // Calculate cable-length error integrals, using the trapezoidal method
  la_e_i = la_e_i + (la_e[0]+la_e[1])*dt/2000.0;
  lb_e_i = lb_e_i + (lb_e[0]+lb_e[1])*dt/2000.0;
  lc_e_i = lc_e_i + (lc_e[0]+lc_e[1])*dt/2000.0;
  ld_e_i = ld_e_i + (ld_e[0]+ld_e[1])*dt/2000.0;

  // Calculate cable-length error time derrivatives
  la_e_d = (la_e[0]-la_e[1])*1000.0/dt;
  lb_e_d = (lb_e[0]-lb_e[1])*1000.0/dt;
  lc_e_d = (lc_e[0]-lc_e[1])*1000.0/dt;
  ld_e_d = (ld_e[0]-ld_e[1])*1000.0/dt;

  // Save current cable-length errors for next controller iteration
  la_e[1] = la_e[0];
  lb_e[1] = lb_e[0];
  lc_e[1] = lc_e[0];
  ld_e[1] = ld_e[0];
  // Calculate pwm outputs based on the control law
  pwmA = la_e[0]*kp + la_e_i*ki + la_e_d*kd;
  pwmB = lb_e[0]*kp + lb_e_i*ki + lb_e_d*kd;
  pwmC = lc_e[0]*kp + lc_e_i*ki + lc_e_d*kd;
  pwmD = ld_e[0]*kp + ld_e_i*ki + ld_e_d*kd;
}


// PID controller based on cable lengths
void controller_PID_camera(){
  // Calculate time step in [ms] from last controller iteration to now
  controlTime[0] = millis();
  dt = controlTime[0] - controlTime[1];
  // Save current time for next iteration
  controlTime[1] = controlTime[0];


  // Get the reference (desired) cable lengths
  convertPositionToCableLengths(referencePosition, &la_r, &lb_r, &lc_r, &ld_r);

  // Get the current cable lengths
  getCableLengths();


  if(iteration%3 == 0){
    // Get position according to camera
    getKinectData();
    // Get cable lengths according to camera
    convertPositionToCableLengths(kinectPosition, &la_c, &lb_c, &lc_c, &ld_c);
    // Calculate current cable-length errors, according to the camera measurements
    la_c_e[0] = la_r - la_c;
    lb_c_e[0] = lb_r - lb_c;
    lc_c_e[0] = lc_r - lc_c;
    ld_c_e[0] = ld_r - ld_c;
    // Calculate camera cable-length error integrals, using the trapezoidal method
    la_c_e_i = la_c_e_i + (la_c_e[0]+la_c_e[1])*dt/2000.0;
    lb_c_e_i = lb_c_e_i + (lb_c_e[0]+lb_c_e[1])*dt/2000.0;
    lc_c_e_i = lc_c_e_i + (lc_c_e[0]+lc_c_e[1])*dt/2000.0;
    ld_c_e_i = ld_c_e_i + (ld_c_e[0]+ld_c_e[1])*dt/2000.0;

    // Save current camera length error values for next iteration
    la_c_e[1] = la_c_e[0];
    lb_c_e[1] = lb_c_e[0];
    lc_c_e[1] = lc_c_e[0];
    ld_c_e[1] = ld_c_e[0];
  }
  // Adjust the reference cable lengths using the camera feedback
  la_r = la_r + ki_c*la_c_e_i;
  lb_r = lb_r + ki_c*lb_c_e_i;
  lc_r = lc_r + ki_c*lc_c_e_i;
  ld_r = ld_r + ki_c*ld_c_e_i;


  // Calculate current cable-length errors, according to the encoder measurements
  la_e[0] = la_r - la;
  lb_e[0] = lb_r - lb;
  lc_e[0] = lc_r - lc;
  ld_e[0] = ld_r - ld;

  // Calculate encoder cable-length error integrals, using the trapezoidal method
  //    The 2000 factor accounts for the 1/2 of the trapezoidal method and and the
  //    dt unit in milliseconds.
  la_e_i = la_e_i + (la_e[0]+la_e[1])*dt/2000.0;
  lb_e_i = lb_e_i + (lb_e[0]+lb_e[1])*dt/2000.0;
  lc_e_i = lc_e_i + (lc_e[0]+lc_e[1])*dt/2000.0;
  ld_e_i = ld_e_i + (ld_e[0]+ld_e[1])*dt/2000.0;
  // Calculate cable-length error time derrivatives
  la_e_d = (la_e[0]-la_e[1])*1000.0/dt;
  lb_e_d = (lb_e[0]-lb_e[1])*1000.0/dt;
  lc_e_d = (lc_e[0]-lc_e[1])*1000.0/dt;
  ld_e_d = (ld_e[0]-ld_e[1])*1000.0/dt;

  // Save current cable-length errors for next controller iteration
  la_e[1] = la_e[0];
  lb_e[1] = lb_e[0];
  lc_e[1] = lc_e[0];
  ld_e[1] = ld_e[0];

  // Calculate pwm outputs based on the control law
  pwmA = (la_e[0]*kp + la_e_i*ki + la_e_d*kd);
  pwmB = (lb_e[0]*kp + lb_e_i*ki + lb_e_d*kd);
  pwmC = (lc_e[0]*kp + lc_e_i*ki + lc_e_d*kd);
  pwmD = (ld_e[0]*kp + ld_e_i*ki + ld_e_d*kd);
}


// ===================================================================================
//      Hardware helper functions and routines
// ===================================================================================

// Move mass to starting position
void zeroPosition(){

  //  Serial.println("   Moving to start position...");

  // Slowly wind all cables for three seconds
  motor(a1, a2, aEnable, -50);
  motor(b1, b2, bEnable, -50);
  motor(c1, c2, cEnable, -50);
  motor(d1, d2, dEnable, -50);
  delay(2000);

 // Faster wind all cables for two seconds, to remove any slack
  motor(a1, a2, aEnable, -100);
  motor(b1, b2, bEnable, -100);
  motor(c1, c2, cEnable, -100);
  motor(d1, d2, dEnable, -100);
  delay(1000);

  // Faster wind all cables for two seconds, to remove any slack
  motor(a1, a2, aEnable, -200);
  motor(b1, b2, bEnable, -200);
  motor(c1, c2, cEnable, -200);
  motor(d1, d2, dEnable, -200);
  delay(1000);

//  Serial.println("   Zeroing position finishes in...");
  delay(1000);
//  Serial.println("   3");
  delay(1000);
//  Serial.println("   2");
  delay(1000);
//  Serial.println("   1");
  delay(1000);

  // Set motor speeds to zero
//  motor(a1, a2, aEnable, 0);
//  motor(b1, b2, bEnable, 0);
//  motor(c1, c2, cEnable, 0);
//  motor(d1, d2, dEnable, 0);
}
```

```
// Motor driver PWM function (intended for use with L298N driver board)
// PWM is from -255 to 255, sign indicates direction of movement
void motor(int in1, int in2, int enablePin, int pwm) {
  int maxPWM = 255;
  int minPWM = 0;
  // Set driver to stop motor if speed is zero
  if(pwm==0){
    digitalWrite(in1,LOW);
    digitalWrite(in2,LOW);
  }
  // Set driver to positive motor direction
  else if(pwm>0){
    digitalWrite(in1,LOW);
    digitalWrite(in2,HIGH);
  }
  // Set driver to negative motor direction
  else if(pwm<0){
    digitalWrite(in1,HIGH);
    digitalWrite(in2,LOW);
  }
  // Limit maximum PWM value
  if(abs(pwm)>maxPWM){
    pwm = maxPWM;
  }
  // Set minimum PWM value
  else if(abs(pwm)<minPWM){
    pwm = minPWM;
  }

  // Write PWM to pin
  analogWrite(enablePin, abs(pwm));   // Write PWM to motor
}


// Get encoder counts from Arduino Nanos using i2c
// countVariable is a pointer to the first element in the count array (array of longs)
void getEncoderCounts(long *countVariable){
  // Initialize byte arrays for storing the incoming serial data
  byte bytesA[4], bytesB[4], bytesC[4], bytesD[4];

  // Request 4 bytes from each Arduino Nano
  Wire.requestFrom(1,4);
  while(Wire.available()<4){} // wait until 4 bytes are available to read
  for(int i=0; i<4; i++){
    bytesA[i] = Wire.read();
  }
  Wire.requestFrom(2,4);
  while(Wire.available()<4){}
  for(int i=0; i<4; i++){
    bytesB[i] = Wire.read();
  }
  Wire.requestFrom(3,4);
  while(Wire.available()<4){}
  for(int i=0; i<4; i++){
    bytesC[i] = Wire.read();
  }
  Wire.requestFrom(4,4);
  while(Wire.available()<4){}
  for(int i=0; i<4; i++){
    bytesD[i] = Wire.read();
  }

  // Empty any extra data in the wire buffer (to deal with any mishandled bytes)
  while(Wire.available()>0){
    Wire.read();
  }
  // Convert the data bytes to long numbers (4-byte, signed)
  // note: countVariable[i] is equivalent to *(countVariable+1)
  //  i.e. countVariable[i] gives the actual value of the array at i
  countVariable[0] = bytesToLong(bytesA);
  countVariable[1] = bytesToLong(bytesB);
  countVariable[2] = bytesToLong(bytesC);
  countVariable[3] = bytesToLong(bytesD);
}


// Get position data from the Kinect camera
void getKinectData(){
  // Request data from Kinect
  Serial.write(1);

  // Wait for all 6 bytes to be available
  while(Serial.available()<6){}

  // Read 6 data bytes
  xBytes[0] = Serial.read();
  xBytes[1] = Serial.read();
  yBytes[0] = Serial.read();
  yBytes[1] = Serial.read();
  zBytes[0] = Serial.read();
  zBytes[1] = Serial.read();

  // Convert the data bytes to position in [m]
  kinectPosition[0] = bytesToInt(xBytes)/10000.0;
  kinectPosition[1] = bytesToInt(yBytes)/10000.0;
  kinectPosition[2] = bytesToInt(zBytes)/10000.0;

  // Send data back to Kinect (for data checking, turn on commDebug in Python code)
  if (commDebug == true){
    Serial.write(xBytes, 2);
    Serial.write(yBytes, 2);
    Serial.write(zBytes, 2);
  }
  // Send full data set back to Kinect (set sendDataToKinect to true above)
  else if(sendDataToKinect == true){
    float                                    data[8]                                     =
{currentTime,kinectPosition[0],kinectPosition[1],kinectPosition[2],la,lb,lc,ld};
    float* dataPointer = data;
    for (int i=0;i<8;i++){
      byte* b = (byte*) (dataPointer+i);
      Serial.write(b[0]);
      Serial.write(b[1]);
      Serial.write(b[2]);
      Serial.write(b[3]);
    }
  }
}


// ====================================================================================
//      Math and conversion functions
// ====================================================================================

// Calculate the current cable lengths based on encoder counts
void getCableLengths(){
  // Get current encoder counts (request data from Arduino nanos)
  getEncoderCounts(encoderCounts);

  // Update actual cable lengths based on encoder count numbers
  la = la_i - (encoderCounts[0]-encoderCounts0[0])*countToLength;
  lb = lb_i + (encoderCounts[1]-encoderCounts0[1])*countToLength;
  lc = lc_i - (encoderCounts[2]-encoderCounts0[2])*countToLength;
  ld = ld_i + (encoderCounts[3]-encoderCounts0[3])*countToLength;
}

// Calculate four ideal cable lengths based on target position
// Position is the cartesian values x, y, z in [m]
// lengthA, lengthB, lengthC, and lengthD are pointers to floats, where the calculated
cable lengths in [m] will be stored
void convertPositionToCableLengths(float position[3], float *lengthA, float *lengthB,
float *lengthC, float *lengthD){
  // Extract x, y, z from position array
  float x = position[0];
  float y = position[1];
  float z = position[2];

  // Convert cartesian position to cable lengths
  *lengthA = sqrt(pow(x, 2) + pow(y, 2) + pow(h - z, 2));
  *lengthB = sqrt(pow(b - x, 2) + pow(y, 2) + pow(h - z, 2));
  *lengthC = sqrt(pow(b - x, 2) + pow(b - y, 2) + pow(h - z, 2));
  *lengthD = sqrt(pow(x, 2) + pow(b - y, 2) + pow(h - z, 2));
}


// Convert 2 bytes of data (least siginificant byte first) to an integer
long bytesToInt(byte byteArray[2]){
  // initialize int variable, 2-byte precision, signed
  int intValue = 0;

  // bytes are assumed to be in little-endian format (least significant byte first)
  intValue += (int)byteArray[0];             // least significant byte to integer
  intValue += (int)byteArray[1] << 8; // bit shift second byte by 8 bits

  return intValue;
}


// Convert 4 bytes of data (least siginificant byte first) to long
long bytesToLong(byte byteArray[4]){
  // initialize long variable, 4-byte precision, signed
  long longValue = 0L;

  // bytes are assumed to be in little-endian format (least significant byte first)
  longValue += (long)byteArray[0];                  // least significant byte to long
  longValue += (long)byteArray[1] << 8;             // bit shift second byte by 8 bits
  longValue += (long)byteArray[2] << 16;     // bit shift third byte by 16 bits
  longValue += (long)byteArray[3] << 24;     // bit shift fourth byte by 24 bits

  return longValue;
}
```

Table A30. *nano_encoder.ino* is an Arduino program that runs on the four Arduino Nano boards, one for each DC motor. This program keeps count of the encoder pulses.

```
// Encoder counter (use with Arduino nano)


// library for i2c serial communication
#include <Wire.h>


// define channel pin numbers
// chanA and chanB are from the encoder
#define chanA 2
#define chanB 3
```

```
// initialize variables
volatile long count = 0L;     // encoder count
int b;                                        // channel B value
byte byteArray[4];


void setup() {
  // join i2c bus with given address number
  Wire.begin(3);
  // register function to run when data is requested
  Wire.onRequest(requestEvent);
  // initialize interrupt for one encoder channel
  attachInterrupt(digitalPinToInterrupt(chanA), interruptRoutine, RISING);
}


// empty loop since interrupt is always listening to channel a of the encoder
void loop() {
}


// when requsted, sen current encoder count through i2c
void requestEvent() {
  // convert long to bytes using little-endian (least significant byte first)
  byteArray[0] = count & 255;
  byteArray[1] = (count>>8) & 255;
  byteArray[2] = (count>>16) & 255;
  byteArray[3] = (count>>24) & 255;
  Wire.write(byteArray, 4);
}


// routine to run when encoder channel a experiences a rising edge
void interruptRoutine(){
  // get value of encoder channel b
  b = digitalRead(chanB);

  // determine direction of encoder rotation and update count accordingly
  // we use the current value of channel b
  // we know channel a is high
  if(b != 1){
      count = count + 1L;
    }
  else{
      count = count - 1L;
    }
}
```

Table A31. *Sweep.ino* is an Arduino program that runs on the Arduino Uno board. This program controls the servo that actuates the cable-disturbance arm pictured in Figure 100.

```
#include <Servo.h>

Servo myservo;  // create servo object to control a servo
// twelve servo objects can be created on most boards

int pos = 0;    // variable to store the servo position

void setup() {
  myservo.attach(3);  // attaches the servo on pin 9 to the servo object
  myservo.write(0);
  while(digitalRead(7)==LOW){}
}

void loop() {
  for (pos = 0; pos <= 180; pos += 1) { // goes from 0 degrees to 180 degrees
    // in steps of 1 degree
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(60);                       // waits 15ms for the servo to reach the position
  }
  for (pos = 180; pos >= 0; pos -= 1) { // goes from 180 degrees to 0 degrees
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(60);                       // waits 15ms for the servo to reach the position
  }
}
```

Table A32. *plotData.m* is a MATLAB script that reads a data file from an experiment trial and plots the recorded values.

```
% Plots experiment results for a 4-cable suspended robot.
%   -Run 'kinect_massTracker.py'
%   -Results are saved in a 'trajectory.dat' file. Place the file in the
%   same folder as this plotting script, and name as desired.
%   -Type trajectory file name below and run script.
% close all;
clear all; clc;


% Data format:  time[s] x[m] y[m] z[m]
%               or
%               time[s] x[m] y[m] z[m] la[m] lb[m] lc[m] ld[m]
%   with a space between values and a new line for each time
fileName = 'CL_levels_5mm_k3_1.dat';
refPath = 5;
camAdjust = [0.02,-0.01,0.03]; %[m]
camAdjust = 0*camAdjust; %[m]


% Open and read file with raw trajectory data
fileId = fopen(fileName,'rt');
dataCell = textscan(fileId, '%f %f %f %f %f %f %f %f', 'Delimiter', ' ');
dataMat = cell2mat(dataCell);
if(isnan(dataMat(1,5)))
    cableLenghtsIncluded = false;
else
    cableLenghtsIncluded = true;
end
fclose(fileId);

% Find indices for data trimming
t = dataMat(:,1);
tStart = 10;
tChangeColor = t(end);
tFinal = t(end);
i0 = find(t>=tStart,1,'first');
i1 = find(t>=tChangeColor,1,'first');
i2 = find(t>=tFinal,1,'first');

% Extract cartesian position data
x = dataMat(:,2)-camAdjust(1);
y = dataMat(:,3)-camAdjust(2);
z = dataMat(:,4)-camAdjust(3);
% First segment
t1 = t(i0:i1);
x1 = x(i0:i1);
```

```
y1 = y(i0:i1);
z1 = z(i0:i1);
% Second segment
t2 = t(i1:i2);
x2 = x(i1:i2);
y2 = y(i1:i2);
z2 = z(i1:i2);
% If cable lengths were saved, extract the length data
if cableLenghtsIncluded
    la = dataMat(:,5);
    lb = dataMat(:,6);
    lc = dataMat(:,7);
    ld = dataMat(:,8);
    % First segment
    la1 = la(i0:i1);
    lb1 = lb(i0:i1);
    lc1 = lc(i0:i1);
    ld1 = ld(i0:i1);
    % Second segment
    la2 = la(i1:i2);
    lb2 = lb(i1:i2);
    lc2 = lc(i1:i2);
    ld2 = ld(i1:i2);
end

% Plot time period between samples
figure;
dt1 = t1(2:end) - t1(1:end-1);
dt2 = t2(2:end) - t2(1:end-1);
plot([dt1;dt2]*1000, '.k')
ylabel('Time period [ms]')
xlabel('Sample number')
ylim([0 100]);
grid on;

% Plot x,y,z seperately vs time
figure;
subplot(3,1,1);
plot(t1, x1, '.k', t2, x2, '.r')
ylabel('X [m]');
xlim([10,70])
grid on;
subplot(3,1,2);
grid on;
plot(t1, y1, '.k', t2, y2, '.r')
ylabel('Y [m]');
xlim([10,70])
grid on;
subplot(3,1,3);
grid on;
plot(t1, z1, '.k', t2, z2, '.r')
hold on
plot([0,35,40,70],[0.095,0.095,0.10,0.10],'-r')
ylabel('Z [m]');
xlim([10,70])
grid on;
xlabel('Time[s]')

% Plot cable lengths seperately vs time
if cableLenghtsIncluded
    figure;
    subplot(4,1,1);
    plot(t1, la1, '.k', t2, la2, '.r')
    ylabel('La [m]');
    xlim([10,70])
    grid on;
    subplot(4,1,2);
    grid on;
    plot(t1, lb1, '.k', t2, lb2, '.r')
    ylabel('Lb [m]');
    xlim([10,70])
    grid on;
    subplot(4,1,3);
    grid on;
    plot(t1, lc1, '.k', t2, lc2, '.r')
    ylabel('Lc [m]');
    grid on;
    xlim([10,70])
    subplot(4,1,4);
    grid on;
    plot(t1, ld1, '.k', t2, ld2, '.r')
    ylabel('Ld [m]');
    grid on;
    xlabel('Time [s]')
    xlim([10,70])
end

% Plot x,y,z in 3D space
figure;
plot3(x1,y1,z1,'.k', x2,y2,z2,'.r')
hold on;
xlabel('X [m]');
ylabel('Y [m]');
zlabel('Z [m]');
xlim([0 0.75]);
ylim([0 0.75]);
zlim([0 1.0]);
grid on;
axis square

switch refPath
    case 1 % Circle
        theta = linspace(0,2*pi);
        % Reference equation from Arduino code:
        b = 0.74;   % m
        h = 0.79;   % m
        x = b/2+0.2*sin(theta);
        y = b/2+0.2*cos(theta);
        z = 0.1*ones(length(theta),1);
        plot3(x,y,z,'g');

    case 2 % T shape
        % Plot T points path
        b = 0.74; % m
        xPoints = [b/2 0.5 0.5 0.4 0.4 0.5 0.5 0.6 0.6 0.1 0.1 0.2 0.2 0.3 0.3 0.2 0.2
0.5];
        yPoints = [b/2 0.1 0.2 0.2 0.5 0.5 0.4 0.4 0.6 0.6 0.4 0.4 0.5 0.5 0.2 0.2 0.1
0.1];
        zPoints = 0.1*ones(1,length(xPoints));
        timeStamps = 0:5:85;
        [xr,yr,zr] = pointsPath(xPoints, yPoints, zPoints, timeStamps, t);
        plot3(xr,yr,zr,'g');

    case 3 % Calibration path (square with chamfered corners)
        b = 0.74; % m
        xPoints = [b/2 7*b/8 7*b/8 7*b/8 7*b/8 6*b/8 6*b/8 b/2 b/2 2*b/8 2*b/8 b/8 b/8
b/8 b/8 b/8 b/8 2*b/8 2*b/8 b/2 b/2 6*b/8 6*b/8 7*b/8 7*b/8 7*b/8];
        yPoints = [b/2 b/2 b/2 6*b/8 6*b/8 7*b/8 7*b/8 7*b/8 7*b/8 7*b/8 7*b/8 6*b/8
6*b/8 b/2 b/2 2*b/8 2*b/8 b/8 b/8 b/8 b/8 b/8 2*b/8 2*b/8 b/2];
        zPoints = 0.1*ones(1,26);
        timeStamps = 0:5:125;
        [xr,yr,zr] = pointsPath(xPoints, yPoints, zPoints, timeStamps, t);
        plot3(xr,yr,zr,'g');
    case 4  % Test path (rotated square with circle within the square)
        b = 0.74; % m
        xPoints = [b/2 7*b/8 b/2 b/8 b/2 7*b/8 b/2+b/6];
        yPoints = [b/2 b/2 7*b/8 b/2 b/8 b/2 b/2];
        zPoints = 0.1*ones(1,length(xPoints));
        timeStamps = [0, 5, 15, 25, 30, 35, 40];
        [xr,yr,zr] = pointsPath(xPoints, yPoints, zPoints, timeStamps, t);
        plot3(xr,yr,zr,'g');
        theta = linspace(0,2*pi);
        x = b/2+b/6*cos(theta);
        y = b/2+b/6*sin(theta);
        z = 0.1*ones(length(theta),1);
        plot3(x,y,z,'g');

     case 5  % Layers path (like test path but with height variation)
        dh = 0.010;    % m
        b = 0.74; % m
        xPoints = [b/2 7*b/8 b/2 b/8 b/8 b/2 7*b/8 b/2+b/6];
        yPoints = [b/2 b/2 7*b/8 b/2 b/2 b/8 b/2 b/2];
        zPoints = [0.095 0.095 0.095 0.095 0.095+dh 0.095+dh 0.095+dh 0.095+dh];
        timeStamps = [0, 5, 15, 25, 30, 35, 40, 45];
        [xr,yr,zr] = pointsPath(xPoints, yPoints, zPoints, timeStamps, t);
        plot3(xr,yr,zr,'g');
        theta = linspace(0,2*pi);
        x = b/2+b/6*cos(theta);
        y = b/2+b/6*sin(theta);
        z = (0.095+dh)*ones(length(theta),1);
        plot3(x,y,z,'g');
end


%% Plot cable length
addpath('C:\Users\ivan\OneDrive\Documents\PhD\CableFDM\cableFDM\Matlab\FinalFiles\util
ities')

measuredLength = la;
B = 0.74;
H = 0.79;
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];    % anchor points for 4 cable
[cableLengths] = positionToCableLengths([xr yr zr], anchorPoints);
referenceLength = cableLengths(:,1);
figure()
subplot(2,1,1)
plot(t,referenceLength,'r',t-10, measuredLength,'-.k')
grid on
xlim([0 40])
ylim([0.7 1.1])
ylabel('Cable length [m]')
legend('Reference','Measured')
subplot(2,1,2)
a = find(t>=10,1,'first');
b = find(t>=50,1,'first');
plot(t(1:b-a), (referenceLength(1:b-a)-measuredLength(a:b-1))*1000,'k')
grid on
xlim([0 40])
xlabel('Time [s]')
ylabel('Length error [mm]')


%% Anchor point disturbance prediction algorithm
addpath('C:\Users\ivan\OneDrive\Documents\PhD\CableFDM\cableFDM\Matlab\FinalFiles\anal
ysis')
addpath('C:\Users\ivan\OneDrive\Documents\PhD\CableFDM\cableFDM\Matlab\FinalFiles\util
ities')

B = 0.74;
H = 0.79;
```

```
anchorPoints = [0 0 H; B 0 H; B B H; 0 B H];     % anchor points for 4 cables
da = [0 0 0.0375];
db = [0 0 0];
dc = [-0.07 0.01 -0.025];
dd = [0 0 0];
positions = [x1 y1 z1];
cableLengths = [la1 lb1 lc1 ld1];

% Moving avg filter
numPoints = length(t1);
states = [positions cableLengths];
states_filt = NaN(numPoints,7);
avgSamples = 100;
for i=avgSamples:numPoints
    avgSum = zeros(1,7);
    for j=1:avgSamples
        avgSum = avgSum + states(i-j+1,:);
    end
    states_filt(i,:) = avgSum/avgSamples;
end
positions = states_filt(:,1:3);
cableLengths = states_filt(:,4:7);

method = 4;
aEstimate = estimateDisturbances(t1, positions, cableLengths, anchorPoints,
[79,89,99,109], method);
bEstimate = estimateDisturbances(t1, positions, cableLengths, anchorPoints,
[109,119,129,19], method);
cEstimate = estimateDisturbances(t1, positions, cableLengths, anchorPoints,
[19,29,39,49], method);
dEstimate = estimateDisturbances(t1, positions, cableLengths, anchorPoints,
[49,59,69,79], method);

trueAnchorPoints = anchorPoints+[da;db;dc;dd];
stem3(trueAnchorPoints(:,1),trueAnchorPoints(:,2),trueAnchorPoints(:,3),'r')
estimatedAnchorPoints =
anchorPoints+[aEstimate(1,:);bEstimate(2,:);cEstimate(3,:);dEstimate(4,:)];
stem3(estimatedAnchorPoints(:,1),estimatedAnchorPoints(:,2),estimatedAnchorPoints(:,3)
,'.r')

% Plot estimated and actual anchor point positions
figure()
subplot(1,4,1)
plot(1:3,da,'or',1:3,aEstimate(1,:),'.k')
xlim([0 4]);ylim([-B/8 B/8]);
ylabel('Diplacement amount [m]')
xticks([1 2 3]);xticklabels({'X','Y','Z'});grid on;
title('a')
subplot(1,4,2)
plot(1:3,db,'or',1:3,bEstimate(2,:),'.k')
xlim([0 4]);ylim([-B/8 B/8]);
xticks([1 2 3]);xticklabels({'X','Y','Z'});grid on;
title('b')
subplot(1,4,3)
plot(1:3,dc,'or',1:3,cEstimate(3,:),'.k')
xlim([0 4]);ylim([-B/8 B/8]);
xticks([1 2 3]);xticklabels({'X','Y','Z'});grid on;
title('c')
subplot(1,4,4)
plot(1:3,dd,'or',1:3,dEstimate(4,:),'.k')
xlim([0 4]);ylim([-B/8 B/8]);
xticks([1 2 3]);xticklabels({'X','Y','Z'});grid on;
title('d')
legend({'Actual','Estimate'},'Location','South')
```

Table A33. *pointsPath.m* is a MATLAB function that is used to create a reference trajectory for a set of given waypoints. This function is used by the *plotData.m* script shown in Table A32. A nearly identical function was used in the analysis code.

```
function [xr,yr,zr] = pointsPath(xCoordinates, yCoordinates, zCoordinates, timeStamps,
currentTime)
% Used to create a cartesian path from waypoints and time stamps
%
% xCoordinates is a list of x positions in meters
% yCoordinates is a list of y positions in meters
% zCoordinates is a list of z positions in meters
% timeStamps is a list of times in seconds
% currentTime is the current time in seconds
%
% Function returns the x,y,z position corresponding to the current time.
% Lists of cooridnates and time stamps should be the same length.
% Optional: instead of time stamps, give the movement speed in [m/s]

    % Get the number of times to sample
    numTimeSamples = length(currentTime);
    % Get the number of points in the path
    numPoints = length(xCoordinates);
    % Calculate the number of segments in the path
    numSegments = numPoints-1;

    % If time stamps not given, calculate them using a constant movement speed
    if(length(timeStamps)==1 && numPoints>1)
        speed = timeStamps; % Movement speed in [m/s]
        timeStamps = NaN(numPoints,1);  % Create array of time stamps
        timeStamps(1) = 0;  % First time stamp is zero seconds
        for(i=1:numSegments)
            % Changes in coordinates for current segment
            dx = xCoordinates(i+1)-xCoordinates(i);
            dy = yCoordinates(i+1)-yCoordinates(i);
            dz = zCoordinates(i+1)-zCoordinates(i);
            distance = norm([dx,dy,dz],2);
            timeStamps(i+1) = timeStamps(i)+distance/speed;
        end
    end

    % Initialize position arrays to be returned
    xr = NaN(numTimeSamples,1);
    yr = NaN(numTimeSamples,1);
    zr = NaN(numTimeSamples,1);

    % Iterate through time samples
    for(i=1:numTimeSamples)
        % Calculate the current segment number
        segment = find(currentTime(i)>=timeStamps,1,'last');

        if(segment==numPoints)
            % If the current time is beyond the final timestamp, stay at final
position
            x = xCoordinates(end);
            y = yCoordinates(end);
            z = zCoordinates(end);
        else
            % Calculate the current segment start and end times
            segmentStartTime = timeStamps(segment);
            segmentEndTime = timeStamps(segment+1);

            % Calculate current segment progress
            segmentProgress = (currentTime(i)-segmentStartTime)/(segmentEndTime-
segmentStartTime);
```

```
            % Calculate start and end coordinates of current segment
            xStart = xCoordinates(segment);
            yStart = yCoordinates(segment);
            zStart = zCoordinates(segment);
            xEnd = xCoordinates(segment+1);
            yEnd = yCoordinates(segment+1);
            zEnd = zCoordinates(segment+1);

            % Calculate current position
            x = xStart + segmentProgress*(xEnd-xStart);
            y = yStart + segmentProgress*(yEnd-yStart);
            z = zStart + segmentProgress*(zEnd-zStart);
        end

        % Return reference position
        xr(i) = x;
        yr(i) = y;
        zr(i) = z;
    end
end
```

Table A34. *Kinect_massTracker.py* is a Python program that operates the Kinect v2 sensor and processes the images in real time to extract the 3-D position of the CSR mass.

```
# Script that uses pykinect and Kinect v2 camera to collect
# video and depth images. OpenCV is used to process the images.

# This version finds a polygon target and creates a focused window that tracks the target
# in the image. Then, further image processing uses only the focused window. The x, y,
and z
# data of the target is locally stored and, optionally, sent to an Arduino via serial
communication.
# Some options provided below:
#   -Display live images as they are captured and processed
#   -Plot 3D graph of the position data
#   -Communicate with Arduino. Requires Arduino to be running 'cableRobotController.ino'
#    -Debug Arduino serial communication by displaying sent data and then echo of the
same data.
#    -Select which data is saved (position only, or full data with cable lengths from
Arduino).
#   -Select whether the color image is used for position calculation. If not, the depth
image and
#    kinect libraries are used to calculate the x,y,z position in mm.


# Kinect v2 library
from pykinect2 import PyKinectV2
from pykinect2 import PyKinectRuntime
# Matrix manipulation library
import numpy as np
# Math functions
import math
# Plotting library
import PyGnuplot as gp
# Serial communication library
import serial
# Python strings and C struct conversion library
import struct
# Computer vision library
import cv2
# Time library
import time
# Operating and system libraries
```

```
import os
import sys


# -----------------------------------------------
# --------- Kinect and openCV settings -----------
# -----------------------------------------------

# set resolutions for color and depth images [x,y]
colorResolution = [1080, 1920]
depthResolution = [424, 512]
#colorFrame = np.zeros((colorResolution[0]*colorResolution[1]*4))
#depthFrame = np.zeros((depthResolution[0]*depthResolution[1]))

# default color image cropping (before any cv processing)
cropX0 = 560
cropX1 = 1340
cropY0 = 180
cropY1 = 960

# default depth image cropping (before any cv processing)
dcropX0 = 140
dcropX1 = 405
dcropY0 = 70
dcropY1 = 340

# focus crop for color image (only the focused area will be cv processed)
# these variables are global because they are updated within the color image handler
function
global focusX0
global focusX1
global focusY0
global focusY1
# starting focus area is the entire default cropped color image
focusX0 = 0
focusX1 = cropX1-cropX0
focusY0 = 0
focusY1 = cropY1-cropY0

# focus crop for depth image (only the focused area will be cv processed)
# these variables are global because they are updated within the depth image handler
function
global dfocusX0
global dfocusX1
global dfocusY0
global dfocusY1
# starting focus area is the entire default cropped depth image
dfocusX0 = 0
dfocusX1 = dcropX1-dcropX0
dfocusY0 = 0
dfocusY1 = dcropY1-dcropY0

# position variables for the target position
# these variables are global because they are updated within the image handler functions
global xPixel
global yPixel
global zDepthVal
global plotX
global plotY
global plotZ
plotX = 0
plotY = 0
plotZ = 0
xPixel = 0
yPixel = 0
zDepthVal = 0
```

```
# script options (setting these to True generally slow down the script)
useColorImage = True          # use color image for x,y position. If false, use depth
image with Kinect library conversions
displayImages = True                # display images captured by camera
plotData = False                            # plot the target data live
arduinoComm = False                     # enable Arduino serial communication
saveArduinoData = False             #  save data provided by Arduino. If false, just
save local x,y,z data
commDebug = False                       # print diagnostic data to debug Arduino serial
communication


# ------------------------------------------------
# ------ Arduino serial communication setup ------
# ------------------------------------------------

# Make sure Arduino communication is on if Arduino data is to be saved
if saveArduinoData == True or commDebug == True:
       arduinoComm = True

# Setup Arduino serial if Arduino communication is requested
if arduinoComm == True:
       # setup serial using com number and baudrate
       arduino = serial.Serial('COM3', 2000000)

       # small time wait for serial setup
       time.sleep(1)


# Function for sending data to Arduino
def sendToArduino(xData, yData, zData):
       # Pack data into bytes (2 bytes per number, little-endian first)
       dataBytes = struct.pack('<hhh', xData, yData, zData)

       # Write data to Arduino (6 bytes)
       arduino.write(dataBytes)

       # Return data from Arduino to debug communication
       if commDebug == True:
              print('Sent to Arduino: ', xData, yData, zData)
              # Get data bytes from Arduino
              xbyte = arduino.read(2)
              ybyte = arduino.read(2)
              zbyte = arduino.read(2)
              # Clear any bytes left in input buffer
              arduino.reset_input_buffer()
              # Convert bytes to short integer numbers
              xdata = struct.unpack('h', xbyte)
              ydata = struct.unpack('h', ybyte)
              zdata = struct.unpack('h', zbyte)
              print('Arduino returned: ', xdata[0], ydata[0], zdata[0])
              print(' ')

       # If saving data from Arduino, retreive it
       if saveArduinoData == True:
              #arduinoDataLine = arduino.readline()
              #dataLines.append(arduinoDataLine)

              # Get data bytes from Arduino
              d1 = arduino.read(4)
              d2 = arduino.read(4)
              d3 = arduino.read(4)
              d4 = arduino.read(4)
              d5 = arduino.read(4)
              d6 = arduino.read(4)
              d7 = arduino.read(4)
              d8 = arduino.read(4)
```

```
                # Clear any bytes left in input buffer
                arduino.reset_input_buffer()
                # Convert bytes to float numbers
                d1f = struct.unpack('f', d1)[0]
                d2f = struct.unpack('f', d2)[0]
                d3f = struct.unpack('f', d3)[0]
                d4f = struct.unpack('f', d4)[0]
                d5f = struct.unpack('f', d5)[0]
                d6f = struct.unpack('f', d6)[0]
                d7f = struct.unpack('f', d7)[0]
                d8f = struct.unpack('f', d8)[0]

                # Sava data for writing to file
                dataLines.append("%f    %f    %f    %f    %f    %f    %f    %f    \n"    %
(d1f,d2f,d3f,d4f,d5f,d6f,d7f,d8f))


# -------------------------------------------------
# ---------- Some other helper functions ---------
# -------------------------------------------------

# Function that prints x y data when mouse clicks on opencv image
def on_click(event,x,y,p1,p2):
        if event == cv2.EVENT_LBUTTONDOWN:
                print(x,y)

# Mapping from Kinect V2 library
def depthToCameraSpace(kinectObject, xP, yP, zD):
        d = PyKinectV2._DepthSpacePoint()
        d.x = xP
        d.y = yP
        # Use depth image data to calculate x,y,z position in [m]
        return kinectObject._mapper.MapDepthPointToCameraSpace(d, zD)

# maping from color and depth camera (from measurements)
def mapToMeters(xP,yP,zD):
        # Calibration data
        # xPixels = np.array([151, 378, 600])
        # xMeters = np.array([0.195, 0.395, 0.595])
        # yPixels = np.array([163, 394, 613])
        # yMeters = np.array([0.195, 0.395, 0.595])
        # zDepthVals = np.array([923, 960, 998])
        # zMeters = np.array([0.145, 0.10, 0.05])
        xPixels = np.array([172, 395, 618])
        xMeters = np.array([0.195, 0.395, 0.595])
        yPixels = np.array([50, 148, 366, 589])
        yMeters = np.array([0.095, 0.195, 0.395, 0.595])
        zDepthVals = np.array([923, 960, 998])
        zMeters = np.array([0.145, 0.10, 0.05])

        # Linear interpolation, find lower index
        ix = np.where(xPixels<=xP)[0]
        iy = np.where(yPixels<=yP)[0]
        iz = np.where(zDepthVals<=zD)[0]

        # x interpolation
        if ix.size == 0:
                xMeter     =     xMeters[0]     +     (float(xP-xPixels[0])/(xPixels[1]-
xPixels[0]))*(xMeters[1]-xMeters[0])
        elif ix.size == xPixels.size:
                xMeter  =  xMeters[-1]  +  (float(xP-xPixels[-1])/(xPixels[-1]-xPixels[-
2]))*(xMeters[-1]-xMeters[-2])
        else:
                xMeter  =  xMeters[ix[-1]]  +  (float(xP-xPixels[ix[-1]])/(xPixels[ix[-
1]+1]-xPixels[ix[-1]]))*(xMeters[ix[-1]+1]-xMeters[ix[-1]])
        # y interpolation
        if iy.size == 0:
                yMeter     =     yMeters[0]     +     (float(yP-yPixels[0])/(yPixels[1]-
yPixels[0]))*(yMeters[1]-yMeters[0])
        elif iy.size == yPixels.size:
                yMeter  =  yMeters[-1]  +  (float(yP-yPixels[-1])/(yPixels[-1]-yPixels[-
2]))*(yMeters[-1]-yMeters[-2])
        else:
                yMeter  =  yMeters[iy[-1]]  +  (float(yP-yPixels[iy[-1]])/(yPixels[iy[-
1]+1]-yPixels[iy[-1]]))*(yMeters[iy[-1]+1]-yMeters[iy[-1]])
        # z interpolation
        if iz.size == 0:
                zMeter   =   zMeters[0]   +   (float(zD-zDepthVals[0])/(zDepthVals[1]-
zDepthVals[0]))*(zMeters[1]-zMeters[0])
        elif iz.size == zDepthVals.size:
                zMeter  =  zMeters[-1]  +  (float(zD-zDepthVals[-1])/(zDepthVals[-1]-
zDepthVals[-2]))*(zMeters[-1]-zMeters[-2])
        else:
                zMeter     =     zMeters[iz[-1]]     +     (float(zD-zDepthVals[iz[-
1]])/(zDepthVals[iz[-1]+1]-zDepthVals[iz[-1]]))*(zMeters[iz[-1]+1]-zMeters[iz[-1]])

        return [xMeter,yMeter,zMeter]

# ---------------------------------------------------
# -------- Kinect frame handler functions --------
# ---------------------------------------------------

# color image handler function
def processColorFrame(frame):
        # access the global variables for the focus window location
        global xPixel
        global yPixel
        global zDepthVal
        global plotX
        global plotY
        global plotZ
        global focusX0
        global focusX1
        global focusY0
        global focusY1

        # reshape the color frame data
        colorImageData = np.reshape(colorFrame,(colorResolution[0], colorResolution[1],-
1)).astype(np.uint8)
        # flip the image horizontally (the image is mirrored by default)
        colorImage = cv2.flip(colorImageData, 1)
        # crop image to default area of interest
        croppedColorImage = np.copy(colorImage[cropY0:cropY1,cropX0:cropX1,:])
        # crop image again to focus on target
        focusedColorImage = np.copy(croppedColorImage[focusY0:focusY1, focusX0:focusX1,
:])
        # filter focused color image based on HSV values
        filteredColorImage = hsvFilter(focusedColorImage)
        # find target in focused image
        targetFound, approx, x, y = findTarget(filteredColorImage)
        # if a target was found
        if targetFound == True:
                # calculate the x, y coordinates in the original color image
                xPixel = (focusX0 + x)
                yPixel = ((cropY1-cropY0) - (focusY0 + y))

                # Convert data to units of [m] (uses color image pixel coordinates and
depth image z depth value)
                mappedValues = mapToMeters(xPixel,yPixel,zDepthVal)
                # Store result for plotting
                #       The 2.5cm offset accounts for eyehook displacement from frame
                plotX = mappedValues[0]-0.025
                plotY = mappedValues[1]-0.025
                plotZ = mappedValues[2]
                # if display of images is requested
                if displayImages == True:
                        # calculate the pixel location of the target in the original color
image coordinates
                        shiftedX = x + cropX0 + focusX0
                        shiftedY = y + cropY0 + focusY0
                        # calculate the pixel location of the polygon approximation in
the original color image coordinates
                        shiftedApprox= approx + [cropX0 + focusX0, cropY0 + focusY0]
                        # show the target location on the original color image
                        cv2.circle(colorImage, (shiftedX, shiftedY), 2, (0,255,0), -1)
                        # show the target polygon on the original color image
                        cv2.drawContours(colorImage, [shiftedApprox], -1, (0,255,0), 2)

                # update focus window based on current location of target
                # the next focus window is centered at the target location and is 200 by
200 px
                focusX0 = focusX0 + x - 100
                focusX0 = max(0, focusX0)
                focusX1 = focusX0 + x + 100
                focusX1 = min(cropX1-cropX0, focusX1)
                focusY0 = focusY0 + y - 100
                focusY0 = max(0, focusY0)
                focusY1 = focusY0 + y + 100
                focusY1 = min(cropY1-cropY0, focusY1)

        # if no target found in color image
        # reset focus image bounds to default values
        else:
                focusX0 = 0
                focusX1 = cropX1-cropX0
                focusY0 = 0
                focusY1 = cropY1-cropY0

        # if display of images is requested
        if displayImages == True:
                # show default cropped area in blue
                cv2.rectangle(colorImage,  (cropX0,cropY0),  (cropX1,cropY1),  (0,0,255),
2)
                # show updated focused area in red
                cv2.rectangle(colorImage,       (cropX0+focusX0,       cropY0+focusY0),
(cropX0+focusX1, cropY0+focusY1), (255,0,0), 2)
                # show the color image in an opencv window
                cv2.imshow('KINECT Color Stream', colorImage)
                # show the focused, filtered color image in another opencv window
                cv2.imshow('Color Vision', filteredColorImage)


# depth image handler function
def processDepthFrame(frame):
        # access the global variables for the focus window location
        global xPixel
        global yPixel
        global zDepthVal
        global plotX
        global plotY
        global plotZ
        global dfocusX0
        global dfocusX1
        global dfocusY0
        global dfocusY1
        # reshape the depth frame data
        depthImageData = np.reshape(frame,(depthResolution[0], depthResolution[1],-1))
        # flip image horizontally (the image is mirrored by default)
        depthImage = cv2.flip(depthImageData, 1)
        # crop image to default area of interest
        croppedDepthImage = np.copy(depthImage[dcropY0:dcropY1,dcropX0:dcropX1])
        # crop image again to focus on target
        focusedDepthImage        =        np.copy(croppedDepthImage[dfocusY0:dfocusY1,
dfocusX0:dfocusX1])
        # filter cropped image based on depth values
        filteredDepthImage = depthFilter(focusedDepthImage)
        # convert depth image to BGR (for display purposes only)
        depthImage = cv2.cvtColor(depthImage, cv2.COLOR_GRAY2BGR).astype(np.uint8)
        # find largest target in the cropped image
        found, approx, x, y = findTarget(filteredDepthImage)
        # if a target was found, do the following
        if found == True:
                # get the depth value from the cropped depth frame, using the found-
target pixel location
                # create matrix of zeros, same size as focused depth image
                mask = np.zeros(focusedDepthImage.shape)
                # place the filled polygon (approximation of the target) in the empty
mask matrix
                cv2.fillConvexPoly(mask, approx, (1))
                # use the mask with polygon to isolate the target pixels in the depth
image
                maskedImage = np.multiply(focusedDepthImage, mask)
                # the depth value is the average depth value in the isolated pixels
                depthVal = int(round(np.average(maskedImage[maskedImage!=0])))
                # calculate the x, y coordinates in the original depth image
                xPixel = np.float32(x + dcropX0 + dfocusX0)
                yPixel = np.float32(y + dcropY0 + dfocusY0)
                zDepthVal = depthVal

                # if color camera is not being used, calculate 3D position using built-
in kinect function
                if useColorImage == False:
                        # convert data to units of [m] (uses depth image pixel coordinates
and z depth value)
                        cameraSpacePosition                                            =
depthToCameraSpace(kinect,xPixel,yPixel,zDepthVal)
                        # store results for plotting (add offsets to account for camera
location)
                        #       The 2.5cm offset accounts for eyehook displacement from
frame
                        plotX = 0.432 + cameraSpacePosition.x - 0.025
                        plotY = 0.398 + cameraSpacePosition.y - 0.025
                        plotZ = 1.065 - cameraSpacePosition.z

                # if display of image is requested
                if displayImages == True:
                        # calculate the pixel location of the target in the original depth
image coordinates
                        shiftedX = x + dcropX0 + dfocusX0
                        shiftedY = y + dcropY0 + dfocusY0
                        # calculate the approximate polygon coordinates in the original
depth image coordinates
                        shiftedApprox = approx + [dcropX0 + dfocusX0, dcropY0 + dfocusY0]
                        # show the target location in the original depth image
                        cv2.circle(depthImage, (shiftedX, shiftedY), 2, (0,255,0), -1)
                        # show the target polygon in the original depth image
                        cv2.drawContours(depthImage, [shiftedApprox], -1, (0,255,0), 1)

                # update focus window based on current location of target
                # the next focus window is centered at the target location and is 80 by
80 px
                dfocusX0 = dfocusX0 + x - 40
                dfocusX0 = max(0, dfocusX0)
                dfocusX1 = dfocusX0 + x + 40
                dfocusX1 = min(dcropX1-dcropX0, dfocusX1)
                dfocusY0 = dfocusY0 + y - 40
                dfocusY0 = max(0, dfocusY0)
                dfocusY1 = dfocusY0 + y + 40
                dfocusY1 = min(dcropY1-dcropY0, dfocusY1)

        # if no target found in depth image
        # reset focus image bounds to default values (focus window becomes entire cropped
depth image)
        else:
                dfocusX0 = 0
                dfocusX1 = dcropX1-dcropX0
                dfocusY0 = 0
                dfocusY1 = dcropY1-dcropY0

        # if display of image is requested
        if displayImages == True:
                # show square of default cropped depth area in blue
                cv2.rectangle(depthImage,     (dcropX0,dcropY0),     (dcropX1,dcropY1),
(0,0,255), 1)
                # show updated focused crop area
                cv2.rectangle(depthImage,     (dcropX0+dfocusX0,     dcropY0+dfocusY0),
(dcropX0+dfocusX1, dcropY0+dfocusY1), (255,0,0), 1)
                # show the image in an opencv window
                cv2.imshow('KINECT Depth Stream', depthImage)
                # show the focused, filtered depth image in another opencv window
                cv2.imshow('Depth Vision', filteredDepthImage)


# HSV filtering
def hsvFilter(bgraImage):
        # HSV filtering threshold values
        # orange tape target
        hmin = 0
        hmax = 50
        smin = 0
        smax = 100
        vmin = 200
        vmax = 255

        # Convert BGRA (blue-green-red-alpha) to HSV (hue-saturation-value)
        hsvImage = cv2.cvtColor(bgraImage, cv2.COLOR_BGR2HSV)

        # Filter image using HSV threshold values
        # Pixels that DO NOT fall within the user-selected HSV value range are set to
value of 0
        # Pixels that DO fall within the user-selected HSV value range are set to value
of 1
        h,s,v = cv2.split(hsvImage) # Splits the HSV matrix into separate H, S, V matrices
        hf = cv2.inRange(h,np.array(hmin),np.array(hmax))    # Filters H matrix
        sf = cv2.inRange(s,np.array(smin),np.array(smax))    # Filters S matrix
        vf = cv2.inRange(v,np.array(vmin),np.array(vmax))    # Filters V matrix
        filteredImage = cv2.bitwise_and(hf, cv2.bitwise_and(sf,vf)) # Creates matrix of
1's and 0's according to filtering

        return filteredImage


# Depth filtering
def depthFilter(depthImage):
        # Depth threshold values
        depthMin = 800
        depthMax = 1000

        # Filter depth image using depth threshold values
        # Pixels that DO NOT fall within the user-selected depth value range are set to
value of 0
        # Pixels that DO fall within the user-selected depth value range are set to value
of 1
        filteredImage = cv2.inRange(depthImage, np.array(depthMin), np.array(depthMax))
        return filteredImage


# Largest target detection
def findTarget(filteredImage):
        # Find all contours in the filtered image
        contours,    hierarchy    =    cv2.findContours(filteredImage,    cv2.RETR_TREE,
cv2.CHAIN_APPROX_NONE)
        # Count the number of contours found
        numContours = len(contours)
        # If there is at least one found contour, do the following
        if numContours>0:
                # Sort the contours from largest to smallest
                contours = sorted(contours, key=cv2.contourArea, reverse=True)
                # Loop over the sorted contours
                for c in contours:
                        # End loop if contour is too small (smaller than 10 px in area)
                        if cv2.contourArea(c) < 10:
                                break
                        # Approximate the contour with an approximate polygon shape
                        perimeter = cv2.arcLength(c, True)
                        approx = cv2.approxPolyDP(c, 0.01*perimeter, True)
                        # If the polygon has at least 4 sides
                        if len(approx) >= 4:
                                #  Find  center  of  the  polygon  (by  pixel  number  in  the
filtered image)
                                # Calculate the moments of the polygon
                                mom = cv2.moments(approx)
                                # If the area of the polygon is not zero
                                if mom['m00'] != 0:
                                        x = int(mom['m10']/mom['m00'])
                                        y = int(mom['m01']/mom['m00'])
                                        return (True, approx, x, y)

        # If no target was found, return the following
        return (False, 0, 0, 0)


# ------------------------------------------------
# ------- Kinect intialization and runtime -------
# ------------------------------------------------

# Create a Kinect runtime object
if useColorImage == True:
        # Create object with color and depth capture
        kinect  =  PyKinectRuntime.PyKinectRuntime(PyKinectV2.FrameSourceTypes_Color  |
PyKinectV2.FrameSourceTypes_Depth)
else:
        # Create object with depth capture only
        kinect = PyKinectRuntime.PyKinectRuntime(PyKinectV2.FrameSourceTypes_Depth)

# Allow time for Kinect to start
time.sleep(3)

# If image display was requested, create display windows
if displayImages == True:
        cv2.namedWindow('KINECT Depth Stream', cv2.WINDOW_NORMAL)
        cv2.resizeWindow('KINECT Depth Stream',  depthResolution[1], depthResolution[0])
        cv2.namedWindow('Depth Vision', cv2.WINDOW_AUTOSIZE)
        cv2.setMouseCallback('KINECT Depth Stream', on_click)
        if useColorImage == True:
                cv2.namedWindow('KINECT Color Stream', cv2.WINDOW_NORMAL)
                cv2.resizeWindow('KINECT    Color    Stream',       colorResolution[1]/2,
colorResolution[0]/2)
                cv2.namedWindow('Color Vision', cv2.WINDOW_AUTOSIZE)
                cv2.setMouseCallback('KINECT Color Stream', on_click)
```

```
# Variable used for setting up plot figures (if requested)
firstPoint = True

# Create list of data to write to trajectory.dat file
dataLines = []

# Main loop, runs until break condition is met
while True:
        # Wait for depth frame to be available
        if kinect.has_new_depth_frame():
                # Store depth frame data
                depthFrame = kinect.get_last_depth_frame()
                # If depth data is not empty
                if depthFrame.size != 0:
                        # Process the depth frame (finds the target pixel location and
depth value)
                        processDepthFrame(depthFrame)

        # If color image is being used for tracking
        if useColorImage == True:
                # Wait for color frame to be available
                if kinect.has_new_color_frame():
                        # Store color frame data
                        colorFrame = np.copy(kinect.get_last_color_frame())
                        # If color data is not empty
                        if colorFrame.size != 0:
                                # Process the color frame (finds the target pixel location)
                                processColorFrame(colorFrame)

        # If data plotting was requested, update plot figure
        if plotData == True:
                # Send plotting data to gnu plotter
                gp.s([[plotX],[plotY],[plotZ]])

                # Setup gnu plot axes and labels when first point is plotted
                if firstPoint == True:
                        gp.c('set xrange [0:1]')
                        gp.c('set yrange [0:1]')
                        gp.c('set zrange [0:1]')
                        gp.c('set grid xtics ytics ztics')
                        gp.c('show grid')
                        gp.c('set xlabel "x [m]"')
                        gp.c('set ylabel "y [m]"')
                        gp.c('set zlabel "z [m]"')
                        gp.c('show xlabel')
                        gp.c('show ylabel')
                        gp.c('show zlabel')
                        gp.c('splot "tmp.dat" with points pointtype 7')
                        firstPoint = False
                # Update plot with any points after the first point
                else:
                        gp.c('replot')

        # If Arduino data is not being saved, save local data instead
        if saveArduinoData == False:
                # Add local data to the data line (to be written in trajectory.dat file)
                dataLines.append("%f %f %f %f \n" % (time.clock(),plotX, plotY, plotZ))

        # If Arduino communication was requested
        if arduinoComm == True:
                # Wait for Arduino to request the data (should not take long)
                while arduino.in_waiting == 0:
                        pass
                # Clear any bytes in input buffer (clear the latest Arduino request for
data)
                arduino.reset_input_buffer()
```

```
			# Send data to Arduino as 6 bytes (2 bytes per coordinate)
			# The decimal data is first converted to integers by multiplying by 10000
			sendToArduino(int(plotX*10000), int(plotY*10000), int(plotZ*10000))

		# Get value of pressed key (an openCV window must be active)
		key = cv2.waitKey(1)
		# If key pressed is 27 ('ESC') or time reaches ___s, end the program
		if key == 27 or time.clock()>140:
			# Close kinect object
			kinect.close()

			# Close all opencv windows
			cv2.destroyAllWindows()

			# Write all the data to a .dat file and close the file
			f = open(os.path.join(sys.path[0], "trajectory.dat"), "w")
			for i in range(len(dataLines)):
				f.write(dataLines[i])
			f.close()

			# Delete the temporary file used for gnu plotting
			if plotData == True:
				os.remove("tmp.dat")

			# Exit while loop
			break
```

APPENDIX C:
MAXIMAL CLIQUE ALGORITHM

Section 3.1.2 contains the kinematics of a general CSR system. In the forward kinematics, the goal was to find the position of the end-effector given the cable lengths. This task is complicated by the possibility of redundant cables and cable slack. In the most general case, it is also possible that some of the cable lengths are too short to connect with the other cables. To solve this problem, a simple algorithm was proposed wherein the first step was to determine the largest group of cables that can reach each other’s ends, assuming that the cables are in tension. This step was posed as a maximal clique problem. Many algorithms were previously studied to solve the maximal clique problem [58]. The details of these methods are outside the scope of this dissertation. However, a unique algorithm for finding a maximal clique was implemented in MATLAB for the purpose of calculating the forward kinematics of a general CSR. The relevant codes were included in Appendix A and are contained in the folder “FindLargestSquare” that is shown in Figure 127. The maximal clique algorithm developed for this dissertation is now described.

First, let the goal simply be described as: find the largest (or one of the largest) groups of cables such that any of the two cables in the group can connect with each other. Then, note that it is easy to determine whether two cables can connect with each other by checking that the sum of the two cable lengths is greater than the Euclidean distance between the two cable anchor points. Once this check is performed for all possible pairs of cables in the system, this information can be expressed in a matrix form. Take, for example, the matrix

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}.$$

In this matrix, each row (or column) represents the possible connections for a single cable. There is one row (or column) for each cable in the given CSR system. The first row shows that cable 1 can connect with cables 2 and 4, but not with cable 3. The zero in the (1, 3) location gives that indication. Cable 2 can connect with all of the other cables. Hence, row 2 is filled with ones. Cable 3 can only connect with cable 2, and cable 4 can only connect with cables 1 and 2. The diagonal of the matrix is always filled with ones, for a reason that will later be explained. This matrix is like an adjacency matrix, but with the diagonal values being ones instead of zeros. The adjacency matrix with zeros in the diagonal is often used in graph theory. From this point forward, the matrix used in this dissertation will be referred to as the cable-connection matrix (CCM).

The order of the cables can be rearranged to produce a different, but equivalent, CCM. Here, “equivalent” means that the matrix gives the same information about the system even if the order of the cables is changed. To rearrange the order of the cables, one can simply swap the appropriate rows and columns of the matrix. The *swapMatrixOrder.m* and *reorderMatrix.m* files in Figure 127 are the MATLAB scripts that perform exactly this kind of operation for a given CCM.

By rearranging the order of the cables in the CCM, one can easily find groups of connecting cables. This is because groups of connecting cables will form diagonal blocks in the matrix when those cables are grouped in the matrix order. A diagonal block is a block of ones along the diagonal of the connection matrix. The maximal clique algorithm is an algorithm that attempts to find the largest possible diagonal block by reordering the matrix order in a CCM. To create the algorithm, the main question is how to efficiently perform this matrix-ordering task.

In this dissertation, diagonal blocks are found by working backwards. That is, rows and columns are removed from the original CCM until only a square filled with ones remains. When row *i* and column *i* of the matrix are simultaneously removed, this is like removing cable *i* from the CSR system. By removing rows and columns until a filled square remains, the algorithm is removing cables until all that remains is a group of cables that can connect with each other, meeting the stated goal.

To determine whether a solution has been found, the algorithm evaluates a cost function. Simply put, the cost function gives the number of zeros left in the matrix when certain cables are removed. The cost is defined as

$$J = \boldsymbol{x}^T(\boldsymbol{1} - \boldsymbol{Q})\boldsymbol{x}. \tag{78}$$

where $\boldsymbol{Q}$ is the original CCM and $\boldsymbol{1}$ is a matrix of ones with the same dimensions as $\boldsymbol{Q}$. The vector $\boldsymbol{x}$ indicates what cables are removed from the system. For example,

$$\boldsymbol{x} = [1 \quad 0 \quad 1 \quad 1]^T \tag{79}$$

indicates that only cable 2 is removed from the system. Then, J will be the number of zeros left in the CCM when row 2 and column 2 are removed. A solution is found, and the algorithm terminates, when J is zero, meaning that the remaining matrix is filled with ones. At that point, the elements of $\boldsymbol{x}$ with a one indicate the cables that can connect with each other. One important thing to note, however, is that there may be more than one solution, and the solution depends on the way the algorithm removes cables.

The working principle of the algorithm in this dissertation is now described. At each iteration, the algorithm removes the cable that would result in the largest decrease in the cost, J. The hope is to find the shortest path to a cost of zero as this would mean removing the fewest number of cables. However, it is not guaranteed that the shortest path will be found if only because, at some iteration, there may be more than one cable that, when removed, would result in the same decrease in cost. Therefore, there may be multiple paths to explore at different steps of the algorithm. In such cases, the algorithm selects the cable with the lowest index number in the $\boldsymbol{x}$ vector. There may be more efficient methods to find a cost of zero, but this possibility is left for future work. The complete algorithm is summarized in Table A35.

Table A35. A summary of the maximal clique algorithm used in this dissertation, posed as a matrix-ordering problem.

Given a CCM, $\boldsymbol{Q}$, that is n by n,
let $\boldsymbol{x} = [1, \ldots, 1]^T$ (*i.e.* $\boldsymbol{x}$ is a column vector of ones with length n).

Step 1. Calculate the cost using (78) and the current $\boldsymbol{x}$.
Step 2. If the cost is zero, stop; A block of ones has been found. If the cost is nonzero, go to Step 3.
Step 3. For every 1 remaining in the current $\boldsymbol{x}$, try changing that 1 to a 0 and calculating the cost again with the modified $\boldsymbol{x}$. Keep a record of the decrease in cost using the modified $\boldsymbol{x}$'s.
Step 4. Look at the record from Step 3 and see what modified $\boldsymbol{x}$ gave the largest decrease in cost. Choose that $\boldsymbol{x}$ as the new current $\boldsymbol{x}$. If more than one modified $\boldsymbol{x}$ gave the same decrease in cost, pick any one of those modified $\boldsymbol{x}$'s as the new current $\boldsymbol{x}$. Return to Step 1.

Finally, an example of the algorithm output is presented. A CCM was created pseudorandomly. This matrix is symmetric, its diagonal is filled with ones, and the rest of its elements are either one or zero. As such, this matrix represents a CSR system where some cables can connect with each other and others cannot. The goal of the algorithm is to find a large (hopefully, the largest) group of cables that can connect with each other. The result will be displayed as a diagonal block in the upper left corner of the CCM after the order of the cables has been changed. In order to visualize the matrix before and after the reordering, a MATLAB function, called *displayMatrix.m*, was used. This function displays the matrix as a grid. The one elements of the matrix are filled with a solid color, and the zero elements are left empty.

Figure 129 shows the CCM before being processed by the algorithm. Only cable 9 can connect with all other cables, as evidenced by the full 9th row. The rest of the cables

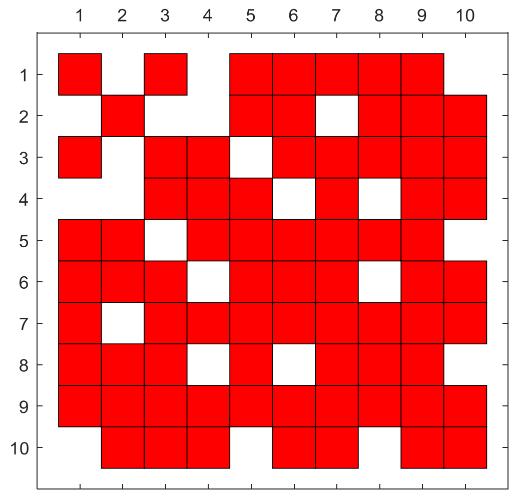


Figure 129. An example CCM, visualized as a grid, before being sorted by the maximal clique algorithm. This is a 10×10 matrix, and the red squares represent elements in the matrix that are ones. The empty spaces represent zeros.

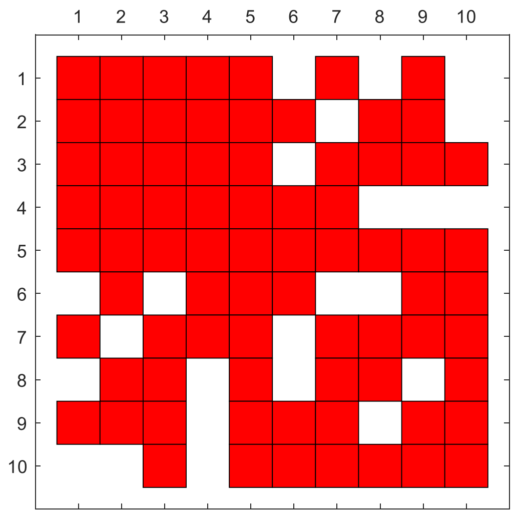


Figure 130. An example of a CCM, visualized as a grid, after being sorted by the maximal clique algorithm. A group of five cables were found to connect to each other and are grouped in the top left corner of the newly-ordered connection matrix.

have at least one cable that they cannot connect with, resulting in the empty spaces of the grid. This matrix was then processed by the MATLAB function, *findCliques.m*, with a running time of 0.015562 s, using the same laptop computer that is pictured in Figure 102. The output of the algorithm is shown in Figure 130, after the cable order has been changed. A 5×5 block now appears in the top left corner of the grid. This means that the algorithm found a group of 5 cables that could connect with each other, and grouped them so that they are now labeled one through five. With respect to the original order of the cables, the new order is: 1, 5, 7, 8, 9, 2, 3, 4, 6, 10. In other words, by changing the original order of the cables to the latter order, the resulting CCM is as shown in Figure 130. Practically speaking, it is now known that the group of five cables depicted are long enough to connect with each other, and they may be able to connect at a common location in the 3-D operating space of the CSR. This is the result that was sought.