THREE DEGREE-OF-FREEDOM PARALLEL ACTUATOR TELESCOPE MOUNT

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Mechanical Engineering

by

Garrett D. Gudgel

December 2015

COMMITTEE MEMBERSHIP

TITLE:	Three Degree-of-Freedom Parallel Actuator Telescope Mount
AUTHOR:	Garrett D. Gudgel
DATE SUBMITTED:	December 2015
COMMITTEE CHAIR:	John Ridgely, Ph.D. Professor of Mechanical Engineering
COMMITTEE MEMBER:	Russell Genet, Ph.D. Research Scholar in Residence
COMMITTEE MEMBER:	Saeed Niku, Ph.D. Professor of Mechanical Engineering

ABSTRACT

Three Degree-of-Freedom Parallel Actuator Telescope Mount

Garrett D. Gudgel

This thesis contains the design, implementation, and testing of an original, small-scaled two degree-of-freedom telescope mount and a medium-scaled three degree-of-freedom telescope mount inspired by the six degree-of-freedom Stewart-Gough platform telescope mount. The end product is intended to achieve research-standard resolution of targeted sky coverage for binary star research. The scaled prototype was carried through concept design, manufacturing, software development, and testing. The mount software development and electronic design is applicable to a full-scale mount as the drivers have been designed to be easily adapted to different actuator configurations. It is recommended that this design be implemented into a telescope in the one to two meter range for economic practicality.

ACKNOWLEDGMENTS

I cannot thank my committee enough for their continued guidance and encouragement throughout my project: Dr. Ridgely, Dr. Genet, and Dr. Niku. I am grateful for the experience and knowledge I have gained through their instruction in mechatronics and astronomy, and hope to continue working with each of them.

To my parents - Dan and Tracy Gudgel - and all of my family: thank you so much for your caring and extensive support. Without you I would not be where I am today, and I am eternally grateful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

A previous senior project study at California Polytechnic State University, San Luis Obispo designed an actuator for a hexapod telescope mount that would open a new market catering to the needs of universities with astronomy programs within the United States [11]. After the aforementioned study subsequent need was seen for a system utilizing fewer actuators to further limit the cost and lower the weight of a parallel actuator mount design. The idea of fewer actuators would make such mounts more physically practical for small universities and private research facilities that desire large mirror telescopes, also known as "light buckets." With a working, automated mount the limited sky coverage of a parallel actuator mount would be usable with the assumption that a quasi-meridian sky could be observed. With such an assumption all stars of interest would cross the telescope's field-of-view through the course of the night. The development of the Stewart-Gough six degree-of-freedom (DOF) platform proved with the Array for Microwave Background Anisotropy (AMiBA) telescope that a full six DOF system can be created for applications in research-grade telescopes. However, practical telescope mount applications for small research sites do not require six DOF nor complete sky coverage. It would be advantageous to make a lighter, reduced degree-of-freedom telescope mount system that becomes easier to transport to different observational sites and benefiting from the reduced cost of fewer actuators.

1.1 Astronomy and Telescope Background

Because the telescope mount is responsible for directing the telescope optics, it is important to understand general methods of measuring sky locations about the celestial sphere. As shown in Figure 1.1, viewing the sky from a constant view angle over the course of eighty minutes will depict stars as arced lines centered about the celestial pole. Two primary coordinate systems have been developed for celestial locating; the altitude-azimuth coordinate system shown in Figure 1.2, and the celestial coordinate system shown in Figure 1.3. Resources containing descriptions of both

Figure 1.1: *80-minute photographic exposure of the night sky [18].*

coordinate systems as well as the translation between the two systems are widely available [4] [9].

1.2 Existing Telescope Mounts

For researchers and amateurs alike, telescopes and mounts have a wide variety of designs and functions. Therefore, an astronomer's desired functions plays a major role in the telescope mount selection process.

1.2.1 *Altitude-Azimuth*

The Altitude-Azimuth Mount (altazimuth) is the most common mounting system among amateur astronomers, shown in Figure 1.4. This mount provides a simple two-axis movement capable of supporting and rotating an instrument about the zenith (azimuth) and above the horizon (altitude). Due to the celestial position rotating about the celestial pole, somewhat sophisticated software is necessary but widely available describing the path of celestial targets through the sky. For higher-end users who desire a third degree-of-motion, the instrumentation package will likely have a rotating mechanism thus allowing longer photo-exposure times.

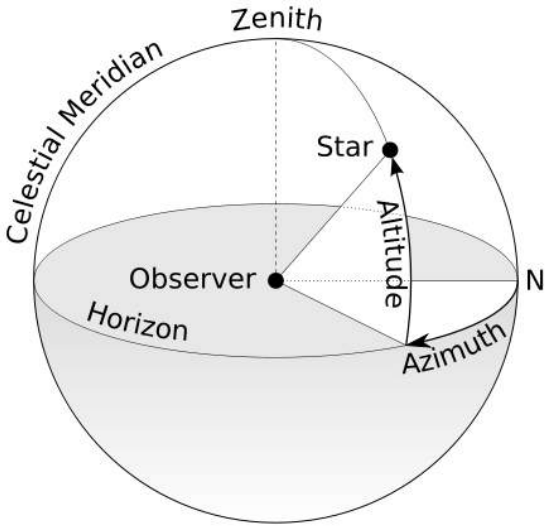


Figure 1.2: *Altitude-Azimuth coordinate system. Example depicted: Star location at* 45° *altitude and* 45° *azimuth [10].*

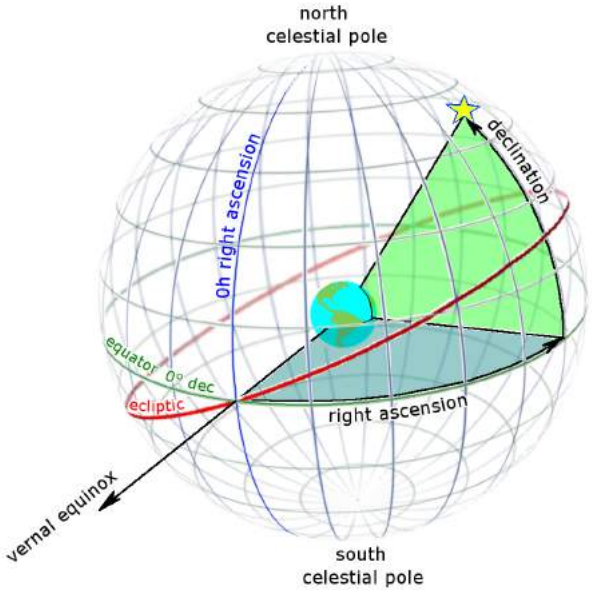


Figure 1.3: *Celestial coordinate system. Example depicted: Star location at 5h right ascension and* 50° *declination [6].*

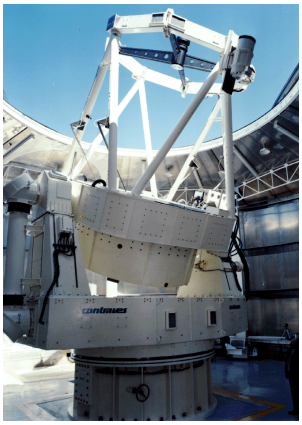


Figure 1.4: *Example of an altazimuth telescope mount [7].*

While the force of gravity is consistent throughout the range of motion of these telescopes, the concentrated load points at the altitude and sometimes azimuth bearings can be problematic and necessitate the use of larger, stronger mounts. A common fix for the load concentration about the azimuth bearing is to expand the azimuth to a trunion-and-roller-base mount to spread the rotation about a greater area. However, this fix requires full weight support over a larger base area as well as precision development ensuring no telescope droop or unintended movement along the telescope track. In addition to the concern on the base weight loading, this type of mount will not operate in a small area about the zenith due to degeneracy between the azimuth mount and the instrument de-rotator causing the rotation rate of the azimuth axis to approach infinity as the system switches from East to West.

1.2.2 *Equatorial*

More common in smaller amateur telescopes, the equatorial mount is significant as its design rotates the image automatically for users by rotating along the same pole as the earth relative to the sky, as shown in Figure 1.5. It is the telescope mount of choice for non-actuated mounts due to the simplicity of following objects through the sky. Once the declination is set the user only needs to rotate the telescope optical-tube-assembly along the right-ascension axis to keep the object in site.

Due to the geometry of the this type of mount, the gravity load is directed at many angles with respect to the bearings and therefore must be heavier compared to

Figure 1.5: *Example of an equatorial telescope mount [15].*

altitude-azimuth mounts to compensate for the increased stresses. It is uncommon for mounts of this design to be built for aperture sizes above 0.3m due to both the gravity loading and the telescope enclosure expenses. The enclosure must be significantly larger due also to the telescope optical-tube-assembly being off-axis and, therefore, having large translations especially when crossing the meridian.

1.2.3 *Altitude-Altitude*

Similar to equatorial yoke mounts, altitude-altitude mounts consist of two rotation joints gimbling the telescope about zenith, shown in Figure 1.6. With the telescope mass located at the center of the mount and oriented towards the zenith, this type of mount demonstrates advantageous control about the zenith. Of the aforementioned mounts, the alt-alt mount is arguably the easiest to assemble for amateur astronomers using stock bearings. However, this mounting system is commonly viewed as unstable due to mass location and has limited viewing capability near the horizon. Therefore the alt-alt mount is better used in applications such as missile and satellite tracking where lighter imaging equipment can be used.

1.2.4 *Stewart-Gough Platform*

A more recent and innovative mount, the Stewart-Gough platform hexapod design utilizes six extendable linear actuators acting in parallel. This configuration is unique as it grants limited but all six spacial degrees-of-freedom for the mounting platform.

Figure 1.6: *Example of an altitude-altitude telescope mount [2].*

Figure 1.7: *The AMiBa telescope in Hawaii uses a Stewart-Gough hexapod design [13].*

It provides a strong structural integrity that typically challenges all axle mounting systems. Because of the high ratio of bearing pressure to system weight, this mount is ideal for large mirror-array telescopes such as the AMiBa in Hawaii and mid-size telescopes such as the Hexapod-Telescope in Chile.

Due to the parallel design of the Stewart-Gough Platform, the systems controlling the hexapod mounts are required to perform much more complex calculations while controlling the actuators. With modern computing power, this previous drawback is no longer a significant limiting factor. A major challenge, however, is the capability of the joints where telescope systems viewing large swaths of sky require significant rotational capabilities. Interference between joints and actuator limitations prevent complete sky coverage.

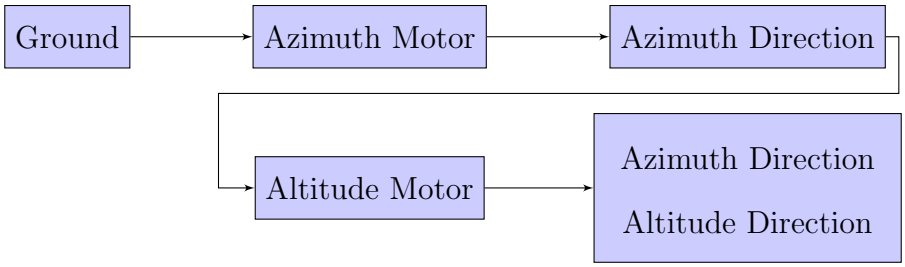


Figure 1.8: *Series manipulator design as applied to telescopes (standard).*

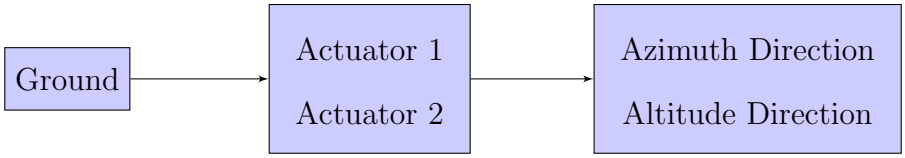


Figure 1.9: *Parallel manipulator design as applied to telescopes.*

1.3 General Background

1.3.1 *Serial vs. Parallel Actuator Design*

Like many common industrial robots most telescope mount systems currently utilize serial manipulators that are designed as a series of links connected by motor-actuated joints. In the case of telescope design this serial-actuator mount system is easily visualized in a typical altitude-azimuth telescope where two sets of gears can be motor-driven to give the telescope pan and tilt capabilities. These system of actuators are called serial manipulators because each actuator builds upon the output of the previous actuator, indicated in Figure 1.8.

In contrast, a parallel manipulator utilizes several actuators to control a single platform. These actuators or motors must synchronously move as a unit rather than independently and the combination movement of all utilized actuators defines the final orientation of the device, indicated in Figure 1.9. This design requires even more careful control of the motors to produce the desired outcome of the device.

1.3.2 *Higher Stiffness Explained*

In traditional telescopes all loads are required to pass through each actuator in series. Because of this, the entire telescope load is supported through each actuator and thus is only supported in one location along telescope frame. In consideration that there is only a single attachment point, the increased length of the telescope is therefore modeled as a cantilever beam and subject to the force of gravity. The longer the telescope the less stiff the system will be without compensating through support structure. Stiffness is also affected through the rotational actuators as the gears necessary to turn the telescope will be subjected to larger moment loads resulting in deflection. However, this affect can be mitigated by placing the encoder after the gear train directly onto the optical assembly.

Using linear actuators in parallel it is not only possible but encouraged to distribute the actuator connect points along the telescope structure. This inherently will create a more stiff structure due to shorter sections of the telescope being left to act in cantilever compared to the traditional mount.

Flexing of gears in the traditional mount is exchanged for axial deformation of the actuator lead screws in a parallel design, however the load is distributed among all actuators in a parallel design whereas the entire load passes through each actuator in the traditional design.

1.3.3 *Axial Loading*

Axial loading down an actuator rod is important as this puts the rod in a state least likely to bend or buckle. In order to assume axial loading in this state, there can be no moments applied at either loading point. This loading is attained by use of two-force members:

A two-force member has forces applied at only two points on the member. Therefore for any two-force member to be in equilibrium, the two forces acting on the member must have the same magnitude, act in opposite directions, and have the same line of action directed along the line joining the two points where these forces act [12, p.224]. Demonstrated in Figure 1.10.

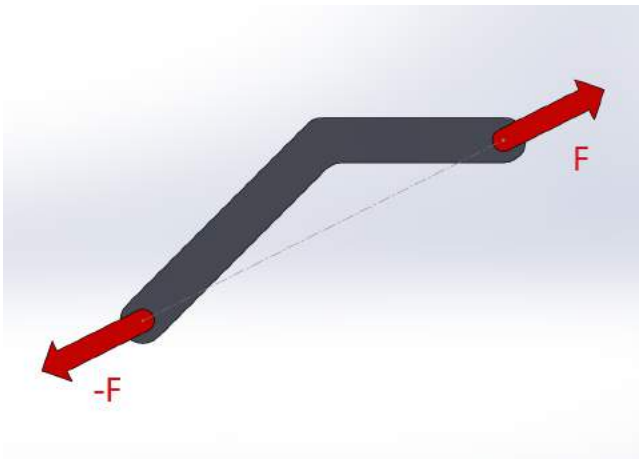


Figure 1.10: *Two-force members contain two points-of-contact comprised only of non-moment forces.*

When expanded to three-dimensional space the same principle applies. Rather than a single hinge at each point-of-contact, a second dimension of hinge needs to be applied to prevent any moment loads from acting on the rod. With no moment loads being applied at either point of contact, the two forces applied to either end of the two-force member must contain the same magnitude in the opposite direction along the same line of action. This assumes the rod's weight is negligible and no other forces are acting on the rod. Under these assumptions with the addition that failure by yielding or wear will not occur, the maximum load that an actuator rod can sustain is determined by Euler's formula for pin-ended columns as depicted by Figure 1.11 [8, p.181]:

$$P_{cr} = \frac{\pi^2 EI}{L^2} \tag{1.1}$$

P_{cr} Critical load

E Young's modulus of the rod material

I Cross-sectional area moment of inertia of the rod

L Length between points of rotation

Assuming this critical load (P_{cr}) is not reached, the deflection between points of rotation can be determined through the axial deflection equation [8, p.149]:

$$\delta = \frac{FL}{AE} \tag{1.2}$$

δ Axial deflection

F Axial load

L Length between points of rotation

A Cross-sectional area of rod

E Young's modulus of the rod material

This axial deflection is important when determining the accuracy of each actuator as deflection could alter a telescope's viewing angle.

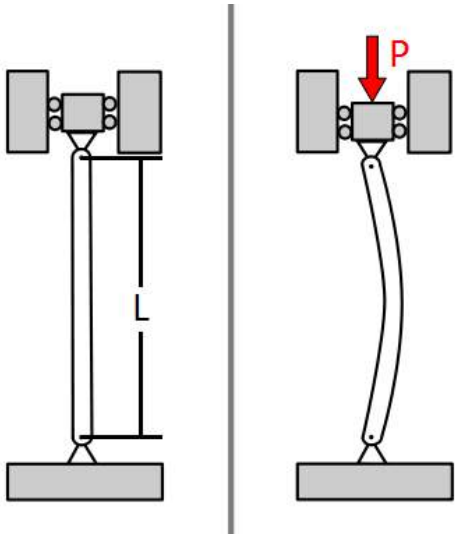
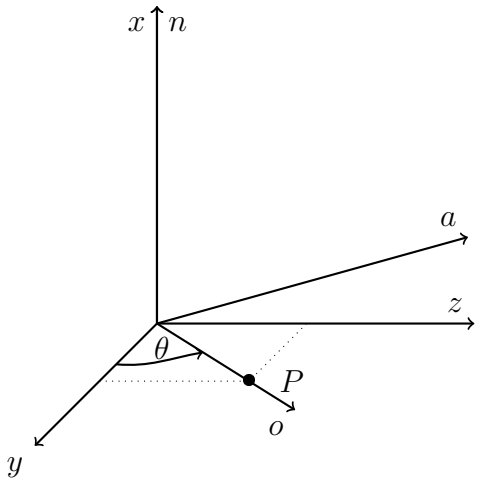


Figure 1.11: *A two-force member will buckle when the critical load is reached [3].*

1.3.4 *Reference Frame Transformation*

Physical locations in three-dimensional space can be referenced by Cartesian x,y,z coordinates to describe and specify the location. However, a frame of reference is necessary to specify the origin and it is possible for multiple frames of reference to exist. In this circumstance it is possible for a given point in space to contain multiple sets of coordinates depending on the reference frame from which the point is observed. For example, frames F_{xyz} and F_{noa} share an origin and contain point P as seen in Figure 1.12. Note that *noa* represent *xyz* axes of the relative reference frame.



$$P_{noa} = [0, \quad 1, \quad 0]$$
$$P_{xyz} = [0, \quad 0.707, \quad 0.707]$$
$$\theta = 45°$$

Figure 1.12: *Point coordinates according to different reference frames.*

A frame is represented by a 3×3 matrix whose vectors represent xyz coordinate directions of the relative axes *noa* according to the universe frame.

$$F_{noa} = \begin{bmatrix} n_x & o_x & a_x \\ n_y & o_y & a_y \\ n_z & o_z & a_z \end{bmatrix} \tag{1.3}$$

Knowing the relationship between reference frames can predict coordinates of a point in the other reference frame. This is accomplished by using what is called a transformation matrix to translate coordinates of a point according to one reference frame to the other. For the purposes of this paper all frames of reference are assumed to share an origin. Under this assumption the only transformations will be rotational as translation transformations are used for adjusting the origin and, therefore, frames and transformation matrices can be represented by 3×3 matrices and points and vectors can be represented by 3×1 arrays. When translating a vector between the universe frame and a relative frame, the correlation can be represented by

$$^{U}p = ^{U}T_R * ^{R}p \tag{1.4}$$

where ^{U}p is a point according to the universe frame, ^{R}p is a point according to the relative reference frame, and $^{U}T_R$ represents the transformation matrix from universe to relative frames.

$$T = \begin{bmatrix} n_x & o_x & a_x \\ n_y & o_y & a_y \\ n_z & o_z & a_z \end{bmatrix} \tag{1.5}$$

A transformation frame, demonstrated in Equation 1.5, consists of 3 linearly independent and homogeneous vectors depicted by variables n, o, and a with the assumption that $n \times o = a$. These vectors are often determined by using one or combining several known transformations shown in Equations 1.6, 1.7, and 1.8.

Note that $C\theta = cos\theta$ and $S\theta = sin\theta$.

$$Rot(x,\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & C\theta & -S\theta \\ 0 & S\theta & C\theta \end{bmatrix} \tag{1.6}$$

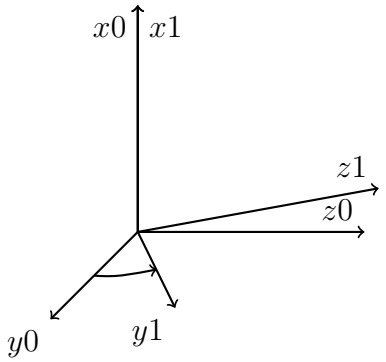


Figure 1.13: *Transformation: Rotation about x-axis [14, p.48].*

$$Rot(y,\theta) = \begin{bmatrix} C\theta & 0 & S\theta \\ 0 & 1 & 0 \\ -S\theta & 0 & C\theta \end{bmatrix} \tag{1.7}$$

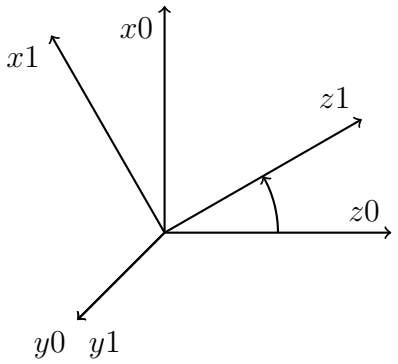


Figure 1.14: *Transformation: Rotation about y-axis [14, p.48].*

Looking at Figure 1.12 as an example, F_{noa} is found by rotating the universe frame about the x-axis where θ is 45° within Equation 1.6. The transformation can

$$Rot(z,\theta) = \begin{bmatrix} C\theta & -S\theta & 0 \\ S\theta & C\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1.8)$$

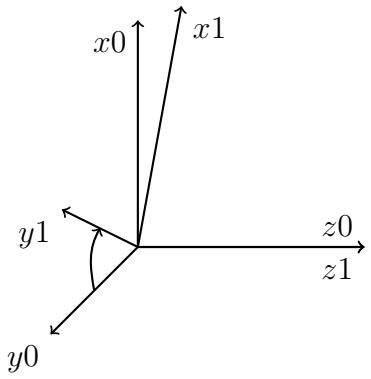


Figure 1.15: *Transformation: Rotation about z-axis [14, p.48].*

be used to find the position of points according to universe frame in the progressive application of the following equations:

$$\begin{aligned} {}^{U}T_R &= Rot(x, 45^\circ) \\ {}^{U}T_R &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.525 & -0.525 \\ 0 & 0.525 & 0.525 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} {}^{U}p &= {}^{U}T_R * {}^{R}p \\ \begin{bmatrix} 0 \\ 0.707 \\ 0.707 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.525 & -0.525 \\ 0 & 0.525 & 0.525 \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \end{aligned}$$

Order of operation is important when combining rotational transformations. Pre-multiplying signifies transformation about the original frame whereas post-multiplying signifies transformation about the current relative frame [14, p.49]. As an example, frame F is rotated 90° about the y-axis, followed by a 30° rotation about the n-axis (x-axis of the relative frame) would be represented by:

$$F = Rot(y, 90^\circ)Rot(x, 30^\circ) \quad (1.9)$$

Note Equation 1.9 can be derived by first rotating about the x-axis and proceeding to rotating about the universe y-axis, which will pre-multiply the x-transformation.

1.3.5 *Mechanism Mobility*

A mechanism is an assembly of links and joints. Joints are connections between links at points of contact called nodes. In our three-dimensional spatial frame of reference links have six inherent degrees of freedom: three axes of translation and three axes of rotation. Mobility, the degrees-of-freedom a mechanism is capable of performing, can be determined through inspection of the links and joints associated in a system by using Gruebler's equation [16, p.11]:

$$DOF_{SPATIAL} = 6(L-1) - 5J_1 - 4J_2 - 3J_3 - 2J_4 - J_5 \tag{1.10}$$

$DOF_{SPATIAL}$	DOF of full assembly
L	Total number of links in the system, including ground as a link
J_x	Number of joints in the system with x-DOF

Equation 1.10 can be better understood by recognizing the coefficients of each J_x variable represent the number of DOF that are constrained by that particular joint. Examples of joint types are shown in Table 1.1.

Table 1.1: *Gruebler's equation joint designations.*

Type	**Designation**	**Example**
Revolute	J_1	Hinge, Rotational Bearing
Prismatic	J_1	Squared Slider
Screw	J_1	Threaded Rod
Cylindrical	J_2	Telescoping Tube
Spherical	J_3	Ball Joint

The DOF's of interest for astronomy are the three DOF's of rotation. Due to the distance of celestial bodies being viewed, the view angle of these bodies from any location on earth is virtually identical and the three DOF's of translation can be ignored. Utilizing all three DOF's of rotation would allow a telescope mount to direct its OTA through its usable field-of-view and additionally rotate the telescope to account for image rotation.

CHAPTER 2: PROBLEM

Most current telescope mounts, both robotic and manual, are predominantly serial-actuator based that require significant stiffening elements on the telescope optical-tube-assembly (OTA) and frame for each additional degree-of-freedom. Practical astronomy applications for amateurs and small research sites do not require six DOF nor do these applications require complete view of the celestial dome. It would be advantageous to make a reduced degree-of-freedom system of higher stiffness and lower weight, thereby enabling transport to different sites.

When observing stars over extended periods of time from a fixed location on Earth, the stars appear to rotate about the North Celestial Pole. This rotation is detrimental to long exposure imaging. In order to prevent star tracks in the image, it is desired to maintain the camera sensor's orientation with the sky. Equatorial telescopes inherently account for this by rotating the OTA about the Earth's axis-of-rotation but are not economically feasible with large scale telescope mount designs. Altitude-azimuth telescope mounts - encompassing all recently implemented large-scale telescopes - counter this undesirable rotation by means of instrument derotators. Variations of derotators take imagery equipment at the end of the optical path and rotate it to maintain orientation with the sky. Using instrument derotators rather than rotating the entire OTA eliminates some astronomical observing capabilities. For example, masks used in adjusting diffraction patterns generated by the optics are often placed over the primary mirror. If the whole OTA can be derotated it eliminates the need to derotate individual optical instrumentation.

Load paths are significant in determining the stiffness of existing telescopes. Some telescopes such as the current $8.2m$ Very Large Telescope (VLT) in Chile weighs approximately 430 tons with the majority of weight being allocated to support structure [1]. While some weight is due to the sheer mass of the mirrors and instrumentation, the serial actuator mount design also requires stronger actuation force and/or counterbalances to drive the telescope and therefore adds more weight. Loads through a series-actuation telescope are compounded in series from the telescope down through

each rotational actuator, thus each element from the telescope to ground becomes heavier because each element must support all elements in series above it.

As mentioned, to simplify telescope structure current telescopes do not rotate their OTAs with the rotation of the sky and therefore must have instrument derotators for image rotation that add both costs and weight. Additionally, constructing a three DOF system proves simpler than the construction of a two DOF system in that there isn't need to constrain the base joint. It would be advantageous to make a three DOF system utilizing fewer materials, and thus less expensive, thereby enabling the construction of more telescopes with OTA rotation capabilities.

CHAPTER 3: SOLUTION

It is desired to have a telescope using less mass while maintaining the same stiffness as well as being capable of controlling all three degrees-of-rotation when viewing the sky. It is possible to design and implement a three DOF parallel actuator design for telescope application. Due to the stiffness-to-weight ratio and positioning resolution exceeding those of conventional serial-actuators, the creation of a parallel actuator telescope mount will enable telescope actuation of research-grade automation and available to the research community at a lower monetary cost. This chapter will cover the theory describing operation and design of such a system.

3.1 Proposed Design

A telescope can be controlled utilizing linear actuators in parallel that provide a higher system stiffness compared to a series-actuator telescope of the same weight. Linear actuators come in a variety of forms including hydraulics, pneumatics, linear motors, lead-screws, etc. Lead-screws were chosen as a suitible option for this thesis due to their high-stiffness, adaptable stroke length beyond the range $L \leq sroke \leq 2L$, and lack of potential for leaking aside from lubrication.

As described in Chapter 2, a three DOF system allowing the telescope to rotate about all three axes of rotation provides the capability of capturing long exposure images during star tracking without the need of an instrument de-rotator. Attaining a mount capable of three DOF means creating a dynamic system of a mobility of three. Recalling Gruebler's equation, Equation 1.10, mobility is defined as the mechanism's motion controlled through the combination of links and joints. To achieve a mobility of three, the links and joints for the telescope mount are defined in Table 3.1 and Table 3.2.

Table 3.1: *3-DOF mount Gruebler links.*

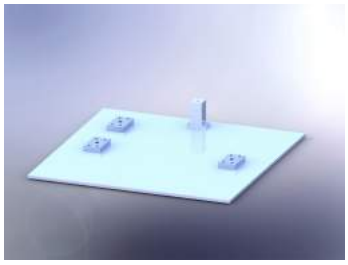
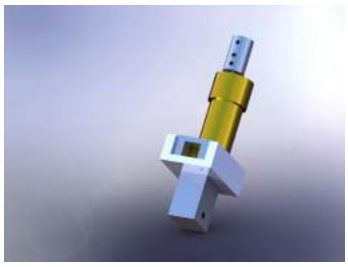
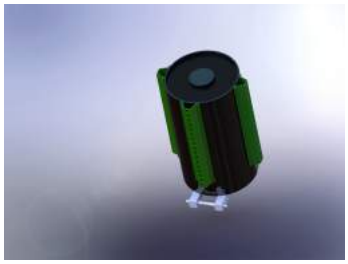
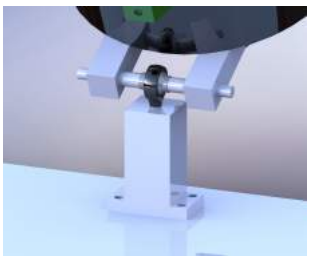
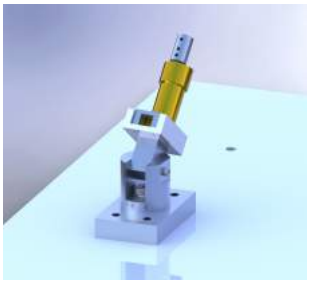
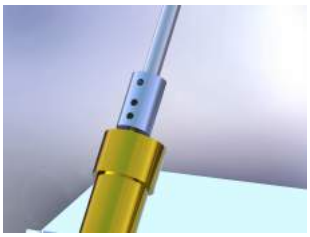
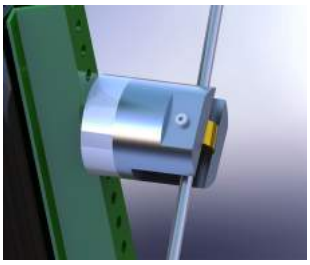
Link	**Quantity**	**Picture**
GND	1	
MOT	3	
ROD	3	
OTA	1	

Table 3.2: *3-DOF mount Gruebler joints.*

Joint	Designation	Type	Quantity	Picture
GND to OTA	J_3	Spherical	1	
GND to MOT	J_2	Dual-Revolute	3	
MOT to ROD	J_1	Revolute	3	
ROD to OTA	J_3	Dual-Revolute and Screw	3	

Inserted into Equation 1.10, this combination of links and joints produces:

$$DOF_{SPATIAL} = 6(L-1) - 5J_1 - 4J_2 - 3J_3 - 2J_4 - J_5$$
$$DOF_{SPATIAL} = 6(8-1) - 5(3) - 4(3) - 3(4) - 2(0) - 0$$
$$DOF_{SPATIAL} = 3$$

Specifying the angles between the motors and threaded rods will constrain the three revolute joints thereby constraining the remaining three DOF of the entire system.

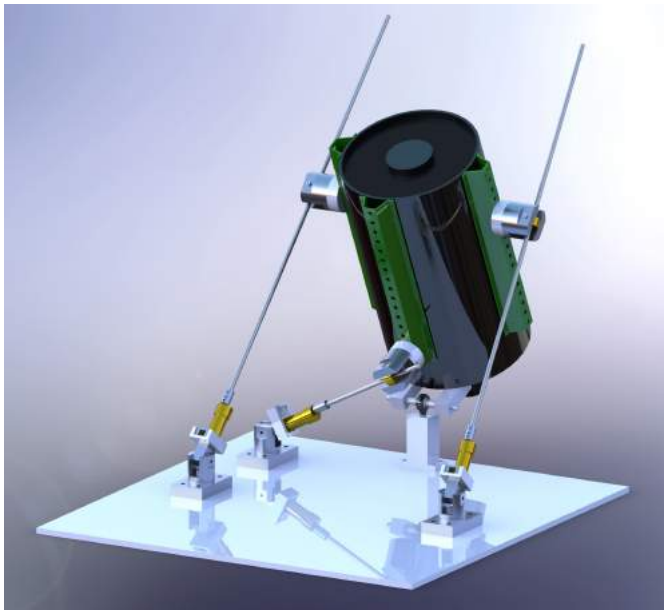


Figure 3.1: *Proposed 3-DOF telescope mount design.*

In typical applications lead-screw type linear actuators have bearings at either end of the rod to minimize side loads and flexing during motion. Utilizing double hinge joints at either end of the screw, the lead-screw becomes a two-force member and any flexing can be assumed to be negligible until buckling occurs. Lead-screw buckling is designed to not occur within the operational usage of the telescope mount. The use of two-force members increases design options for parallel actuator systems, e.g., the allowance of free floating lead-screw ends without the weight of unnecessary support structure.

Figure 3.1 displays my parallel actuator telescope mount system allowing three DOF of rotation about a spherical joint at the base of the telescope. Three lead-screw linear actuators constrain the three DOF of the telescope. The two contact points

of each actuator, one at ground and one at the OTA, consist of dual-revolute joints ensuring only axial loads are experienced by each of the lead-screws.

3.2 Operating Calculations

When using a parallel actuator telescope mount system there is no longer a single formula relating the telescope direction and the required actuator drive distances. Instead it is possible to rotate a mathematical representation of the telescope's orientation and back-calculate the necessary actuator lengths. This rotation of the telescope and determination of points-of-interest is accomplished by use of reference frame transformations.

The calculations laid out in this chapter assume the following as depicted in Figure 3.2:

1. The telescope is positioned in the northern hemisphere;
2. The Universe frame at the telescope location is oriented with its x-axis directed along the zenith;
3. The Universe frame at the telescope location is oriented with its z-axis directed toward the North Celestial Pole; and,
4. The mount is located on a flat surface with no roll or pitch angle.

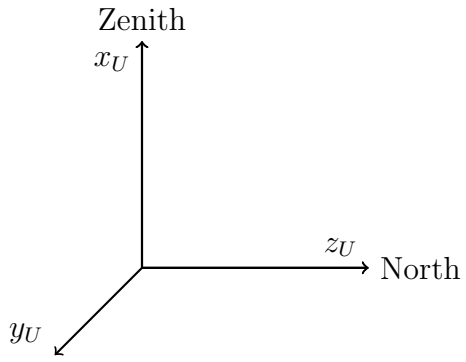


Figure 3.2: *Universe frame directing the x-axis toward the zenith and directing the z-axis toward the North Celestial Pole.*

3.2.1 *Theory*

Lengths

Using a three DOF mount, there are seven points-of-interest (hardpoints) in the mount used to calculate the actuator lengths depicted in Figure 3.3. The yellow point represents the telescope OTA's point-of-rotation and each remaining color pair represents each actuator's points-of-rotation about ground and about the telescope OTA. While the OTA point-of-rotation and the three actuator ground connections remain constant throughout OTA movement, the remaining three points-of-interest will rotate in constant geometry about the OTA point-of-rotation. Referencing Figures 3.3 and 3.4, note the change in the telescope hardpoint positions from $p_{OTAx-zero}$ to p_{OTAx} after accomplishing a 30 degree altitude realignment. One can calculate the linear actuator lengths by use of vector mathematics as defined by Equation 3.1a, b, c.

$$\begin{aligned} a)\ Length_{Act1} &= \|p_{OTA1} - p_{GND1}\| \\ b)\ Length_{Act2} &= \|p_{OTA2} - p_{GND2}\| \\ c)\ Length_{Act3} &= \|p_{OTA3} - p_{GND3}\| \end{aligned} \tag{3.1}$$

Further explanation on how to calculate p_{OTA} variables will be discussed in Section 3.2.4.

Applied to Astronomy

A unique address is comprised of independ of time and location on earth but instead based on components comprised of right ascention and declination Stars and planetary bodies are assigned coordinate "addresses" independent of time and location on earth comprised of *right-ascension* and *declination*. Subsequently, this unique address can be translated into components of altitude, azimuth, and roll for any given latitude, longitude, and calendar time and date. Once the astronomical point-of-interest is defined, telescope actuator lengths can be found by tranforming the telscope reference frame between the universe frame until it aligns with the desired astronomical reference frame.

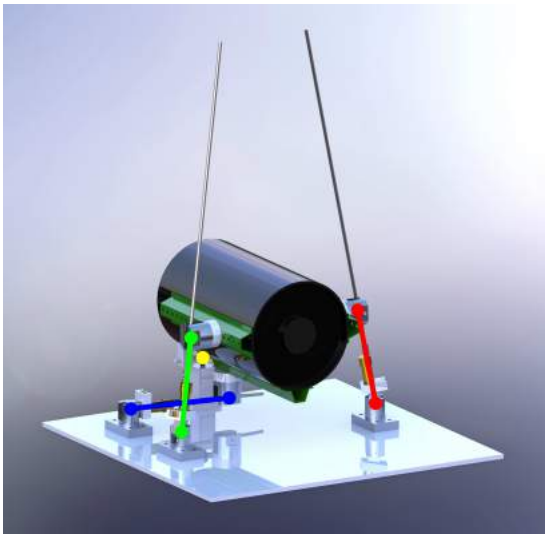


Figure 3.3: *Telescope hardpoints at the zero-position.*

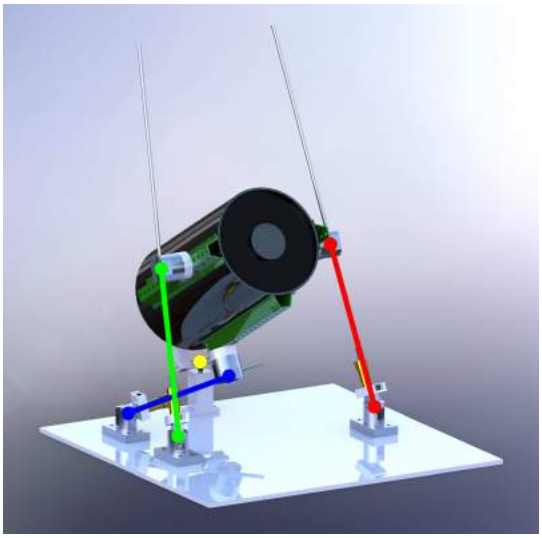


Figure 3.4: *Telescope hardpoints after a* 30° *altitude transformation.*

Figures 3.5, 3.6, and 3.7 illustrate the progression of performing all three rotations on the telescope mount's relative reference frame thereby orienting the telescope to any given astronomical point-of-interest. This progression can be represented by the transformation

$$F_{mount} = Rot(x, \theta_1)Rot(y, \theta_2)Rot(z, \theta_3) \tag{3.2}$$

θ_1 Rotation angle about zenith ($azimuth = -\theta_1$)

θ_2 Altitude angle

θ_3 Image rotation angle

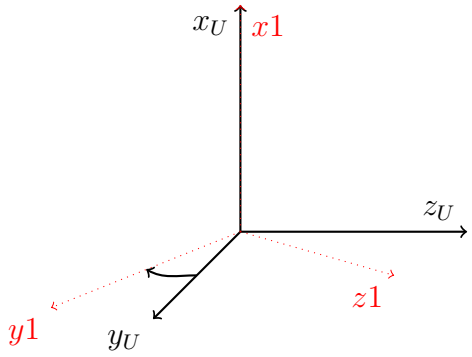


Figure 3.5: *Negative rotation about the x-axis for the azimuth angle indicated in red.*

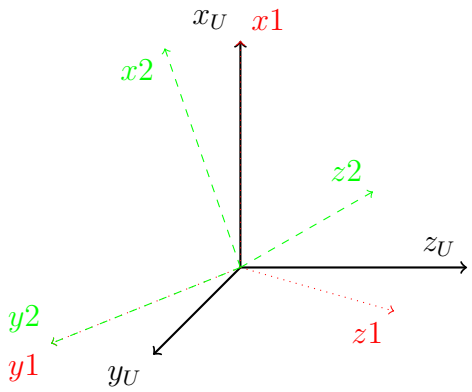


Figure 3.6: *Adding a positive rotation about the relative y-axis for the altitude angle indicated in green (built upon Figure 3.5).*

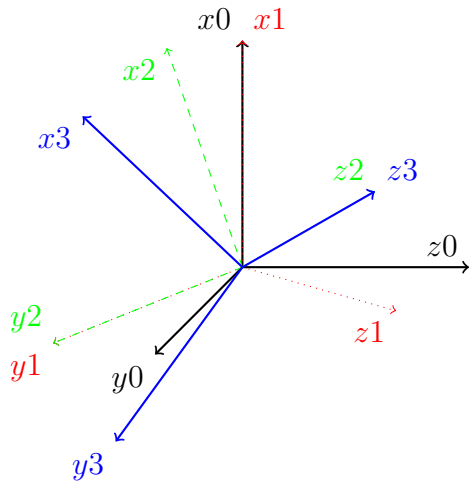


Figure 3.7: *Adding a positive rotation about the relative z-axis for the image-rotation angle indicatated in blue (built upon Figure 3.6).*

Telescope Alignment Correction

If a telescope is set up with the optics perfectly aligned with the mounting hardware and is directed towards $Azimuth = 0°$, $Altitude = 0°$ when $\theta_1 = \theta_2 = \theta_3 = 0°$, then the mount reference frame F_{mount} will be equal to the OTA reference frame F_{OTA} as defined in Equation 3.2. Based on the geographic location of the astronomical observatory it is not always ideal to have a telescope mount directed toward the North Celestial Pole, particularly with a limited field-of-view telescope mount. When redirecting the telescope, the mathematically described frame transformation can be accounted for by pre-multiplying by $Rot(x, \phi_1)$, where ϕ_1 is the azimuth difference between the telescope mount frame and the North Celestial Pole.

$$F_{OTA} = Rot(x, \phi_1)[Rot(x, \theta_1)Rot(y, \theta_2)Rot(z, \theta_3)] \tag{3.3}$$

Setup of the telescope is unlikely to match alignment between the OTA and the mount hardware. This difference between the mount and the optics affects the relative reference frame of the telescope and thus is post-multiplied to the telescope mount transformation. The horizontal offset is represented by ϕ_2 and the vertical offset is represented by ϕ_3.

$$F_{OTA} = Rot(x, \phi_1)[Rot(x, \theta_1)Rot(y, \theta_2)Rot(z, \theta_3)]Rot(x, \phi_2)Rot(y, \phi_3) \tag{3.4}$$

ϕ_1 Azimuth offset ($azimuth = -\phi_1$)

ϕ_2 Telescope horizontal offset

ϕ_3 Telescope vertical offset

These alignment angles can be calculated and remain as constants throughout telescope operation.

3.2.2 *Calibration of Telescope Mount Offsets*

In order to solve for the three constant ϕ variables associated with the mount setup in Equation 3.4, users can manually input θ values into the system until the OTA is directed toward known celestial objects. As previously defined in Section 3.1 the OTA viewing vector v_{OTA} and celestial body's direction vector v_{star} are both along their respective reference frame z-axis. If the celestial object is viewable and centered through the OTA then the z-axis of the OTA is aligned with the z-axis of the object.

$$v_{star} = v_{OTA}$$

where:

$$\begin{aligned} v_{z-axis} &= \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \\ v_{star} &= F_{star} \cdot v_{z-axis} \\ v_{OTA} &= F_{OTA} \cdot v_{z-axis} \end{aligned}$$

It is not necessary to match the entire reference frame of the OTA with the reference frame of the celestial object. The significant computation relating the rotation of the image is not necessary because calibration can be found by aligning the OTA viewing vector with multiple celestial direction vectors in succession. The mount offset calibration process is easiest to visualize in altitude-azimuth coordinate system as the celestial body reference frame F_{star} can then be defined as:

$$F_{star} = Rot(x, -Azimuth)Rot(y, Altitude)$$

The associated direction vector v_{star} is then defined as:

$$v_{star} = Rot(x, -Azimuth)Rot(y, Altitude) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} v_{star_{zx}} \\ v_{star_{zy}} \\ v_{star_{zz}} \end{bmatrix}$$

or:

$$\begin{bmatrix} v_{star_{zx}} \\ v_{star_{zy}} \\ v_{star_{zz}} \end{bmatrix} = \begin{bmatrix} S(Altitude) \\ S(Azimuth) \cdot C(Altitude) \\ C(Azimuth) \cdot C(Altitude) \end{bmatrix}$$

All variables are known so the resulting vector v_{star} will be constant for each altitude-azimuth address. To find the correcsponding OTA viewing vector, Equation 3.4 can be truncated to:

$$v_{OTA} = Rot(x, \phi_1)[Rot(x, \theta_1)Rot(y, \theta_2)Rot(z, \theta_3)]Rot(x, \phi_2)Rot(y, \phi_3) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The vector v_{OTA} is referenced later as:

$$v_{OTA} = \begin{bmatrix} v_{OTA_{zx}} \\ v_{OTA_{zy}} \\ v_{OTA_{zz}} \end{bmatrix}$$

When v_{OTA} is expanded, $v_{OTA_{zx}}$ and $v_{OTA_{zy}}$ have combinations of ϕ_1, ϕ_2, and ϕ_3. Subsequently, $v_{OTA_{zz}}$ can be ignored. Combining terms containing ϕ, the first term $v_{OTA_{zx}}$ can be expressed as:

$$v_{OTA_{zx}} = S\theta_2 x_1 - C\theta_2 S\theta_3 x_2 + C\theta_2 C\theta_3 x_3 \tag{3.5}$$

where:

$$\begin{aligned} x_1 &= C\phi_2 C\phi_3 \\ x_2 &= S\phi_2 C\phi_3 \\ x_3 &= S\phi_3 \end{aligned}$$

Because ϕ values are constants then x_1, x_2, and x_3 will also be constants. A minimum of three sets of points are required to solve x values through this system of linear

equations, but it is recommended to use more points.

$$\begin{bmatrix} (S\theta_2)_1 & (-C\theta_2 S\theta_3)_1 & (C\theta_2 C\theta_3)_1 \\ (S\theta_2)_2 & (-C\theta_2 S\theta_3)_2 & (C\theta_2 C\theta_3)_2 \\ (S\theta_2)_3 & (-C\theta_2 S\theta_3)_3 & (C\theta_2 C\theta_3)_3 \\ & ... & \\ (S\theta_2)_n & (-C\theta_2 S\theta_3)_n & (C\theta_2 C\theta_3)_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} v_{star_{zx_1}} \\ v_{star_{zx_2}} \\ v_{star_{zx_3}} \\ ... \\ v_{star_{zx_n}} \end{bmatrix} \tag{3.6}$$

With Equation 3.6 already in the form $Ax = b$, x can be solved through use of the least squares equation:

$$A^T Ax = A^T b$$

$$x = (A^T A)^{-1} A^T b$$

After x is solved, ϕ_3 and ϕ_2 can be solved using x_3 and x_2:

$$\phi_3 = asin(x_3)$$

$$\phi_2 = asin(\frac{x_2}{C\phi_3})$$

Proceeding, the second term v_{OTAzy} expands to a much longer equation but of the six angle terms only ϕ_1 is unknown. Subsequently, v_{OTAzy} can be expressed as:

$$v_{OTA_{zy}} = c_1 x_1 + c_2 x_2 \tag{3.7}$$

where:

$$\begin{aligned} c1 &= C\phi_3(S\phi_2[S\theta_3 S\theta_2 S\theta_1 + C\theta_3 C\theta_1] + C\phi_2 C\theta_2 S\theta_1) - S\phi_3(C\theta_3 S\theta_2 S\theta_1 - S\theta_3 C\theta_1) \\ x1 &= C\phi_1 \\ c2 &= C\phi_3(S\phi_2[S\theta_3 S\theta_2 C\theta_1 - C\theta_3 S\theta_1] + C\phi_2 C\theta_2 C\theta_1) - S\phi_3(C\theta_3 S\theta_2 C\theta_1 + S\theta_3 S\theta_1) \\ x2 &= S\phi_1 \end{aligned}$$

Using all celestial locations contributing to Equation 3.6, Equation 3.7 can be expanded into the form $Ax = b$:

$$\begin{bmatrix} c_{1_1} & c_{2_1} \\ c_{1_2} & c_{2_2} \\ c_{1_3} & c_{2_3} \\ ... & \\ c_{1_n} & c_{2_n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} v_{star_{zy_1}} \\ v_{star_{zy_2}} \\ v_{star_{zy_3}} \\ ... \\ v_{star_{zy_n}} \end{bmatrix} \tag{3.8}$$

Solving Equation 3.8 using the least squares method provides the means to solve for ϕ_1. Because ϕ_1 has the potential to be outside the range $-90^\circ < \phi_1 < 90^\circ$, it is important to select an angle of the proper quadrant. Thus both x_1 and x_2 should be used to find ϕ_1:

$$\phi_1 = atan2(x_2, x_1)$$

To ensure a linearly independent set in Equations 3.6 and 3.8, a minimum of two θ angles must be used for each axis of rotation over the course of the three or more known celestial locations.

3.2.3 *Definition of the Star Transformation Matrix*

The star transformation matrix, F_{star}, represents the transformation matrix required to redirect viewing angle from the universe frame defined in Figure 3.2 to a desired celestial location with a defined image rotation angle. Calculation of F_{star} using altitude-azimuth coordinates allows for minimal input in regard to geographic location. However, tracking an object defined by this coordinate system will require constant updating from either the telescope operator or a planetarium software program directing the telescope to new viewing angles. Calculation of F_{star} using the right ascension and declination coordinate system requires geographic latitude and longitude as well as local time. This calculation can be automated without the use of planetarium software while maintaining OTA rotation with the targeted celestial body.

Altitude / Azimuth Coordinates

In this coordinate system the operator will manually input values for altitude, azimuth, and the desired image rotation. Because the universe frame is defined relative to the North Celestial Pole along the horizon the transformation matrix F_{star} is defined as:

$$F_{star_{Alt-Az}} = Rot(x, -Azimuth)Rot(y, Altitude)Rot(z, ImageRotation) \qquad (3.9)$$

Right Ascension / Declination Coordinates

An argument can be made that star tracking is easier to automate using the celestial coordinate system because the rotation of the earth is removed as a factor. The OTA is first directed the OTA to the celestial pole, then rotated to the desired right ascension angle, and then likewise rotated to the desired declination angle. Rotation of the OTA relative to the celestial coordinate system maintains constant targeting of a celestial body, including the target's rotation, thus enabling long imaging and masking exposure times.

The celestial pole resides at an angle above the horizon equivalent to the telescope location's latitude and therefore this angle is defined as ϕ_{lat}. Using California Polytechnic State University at San Luis Obispo (Cal Poly SLO) as an example, the campus latitude is 35.3°N and therefore Polarius, the celestial pole, is observed at an altitude of $\phi_{lat} = 35.3°$.

Having moved the coordinate system view angle to the celestial pole, the rotation angle must be established to attain the subsequent right ascension angle (RA). The telescope operator must determine the telescope's local sidereal time (LST) using astronomy software or through approximation [9]. LST represents the right ascension angle directed along the zenith angle. The LST is therefore calculated based on the telescope's geographic longitude and local time. The local time is often defined by utilizing a date convention, e.g., J2000 represents the time elapsed since 1200 Universal Time (UT) on the date of January 1, 2000. LST is then used to calculate the hour angle (HA) defined as the angle difference between the current LST and the desired RA. Hour angle rotation, represented by θ_{HA}, is achieved by a negative rotation about a relative z-axis, or current viewing direction.

$$HA = LST - RA$$

Declination is defined as 0° about the equator and 90° when directed at the North celestial pole. The angle of travel from the celestial pole to the desired declination, θ_{Dec}, is defined by:

$$\theta_{Dec} = 90° - Declination$$

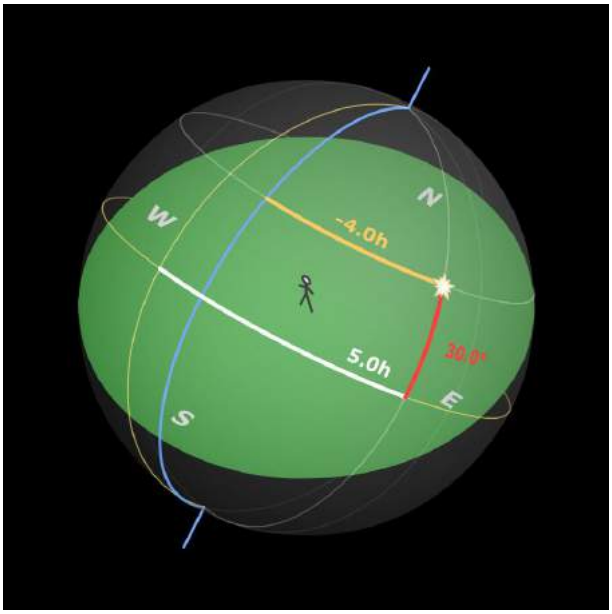


Figure 3.8: *Example hour angle: LST=1h, RA=5h, HA=-4h [5].*

Utilizing ϕ_{lat}, θ_{HA}, and θ_{Dec}, F_{star} can be defined as:

$$F_{star} = Rot(y, \phi_{lat})Rot(z, -\theta_{HA})Rot(y, \theta_{Dec})$$

However, if the hour angle is outside the range $-1h \leq HA \leq 1h$, or $-15^{\circ} \leq \theta_{HA} \leq 15^{\circ}$ then the telescope mount will have difficulty achieving the necessary OTA rotation. An excessive OTA rotation can be restricted by applying a constant offset, $\phi_{offset} = \theta_{HA} + 15^{\circ}$, to the relative z-axis. While celestial locations still need to be confirmed within telescope viewing capabilities in the altitude and azimuth coordinates, this excessive OTA rotation offset keeps the OTA within the system limited two hour roll angle range for two hours.

$$F_{star} = Rot(y, \phi_{lat})Rot(z, -\theta_{HA})Rot(y, \theta_{Dec})Rot(z, \phi_{offset}) \tag{3.10}$$

3.2.4 *Directing the Telescope*

In order to direct the OTA to a particular celestial location, the telescope controller unit requires actuator length changes. To calculate these actuator lengths it is necessary to determine point coordinates of the actuator connections to establish the OTA in the desired configuration. The first step in this process is to determine

what coordinate system to use and celestial location to find. A location can be targeted in units of altitude and azimuth coordinates or right ascension and declination coordinates, either method will produce the direction transformation matrix F_{star}.

To establish the OTA in the desired direction the transformation matrices for the celestial location and OTA are equivalent:

$$F_{star} = F_{OTA} \tag{3.11}$$

To determine how the mount can configure itself to achieve the desired OTA target angles, Equations 3.2 and 3.4 are substituted into Equation 3.11 to find F_{mount}.

$$F_{star} = Rot(x, \phi_1) F_{mount} Rot(x, \phi_2) Rot(y, \phi_3)$$

$$F_{mount} = Rot(x, \phi_1)^{-1} F_{star} [Rot(x, \phi_2) Rot(y, \phi_3)]^{-1} \tag{3.12}$$

After F_{mount} has been defined it can be used to transform hardpoints p_{OTA1}, p_{OTA2}, and p_{OTA3} from their predefined zero-position in the matrix P_{zero} to the new coordinates, P_1, that are required to direct the OTA. Note that p_{OTAx} represents the x, y, z coordinates of the points after the transformation and $p_{OTAx-zero}$ represents the predefined coordinates of each point with mount angles $\theta_1 = \theta_2 = \theta_3 = 0$.

$$P_1 = F_{mount} P_{zero}$$

where:

$$P_1 = \begin{bmatrix} p_{OTA1} & p_{OTA2} & p_{OTA3} \end{bmatrix} \quad and \quad P_{zero} = \begin{bmatrix} p_{OTA1-zero} & p_{OTA2-zero} & p_{OTA3-zero} \end{bmatrix}$$

When new actuator coordinates have been defined, the last step is to calculate the actuator lengths, L_x, as the magnitude of the vector between p_{OTAx} and p_{GNDx}, the latter representing the coordinates of each actuator's connection to ground.

$$\begin{aligned} L_1 &= \|p_{OTA1} - p_{GND1}\| \\ L_2 &= \|p_{OTA2} - p_{GND2}\| \\ L_3 &= \|p_{OTA3} - p_{GND3}\| \end{aligned}$$

3.2.5 *Calculation of Telescope Direction From Actuator Lengths*

As shown in Section 3.2.4, calculating the actuator lengths from desired direction angles requires the use of trigonometry and Pythagorean equations. The calculations can be done progressively without the need for solving equations simultaneously. Calculating the direction angles from actuator lengths requires using those same equations that are nonlinear and have no simple solutions in the form $Ax = b$. To solve nonlinear systems of equations the Newtonian update method can be employed. In this method u represents the variables being solved, $G(u) = 0$ represents the system boundary equations, and $J = \nabla G(u)$ represents the Jacobian matrix. The Newtonian iterative update of variables is solved through Equations 3.13 and 3.14:

$$u^{[j+1]} = u^{[j]} - \delta^{[j]} \tag{3.13}$$

$$J^{[j]}\delta^{[j]} = G(u^{[j]}) \tag{3.14}$$

Knowing the iterative process to be used the next step is to determine boundary equations.

The process of solving the mount direction angles begins by finding the coordinates of each actuator's connection to the OTA represented as points p_{OTA1}, p_{OTA2}, and p_{OTA3} according to the current actuator lengths of L_1, L_2, and L_3. These coordinates can be used to find the reference frame transformation F_{mount}, and end with using this transformation to determine the three θ angles.

To find p_{OTA1}, p_{OTA2}, p_{OTA3}, there are nine variables that must be solved. The notation for these variables will be labeled as:

$$p_{OTA1} = \begin{bmatrix} x_{11} \\ y_{11} \\ z_{11} \end{bmatrix} \quad , \quad p_{OTA2} = \begin{bmatrix} x_{21} \\ y_{21} \\ z_{21} \end{bmatrix} \quad , \quad p_{OTA3} = \begin{bmatrix} x_{31} \\ y_{31} \\ z_{31} \end{bmatrix}$$

Because these OTA points rotate in constant geometry about the OTA base (spherical joint), the distances between OTA points and the distances from each OTA point to the OTA base remain constant through telescope motion and can be defined as constants at the beginning of telescope operation. Note that the location of the OTA

base, p_{base}, is the reference frame origin or the zero-vector. Also note that points labeled *zero* are the coordinates of the actuator-OTA connection when the mount is defined at $\theta_1 = \theta_2 = \theta_3 = 0$.

$$p_{OTA1-zero} = \begin{bmatrix} x_{10} \\ y_{10} \\ z_{10} \end{bmatrix} \quad , \quad p_{OTA2-zero} = \begin{bmatrix} x_{20} \\ y_{20} \\ z_{20} \end{bmatrix} \quad , \quad p_{OTA3-zero} = \begin{bmatrix} x_{30} \\ y_{30} \\ z_{30} \end{bmatrix}$$

where:

$$\begin{aligned} D_{12} &= \|p_{OTA1-zero} - p_{OTA2-zero}\| \\ D_{13} &= \|p_{OTA1-zero} - p_{OTA3-zero}\| \\ D_{23} &= \|p_{OTA2-zero} - p_{OTA3-zero}\| \\ D_{1b} &= \|p_{OTA1-zero} - p_{base}\| = \|p_{OTA1-zero}\| \\ D_{2b} &= \|p_{OTA2-zero} - p_{base}\| = \|p_{OTA2-zero}\| \\ D_{3b} &= \|p_{OTA3-zero} - p_{base}\| = \|p_{OTA2-zero}\| \end{aligned}$$

The relationship of these points and lengths provide six equations, and the following three equations can be found utilizing the known lengths of the actuators.

$$\begin{aligned} L_1 &= \|p_{OTA1} - p_{GND1}\| \\ L_2 &= \|p_{OTA2} - p_{GND2}\| \\ L_3 &= \|p_{OTA3} - p_{GND3}\| \end{aligned}$$

Recall that p_{GNDx} represent the coordinates of each actuator's point-of-rotation at the ground. In total this provides nine nonlinear equations for the nine coordinate variables to be solved in the Newtonian update method. Subsequently the u, G, and J matrices can be defined as follows:

$$u_1 = \begin{bmatrix} x_{11} \\ y_{11} \\ z_{11} \\ x_{21} \\ y_{21} \\ z_{21} \\ x_{31} \\ y_{31} \\ z_{31} \end{bmatrix} \quad , \quad G_1 = \begin{bmatrix} l_{11-10} - L_1 \\ l_{21-20} - L_2 \\ l_{31-30} - L_3 \\ l_{11-21} - D_{12} \\ l_{11-31} - D_{13} \\ l_{21-31} - D_{23} \\ l_{11-b} - D_{1b} \\ l_{21-b} - D_{2b} \\ l_{31-b} - D_{3b} \end{bmatrix}$$

$$J_1 = \begin{bmatrix}
\frac{x_{11}-x_{10}}{l_{11-10}} & \frac{y_{11}-y_{10}}{l_{11-10}} & \frac{z_{11}-z_{10}}{l_{11-10}} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{x_{21}-x_{20}}{l_{21-20}} & \frac{y_{21}-y_{20}}{l_{21-20}} & \frac{z_{21}-z_{20}}{l_{21-20}} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{x_{31}-x_{30}}{l_{31-30}} & \frac{y_{31}-y_{30}}{l_{31-30}} & \frac{z_{31}-z_{30}}{l_{31-30}} \\
\frac{x_{11}-x_{21}}{l_{11-21}} & \frac{y_{11}-y_{21}}{l_{11-21}} & \frac{z_{11}-z_{21}}{l_{11-21}} & \frac{-(x_{11}-x_{21})}{l_{11-21}} & \frac{-(y_{11}-y_{21})}{l_{11-21}} & \frac{-(z_{11}-z_{21})}{l_{11-21}} & 0 & 0 & 0 \\
\frac{x_{11}-x_{31}}{l_{11-31}} & \frac{y_{11}-y_{31}}{l_{11-31}} & \frac{z_{11}-z_{31}}{l_{11-31}} & 0 & 0 & 0 & \frac{-(x_{11}-x_{31})}{l_{11-31}} & \frac{-(y_{11}-y_{31})}{l_{11-31}} & \frac{-(z_{11}-z_{31})}{l_{11-31}} \\
0 & 0 & 0 & \frac{x_{21}-x_{31}}{l_{21-31}} & \frac{y_{21}-y_{31}}{l_{21-31}} & \frac{z_{21}-z_{31}}{l_{21-31}} & \frac{-(x_{21}-x_{31})}{l_{21-31}} & \frac{-(y_{21}-y_{31})}{l_{21-31}} & \frac{-(z_{21}-z_{31})}{l_{21-31}} \\
\frac{x_{11}}{l_{11-b}} & \frac{y_{11}}{l_{11-b}} & \frac{z_{11}}{l_{11-b}} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{x_{21}}{l_{21-b}} & \frac{y_{21}}{l_{21-b}} & \frac{z_{21}}{l_{21-b}} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{x_{31}}{l_{31-b}} & \frac{y_{31}}{l_{31-b}} & \frac{z_{31}}{l_{31-b}}
\end{bmatrix}$$

where:

$$\begin{aligned}
l_{11-10} &= \sqrt{(x_{11}-x_{10})^2+(y_{11}-y_{10})^2+(z_{11}-z_{10})^2} \\
l_{21-20} &= \sqrt{(x_{21}-x_{20})^2+(y_{21}-y_{20})^2+(z_{21}-z_{20})^2} \\
l_{31-30} &= \sqrt{(x_{31}-x_{30})^2+(y_{31}-y_{30})^2+(z_{31}-z_{30})^2} \\
l_{11-21} &= \sqrt{(x_{11}-x_{21})^2+(y_{11}-y_{21})^2+(z_{11}-z_{21})^2} \\
l_{11-31} &= \sqrt{(x_{11}-x_{31})^2+(y_{11}-y_{31})^2+(z_{11}-z_{31})^2} \\
l_{21-31} &= \sqrt{(x_{21}-x_{31})^2+(y_{21}-y_{31})^2+(z_{21}-z_{31})^2} \\
l_{11-b} &= \sqrt{(x_{11}-0)^2+(y_{11}-0)^2+(z_{11}-0)^2} \\
l_{21-b} &= \sqrt{(x_{21}-0)^2+(y_{21}-0)^2+(z_{21}-0)^2} \\
l_{31-b} &= \sqrt{(x_{31}-0)^2+(y_{31}-0)^2+(z_{31}-0)^2}
\end{aligned}$$

Assuming values L_1, L_2, and L_3 are within range of the telescope's physical capability, values in u will converge after nine update iterations of Equation 3.13 and Equation 3.14. These points can be assembled into point matrices

$$P_1 = \begin{bmatrix} x_{11} & x_{21} & x_{31} \\ y_{11} & y_{21} & y_{31} \\ z_{11} & z_{21} & z_{31} \end{bmatrix} \quad , \quad P_{zero} = \begin{bmatrix} x_{10} & x_{20} & x_{30} \\ y_{10} & y_{20} & y_{30} \\ z_{10} & z_{20} & z_{30} \end{bmatrix}$$

and set into a matrix transformation equation:

$$P_1 = F_{mount} P_{zero}$$

As both P_1 and P_{zero} are known, F_{mount} can be solved:

$$F_{mount} = P_1 (P_{zero})^{-1} = \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{bmatrix} \tag{3.15}$$

After solving for the transformation matrix used to represent the telescope's motion to the new point from zero-position, the three angles of rotation can be solved using the nine equations representing each cell of the transformation matrix, F_{mount}. Subsequently, the T constants on Equation 3.15 are used to define u, G, and J matrices as follows:

$$u_2 = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} \quad , \quad G_2 = \begin{bmatrix} -T_{11} + C\theta_2 C\theta_3 \\ -T_{12} - C\theta_2 S\theta_3 \\ -T_{13} + S\theta_2 \\ -T_{21} + C\theta_1 S\theta_3 + S\theta_1 S\theta_2 C\theta_3 \\ -T_{22} + C\theta_1 C\theta_3 - S\theta_1 S\theta_2 S\theta_3 \\ -T_{23} - S\theta_1 C\theta_2 \\ -T_{31} + S\theta_1 S\theta_3 - C\theta_1 S\theta_2 C\theta_3 \\ -T_{32} + S\theta_1 C\theta_3 + C\theta_1 S\theta_2 S\theta_3 \\ -T_{33} + C\theta_1 C\theta_2 \end{bmatrix}$$

$$J_2 = \begin{bmatrix} 0 & -S\theta_2 C\theta_3 & -C\theta_2 S\theta_3 \\ 0 & S\theta_2 S\theta_3 & -C\theta_2 C\theta_3 \\ 0 & C\theta_2 & 0 \\ -S\theta_1 S\theta_3 + C\theta_1 S\theta_2 C\theta_3 & S\theta_1 C\theta_2 C\theta_3 & C\theta_1 C\theta_3 - S\theta_1 S\theta_2 S\theta_3 \\ -S\theta_1 C\theta_3 - C\theta_1 S\theta_2 S\theta_3 & -S\theta_1 C\theta_2 S\theta_3 & -C\theta_1 S\theta_3 - S\theta_1 S\theta_2 C\theta_3 \\ -C\theta_1 C\theta_2 & S\theta_1 S\theta_2 & 0 \\ C\theta_1 S\theta_3 + S\theta_1 S\theta_2 C\theta_3 & -C\theta_1 C\theta_2 C\theta_3 & S\theta_1 C\theta_3 + C\theta_1 S\theta_2 S\theta_3 \\ C\theta_1 C\theta_3 - S\theta_1 S\theta_2 S\theta_3 & C\theta_1 C\theta_2 S\theta_3 & -S\theta_1 S\theta_3 + C\theta_1 S\theta_2 C\theta_3 \\ -S\theta_1 C\theta_2 & -C\theta_1 S\theta_2 & 0 \end{bmatrix}$$

It is important to note that mount angles, θ_x, are defined in the range $-180° < \theta_x \leq 180°$. During iteration cycles, values θ_1, θ_2, or θ_3 might exceed this specified range and should be redefined by the appropriate coterminal angle before the next iteration.

Astronomical Angles

Note that the solutions in the second set of Newtonian equations, u_2, G_2 and J_2, are relative to F_{mount} coordinates. If it is desired to calculate the astronomical angles, $Altitude$, $Azimuth$, and $ImageRotation$, of the OTA then it is necessary to take into account the OTA mount calibration angles defined in Equation 3.4. Instead of using the T constants from Equation 3.15 in G_2 from F_{mount}, T constants will be determined using Equation 3.16.

$$F_{OTA} = Rot(x, \phi_1) F_{mount} Rot(x, \phi_2) Rot(y, \phi_3) = \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{bmatrix} \tag{3.16}$$

Noting these new T constants, the same equations for u_2, G_2, and J_2 can be utilized except that θ_1 now falls within the range $0^\circ \geq \theta_1 > -360^\circ$ and should be redefined to the appropriate coterminal angle between iterations. Keeping θ_1 negative is intentional as θ_1, θ_2, and θ_3 convert into astronomical angles as follows:

$$\begin{aligned} Azimuth &= -\theta_1 \\ Altitude &= \theta_2 \\ ImageRotation &= \theta_3 \end{aligned}$$

Error Potential

The first set of Newtonian update equations represented by u_1, G_1, J_1 are stable and converge on the correct u_1 solution after nine iterations regardless of the initial guess for u_1. The second set of Newtonian update equations represented by u_2, G_2, J_2 has the potential to converge on false solutions should the initial guess be too distant from the correct solution. Empirical testing of the equation shows that the correct solution converges within nine iterations if the intial guess for all θ angles is within the range $\theta_{actual} - 5^\circ < \theta_{guess} < \theta_{actual} + 5^\circ$. To ensure the initial guess is within the proper range, an actuator length value table can be populated containing actuator lengths for all combinations of θ values at 5° increments. At the start calculating telescope angles from actuator lengths, a close u_2 guess can be found by locating the closest telescope angle point within the value table.

3.3 Determining Telescope Configuration

Empirical tests were iterated to determine new telescope configurations. Given the desired resolution, actuator size, and optical assembly the user can iterate over the desired sky coverage to determine if the test is sufficient for use.

3.3.1 *Actuator Length Test*

As previously defined, actuator length is the distance between the point-of-rotation at the base of each motor assembly and the point-of-rotation at the connection of the threaded rod and OTA assembly. To determine a mount's capability, an operator can mathematically rotate the telescope about the desired celestial coverage to determine each actuator's minimum and maximum length. These minima and maxima must then be compared to the physical system to ensure these orientations are physically possible.

In the case of my prototype defined in Chapter 4, the physical minimum length l_{min} is determined by driving the OTA completely down the threaded rod to the motor assembly and measuring the distance between points-of-rotation. The physical maximum is determined by measuring the distance from the motor assmebly point-of-rotation to the far end of the threaded rod. These lengths are defined in Equations 3.17 and 3.18 and illuestrated in Figure 3.9.

$$l_{min} = l_{motor-assembly} + \tfrac{1}{2} l_{threaded-nut} \tag{3.17}$$

$$l_{max} = l_{motor-assembly} + l_{threaded-rod} \tag{3.18}$$

3.3.2 *Sky Resolution Test*

Sky resolution is the minimum movement a telescope is able to predictably assess. Astronomers typically work with a resolution unit called an arcsecond, or 1/3600 of a degree. The available data feedback devices for this telescope mount are rotary

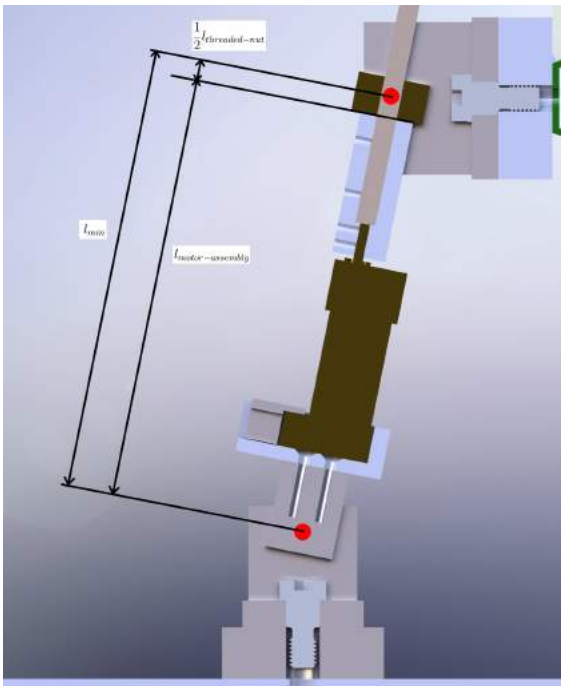


Figure 3.9: *Minimum actuator length.*

encoders placed at the motor shaft. These rotary encoders convert to linear resolution $LinearRes[\frac{inch}{count}]$ by calculating the angular resolution of the encoder, the gear ratio from the motor, and the pitch of the lead-screw as expressed in Equation 3.19.

$$LinearRes[\tfrac{inch}{count}] = \frac{ScrewPitch[\frac{inch}{turn}]}{GearRatio * EncoderRes[\frac{count}{turn}]} \tag{3.19}$$

In order to determine if an actuator's linear resolution is sufficient to meet the required sky resolution the minimum change in actuator length throughout the telescope's field-of-view using the required sky resolution is needed. If the minimum change in all actuator lengths is greater than an actuator's linear resolution then the telescope mount is capable of obtaining the required sky resolution. Because actuators are acting in parallel and do not affect any single output direction of the telescope, the combined minimum actuator travel can be defined as the norm of all three individual travel distances, $ActTravelNorm$.

$$ActTravelNorm = \sqrt{(d_1)^2 + (d_2)^2 + (d_3)^2} \tag{3.20}$$

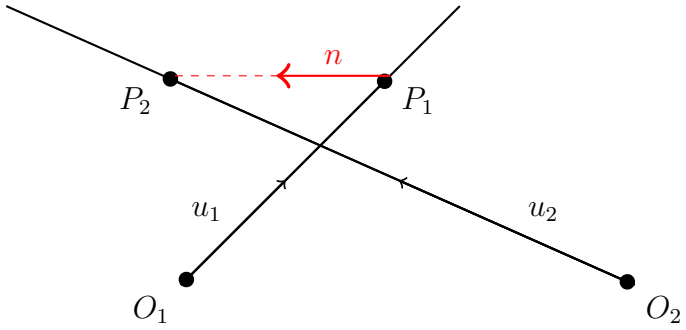


Figure 3.10: *Determining the shortest distance between two skewed lines.*

3.3.3 *Interference Test*

In order to ensure no degenerate actuator combinations are reached, no two axes are physically parallel at any given time. Because of this there is a possibility of interference between actuators. Interference is determined through the calculation of distance between skew lines. There are four lines of interest in this system, comprised of three linear actuator axes and the OTA path. If any two lines - the diameter of each taken into account - conflict over the desired field-of-view of the telescope, then the given configuration will not work.

Referencing Figure 3.10, points O_1 and O_2 represent the base point-of-rotation of any two actuators or OTA and vectors u_1 and u_2 represent the direction each line points in space. Points P_1 and P_2 represent the points along these actuators that are closest to one another. P_1 and P_2 can be solved as depicted in Equation 3.21a and Equation 3.21b, respectively.

$$\begin{aligned} a)\ P_1 &= t_1\overrightarrow{u_1} + O_1 \\ b)\ P_2 &= t_2\overrightarrow{u_2} + O_2 \end{aligned} \tag{3.21}$$

Variables t_1 and t_2 are scalar magnitude values to be calculated. The shortest distance between the two screwed lines lies along the vector n, the normal vector to both u_1 and u_2.

$$n = u_1 \times u_2 \tag{3.22}$$

Both a dot product method and a vector addition method produce the same result to solve for P_1 and P_2. While the dot product method arguably is computationally easier, I chose to use the vector addition method because it provides a more straightforward input into code.

Dot Product Method

The shortest distance between two lines in space is normal to the direction of those two lines, therefore, the dot product of the minimum-distance vector and each line vector is equal to zero as shown in Equations 3.23 a and b.

$$\begin{aligned} a) \quad & \overrightarrow{P_1P_2} \cdot \overrightarrow{u_1} = 0 \\ b) \quad & \overrightarrow{P_1P_2} \cdot \overrightarrow{u_2} = 0 \end{aligned} \tag{3.23}$$

Substituting terms from Equation 3.21 into Equation 3.23, a system of two equations can be assembled in the form $Ax = b$ utilizing the following equations to solve for x.

$$A = \begin{bmatrix} -[u_{1x}^2 + u_{1y}^2 + u_{1z}^2] & [u_{1x}u_{2x} + u_{1x}u_{2x} + u_{1x}u_{2x}] \\ -[u_{1x}u_{2x} + u_{1x}u_{2x} + u_{1x}u_{2x}] & [u_{2x}^2 + u_{2y}^2 + u_{2z}^2] \end{bmatrix} \tag{3.24}$$

$$x = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \tag{3.25}$$

$$b = \begin{bmatrix} (O_{2x} - O_{1x})u_{1x} + (O_{2y} - O_{1y})u_{1y} + (O_{2z} - O_{1z})u_{1z} \\ (O_{2x} - O_{1x})u_{2x} + (O_{2y} - O_{1y})u_{2y} + (O_{2z} - O_{1z})u_{2z} \end{bmatrix} \tag{3.26}$$

After solving for t_1 and t_2, $\overrightarrow{P_1P_2}$ can be found. The magnitude of this vector is the minimum distance between the two lines. If this minimum distance is small enough to cause interference between two linear actuators and/or OTA, the magnitude of vectors $t_1\overrightarrow{u_1}$ and $t_2\overrightarrow{u_2}$ will determine if interference can be bypassed beyond the tip of either actuator.

Vector Addition Method

The vectors between all points-of-interest form a polygon, therefore the polygon law of vector addition can be utilized:

If a number of vectors are represented completely by different sides of a polygon taken in an order, then their resultant is completely given by the closing side of the polygon taken in opposite order [17, p.4].

The points surrounding the polygon in Figure 3.10 are O_1, O_2, P_1, and P_2, when combined can be described by Equation 3.27.

$$\overrightarrow{O_1O_2} = \overrightarrow{O_1P_1} + \overrightarrow{P_1P_2} + \overrightarrow{P_2O_2} \tag{3.27}$$

where:

$$\overrightarrow{O_1P_1} = t1\overrightarrow{u_1}$$
$$\overrightarrow{P_2O_2} = -t_2\overrightarrow{u_2}$$
$$\overrightarrow{P_1P_2} = t_3\overrightarrow{n}$$

Using only Equation 3.27 without further substitution, a system of three equations can be described in the form $Ax = b$ utilizing the following:

$$A = \begin{bmatrix} \overrightarrow{u_1} & -\overrightarrow{u_2} & \overrightarrow{n} \end{bmatrix} \tag{3.28}$$

$$x = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \tag{3.29}$$

$$b = \begin{bmatrix} \overrightarrow{O_1O_2} \end{bmatrix} \tag{3.30}$$

After finding scalar variables t_1, t_2, and t_3, the minimum distance is the magnitude component of vector $t_3\overrightarrow{n}$. If this minimum distance is short enough to cause interference between two actuators, the magnitude component of vectors $t_1\overrightarrow{u_1}$ and $t_2\overrightarrow{u_2}$ will determine if interference will be caused within the maximum travel length of the actuators.

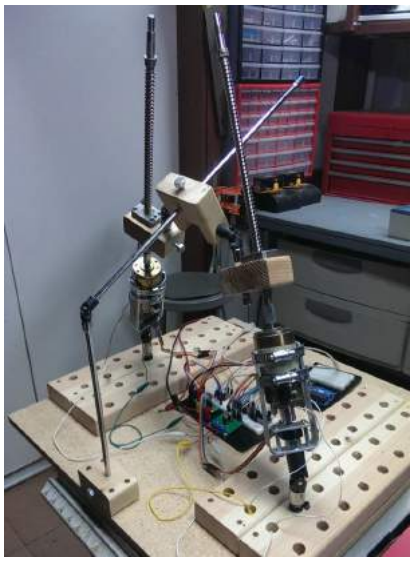


Figure 3.11: *2-DOF proof-of-concept model.*

3.4 Proof of Concept

To demonstrate the feasibility of a parallel actuator telescope mount in operation, a two degree-of-freedom proof-of-concept model was created. A universal joint located at the base of the primary rod simulating the OTA allowed two degrees of rotation. These were subsequently constrained by use of two linear actuators.

The proof-of-concept model also offered the opportunity to test electrical components possibly to be implemented on the final prototype. An *Arduino Uno* microcontroller performed position control utilizing DC motor drivers, shown in Figure 3.12, for the two linear actuators over a series of predetermined positions thus directing the primary rod.

A grid of mounting positions, shown in Figure 3.11, was created for varying the actuator base locations. This allowed the testing of a multitude of actuator configurations to evaluate difference in interference, telescope slew speed, and relative resolution. Because of the mechanical slop in workbench materials for the proof-of-concept (the joints were comprised of impact universal joints) holding accuracy could slop up to multiple degrees. However the proof-of-concept model was successful in that it could successfully track horizontally across its field-of-view at low pitch attitudes and maintain control at varying angles around zenith in configurations such as

in Figures 3.13 and 3.14 without interference between the actuators and simulated-OTA.

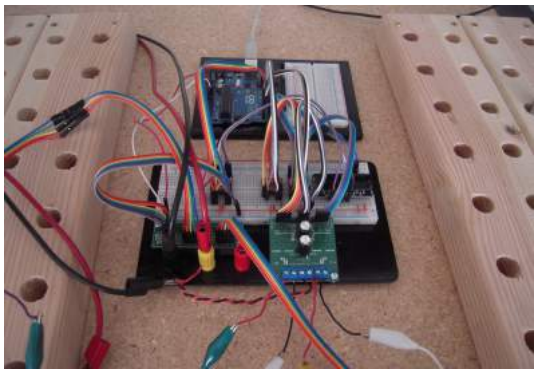


Figure 3.12: *Electronics driving the 2-DOF proof-of-concept model consisting of an Arduino Uno microcontroller, LS7366R encoder counters, and VNH2SP30 motor drivers.*

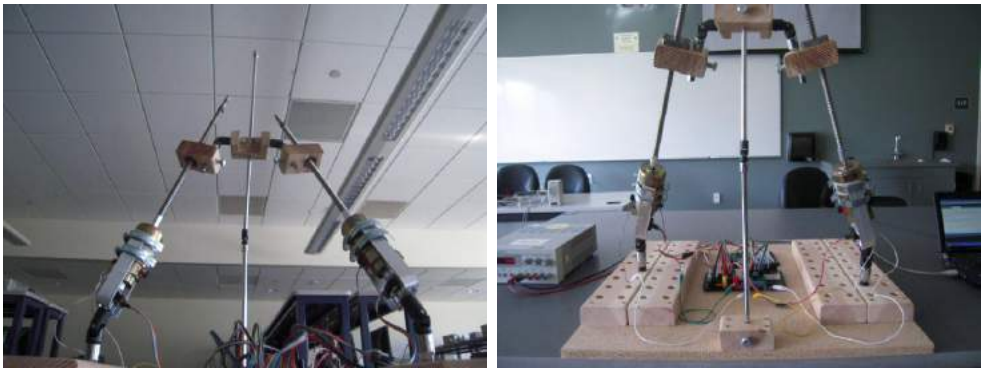


Figure 3.13: *2-DOF proof-of-concept model demonstrating motion about the zenith.*

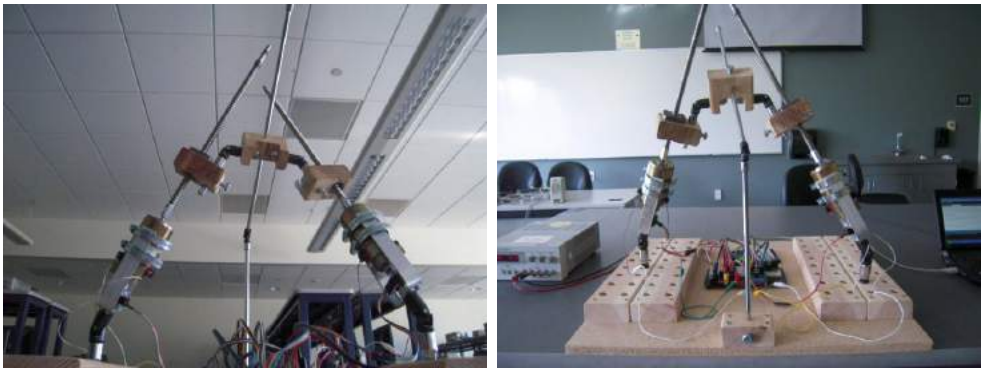


Figure 3.14: *2-DOF proof-of-concept model demonstrating motion at low altitude angles.*

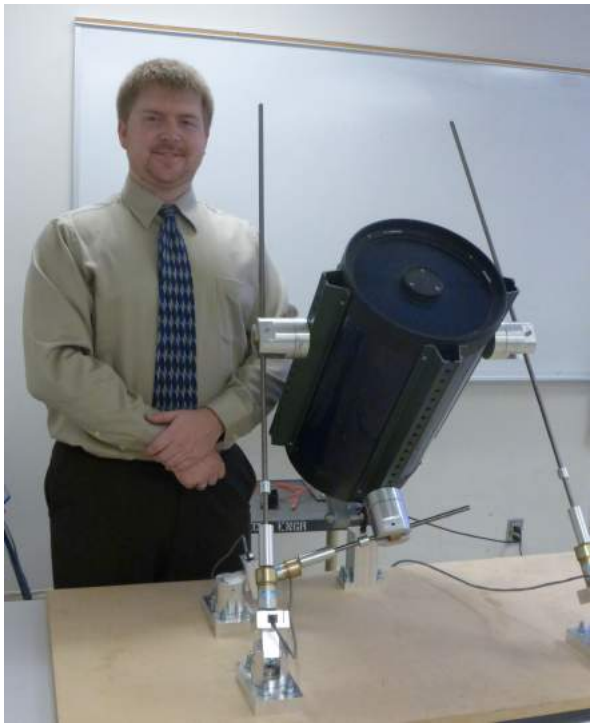


Figure 4.1: *3-DOF parallel-actuator telescope mount prototype. 10" LX200-Meade OTA provided by Dr. Russell Genet, Cal Poly SLO.*

4.1 Mechanical Components

In order to diminish destructive alterations to the provided OTA, attach points for the purpose of this thesis were designed to be minimally invasive. A rail system was designed to allow for adjustable attach points for the linear actuators without further modifications to the OTA as shown in Figure 4.3. The Pittman motor, model number GM8723J133-R1, was selected to actuate the OTA mount due that motor's availability and torque provided by its 31:1 gear ratio. This model of motor was used in each of three linear actuator assemblies. Motor assemblies were designed to hold the brass cased motors rigidly with the motor output shaft extending through the point-of-rotation at the base of the assembly. These motor assemblies connect to

Figure 4.2: *Mount components manufactured by Garrett Gudgel and Cal Poly Shop Tech David Schaeffer.*

ground as shown in Figure 4.4.

The spherical joint that allows the OTA to rotate was fixed to the rear plate outside the OTA cross section thereby allowing sufficient space for a rear mounted camera as shown in Figure 4.5. Tests defined in Section 3.3 were performed while adjusting the linear actuator connections along the rails until a configuration was deemed capable of actuating the OTA through the desired celestial field of view.

All part drawings are shown in Appendix A.1.

4.2 Electronics

The OTA mount is actuated by three DC motors equipped with quadrature encoders. Circuit design needed to support three motors, three encoders, and provide other sensor inputs, e.g., electrical current feedback. The supplied motors will draw current at a rate of less than $1A$ per motor. I specified each motor driver circuit to be capable of supplying $30A$ thereby allowing the controller to be used in systems requiring larger motors.

Preliminary circuit designs were tested through the creation of individual boards for each major integrated circuit component as shown in Figure 4.6. This ensured

Figure 4.3: *OTA (pictured upside down) with rotation assemblies adjustable along positioning rails. Threaded rods (not shown) extend through the rotation assemblies.*

Figure 4.4: *Motor assembly connecting the ground to the threaded rod, three identical assemblies were used in the OTA mount system.*

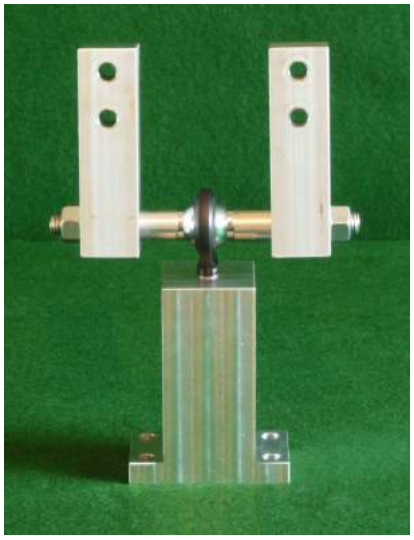


Figure 4.5: *OTA base assembly connecting the ground to the OTA through a spherical joint. Raised arms attach to the OTA rear-plate.*

that integrated circuits and passive units were capable of driving each motor/encoder unit before being applied in the final design of the triple motor/encoder driver circuit shown in Figures 4.8 and 4.9. Integrated circuits include three DC drive units, three encoder counter circuits, and a single four-channel analog-to-digital converter for sensing current supplied by the DC drivers that enable detection of motor stall conditions.

Initially the project had been intended to operate astronomy calculations and motor control from a single computer unit, i.e., a Raspberry Pi B+ noted in Figure 4.7. Preliminary tests showed the Raspberry Pi B+, an off-the-shelf Linux OS computer with GPIO capability, capable of operating the motors and encoders in closed-loop control. However, the addition of astronomy calculations and user interface to that same closed-loop motor control proved non-deterministic as motors would occasionally be erroneously driven during bench testing. Instead I decided to make a dedicated motor controller capable of communicating with a computer through USB where the user interface, astronomy calculations, and OTA directional path planning could be conducted. I determined the Mbed LPC1768 Prototype Board was capable of meeting my needs, including being off-the-shelf. I selected the board for its C++ programming, its 100MHz processing speed, numerous number of digial I/O pins to accomodate motor and encoder integrated, and its ability to communicate with

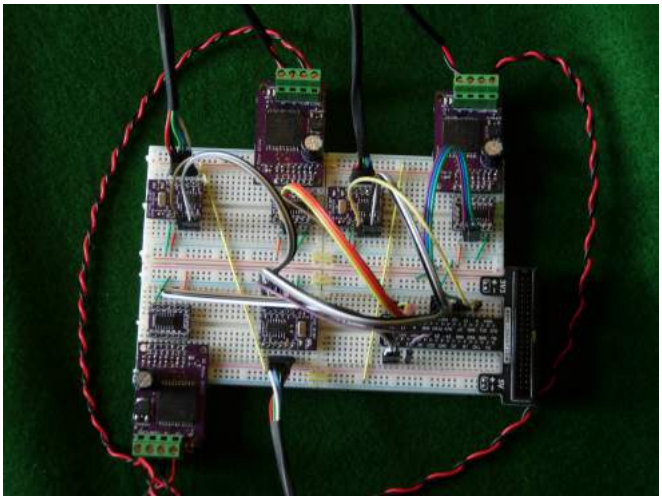


Figure 4.6: *Circuit boards designed to test feasibility of component use.*

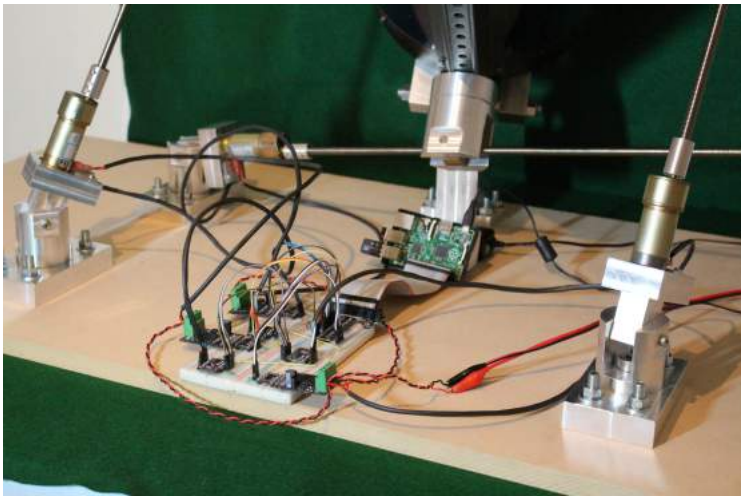


Figure 4.7: *Initial system build and test circuits controlled through a Rasperberry Pi B+.*

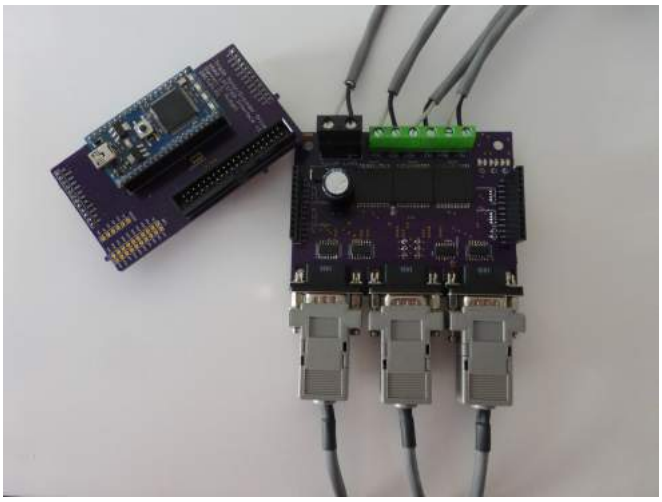


Figure 4.8: *Top-view of Mbed controller shield and triple motor/encoder driver board.*

the controlling computer via USB, SPI, and/or I2C communication protocols. The Mbed controller board was designed into a custom shield board that attaches to the motor/encoder drive board thereby allowing user-flexibility in selecting different controller boards. The completed mbed shield board and motor/encoder driving board are shown in Figure 4.8.

Electronic schematics and board design are shown in Appendix A.2.

4.3 Firmware and Software

The Mbed motor/encoder controller is programmed using C++. Closed-control proportional-integration-derivative loops for the motors are performed at $100Hz$ and the remaining idle time is dedicated to communications with the controlling computer. Computer-to-Mbed commands consist of calibrating the motors, sending the motors to desired positions, and turning off the motors. Motor calibration consists of driving all motors at a constant voltage in the negative direction until all threaded rods have reached a hard stop. This hard stop can be detected through a combination of amperage spikes and cessation of motor shaft movement.

The computer portion of software is a combination of user interface and background calculations programmed in Python. For testing, the user interface depicted

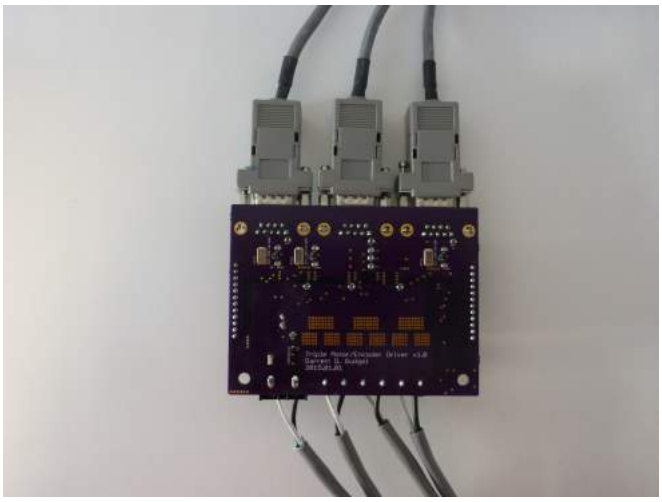


Figure 4.9: *Bottom-view of triple motor/encoder driver board.*

in Figure 4.10 only supports operations for calibrating the telescope, moving the OTA mount to desired θ_1, θ_2, θ_3 locations, and emergency shutoff of the motors. Behind the user interface, Python programming interprets user commands and communicates with the motor controller at an update frequency of $10Hz$ to ensure smooth motion and constant feedback variable updates on the user interface. When a new OTA mount location is chosen, the software plans the path to that mount location by a series of 1° incremental waypoints in the target direction. When feedback from the controller indicates the OTA mount has reached its target location it will hold position until a new target has been defined and the process is repeated.

See Appendix B for all code.

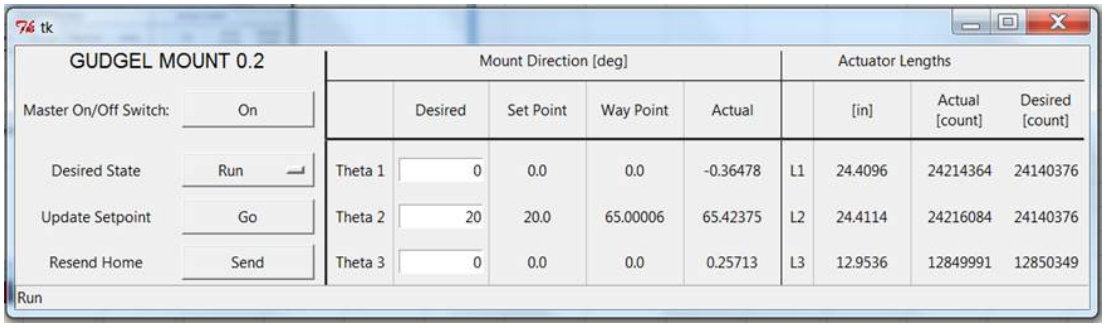


Figure 4.10: *User interface for limited OTA mount control.*

4.4 Auxiliary Accessory: Focusing Mirror Controller

The provided OTA has a stepper motor built-in to focus the primary mirror. While functionality of this component was not necessary for the purpose of this thesis, a simple stepper-motor controller was designed and built, see Figure 4.11, for easier operation of the OTA in the future.

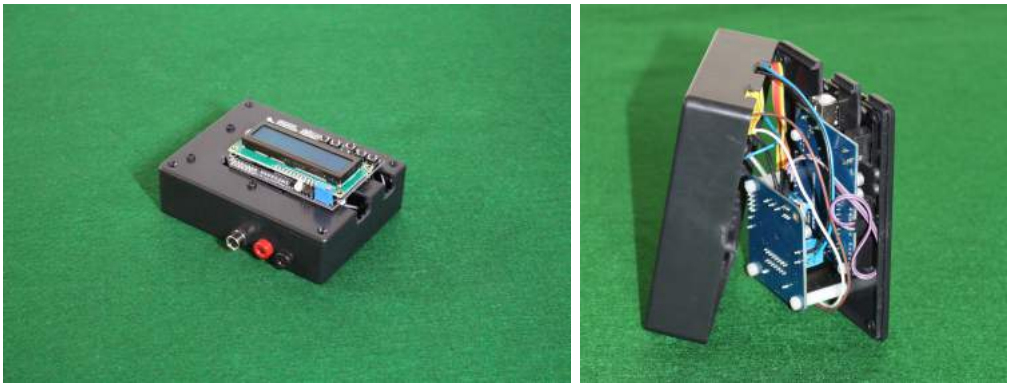


Figure 4.11: *Stepper motor controller comprised of an Arduino Uno and a L298N bridge driver.*

CHAPTER 5: PROTOTYPE TESTING AND RESULTS

5.1 Verification Testing and Results

To test the OTA mount's motion reliability and rotation accuracy, five sets of nine-dot arrays were placed on a wall $600cm$ from the OTA point-of-rotation; see Figure 5.1. Each nine-dot array was configured in a square such that each dot was spaced $1cm$ from adjacent vertical and horizontal dots. The five sets were configured in a rectangular formation with outer dimensions of $50cm$ wide and $41.5cm$ tall with the edge of the bottom row $160cm$ above the ground. The dimensions of the nine-dot arrays were arbitrary but knowing the spacing enabled accurate analysis of OTA images. The images for analyzing OTA motions were obtained from two cameras fixed to the OTA; one camera on the rear mount and the other camera on the right side rail mount as shown in Figure 5.2.

5.1.1 *Slew Rate*

A benchtop test of the listed motors demonstrated the linear actuator lead-screws turn at a rate of approximately two revolutions per second when using a $12V$ power supply. The threading of the lead-screws is 1/16" pitch, therefore, two revolutions per second convert to $8s$ per inch of linear motion. In this parallel design, linear speed doesn't translate directly into rotational speed of the telescope. The average slew rate can be determined by measuring the time elapsed to redirect the OTA between extreme locations of the field-of-view. For example, rotating the telescope from 30° tilt to 110° tilt (20° past zenith) requires approximately thirty inches of motion from linear actuators #1 and #2. At a rate 8 seconds per inch over 30 inches of motion, the OTA will take 240 seconds to rotate 80°. This equates to an approximate rate-of-motion of 6 seconds per degree. Due to slowness, the test with the OTA setup was instead performed using a $15V$ power supply to increase motor speed. Results are shown in Table 5.1. Slew rates can be increased further by increasing the power supply up to $24V$ or switching to motors capable of higher torque and speed.

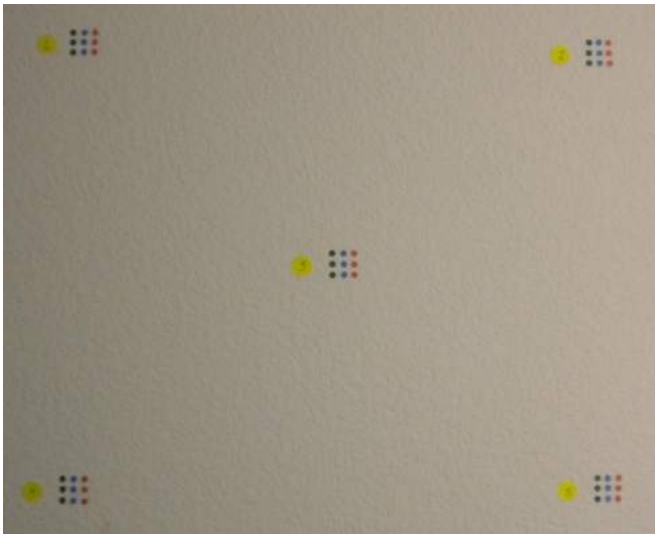


Figure 5.1: *Five sets of nine-dot arrays for testing motion capabilities.*

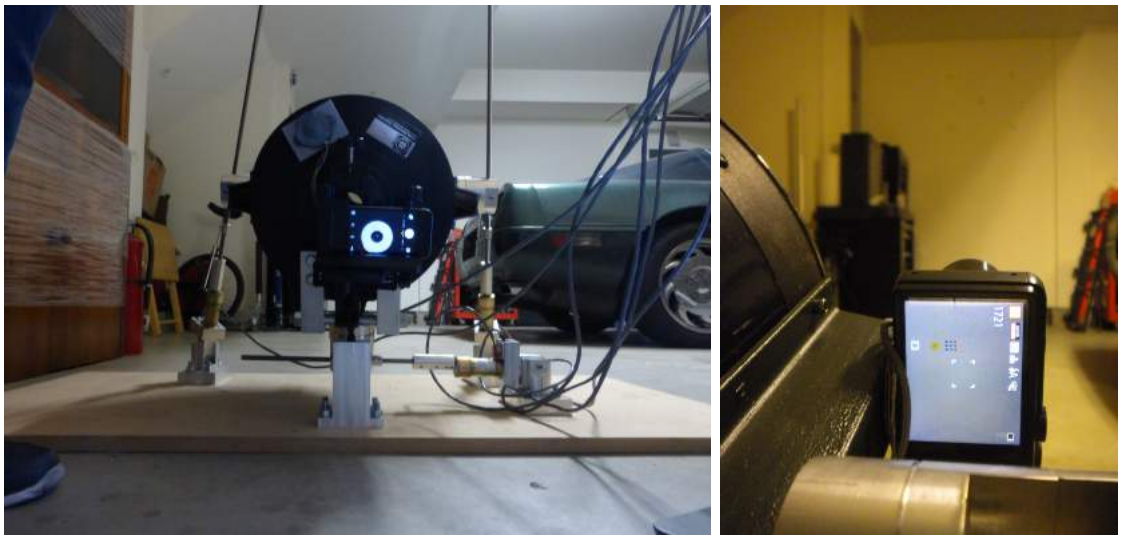


Figure 5.2: *Rear mount camera looking through telescope (no magnification) and right hand rail mounted camera (20x magnification).*

Table 5.1: *Slew rate test results using* $15V$ *power supply.*

Start Point [deg]	**End Point [deg]**	**Time [s]**	**Average Slew Rate [deg/s]**
20°	80°	89	$0.67\frac{deg}{s}$
80°	20°	90	$0.67\frac{deg}{s}$
20°	80°	89	$0.67\frac{deg}{s}$
80°	20°	90	$0.67\frac{deg}{s}$
20°	80°	89	$0.67\frac{deg}{s}$
80°	20°	90	$0.67\frac{deg}{s}$
20°	50°	50	$0.60\frac{deg}{s}$
50°	80°	39	$0.77\frac{deg}{s}$
80°	50°	41	$0.73\frac{deg}{s}$
50°	20°	49	$0.61\frac{deg}{s}$

5.1.2 *Repeatability*

To test the directional repeatability of the mount, the OTA was directed to each of the dot arrays using $\theta_3 = 0$. Once the cameras were centered on each dot array the current direction angles, θ_1 and θ_2, were noted and OTA images were obtained. After images were obtained of all points, the mount was again directed in turn to each set of dot array angles obtaining a second set of images. From the images a relative distance can be measured between center dots. Note the direct distance from OTA to the center dot-array, taking into account OTA height for the vertical component:

$$D_{OTA-dot} = \sqrt{(600cm)^2 + (160cm - 16cm)^2} = 617cm$$

The relative error in linear distance can be converted to an angular error through a sine inverse calulation. The results of this test are indicated in Table 5.2.

5.1.3 *Point Rotation*

To test the rotational accuracy of the mount, the OTA was directed to the center dot array. Keeping θ_1 and θ_2 fixed for the remainder of the test, OTA images were

taken in the following order: $\theta_3 = 0°$, $\theta_3 = 5°$, $\theta_3 = 0°$, $\theta_3 = -5°$, $\theta_3 = 0°$. The results of this test are indicated in Table 5.3.

5.2 Investigation

In a research-grade telescope, the results from Section 5.1 would be unacceptable. After locking all motor positions to supposedly constrain all three DOF, applying side loads to the telescope demonstrated the extent of backlash in the system as seen in Figure 5.3. Inspection of all joints revealed a significant amount of backlash in the third OTA rotator joint as seen in Figure 5.4. It is possible that similar backlash would have been seen in the first and second OTA rotator joints as well had the telescope not been in this low altitude position where gravity is holding the joint in place. It is possible that connections between the motors and threaded rods were not rigid. Apparently minimal, there still appears to be a small amount of deflection at the motor shaft as seen Figure 5.5. A full-scale design of support and bearings between the motor output shaft and threaded rod needs to ensure a rigid connection.

5.3 Summary

The three DOF OTA mounting assembly (machine system) did perform the requested tasks directing the OTA. However, the system results were not as precise as I had thought the design should have been able to support. This lack of desired precision I attribute to the compounding of large tolerances within my manufacturing processes. This compounding of large tolerances resulted in my machine not locating with sufficient precision a targeted celestial point consistently. In reviewing the machine design, software engineering, and known astronomical calculations, I believe that a better manufacturing process with closer tolerances will improve the machines performance and prove the system design meets the goals for its use in the astronomy community.

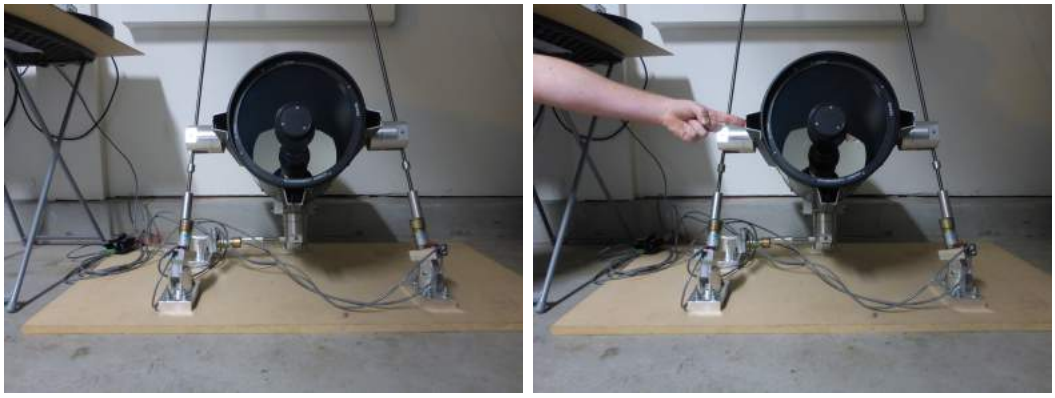


Figure 5.3: *Telescope can be pushed several degrees in what appears to be a single line of motion.*

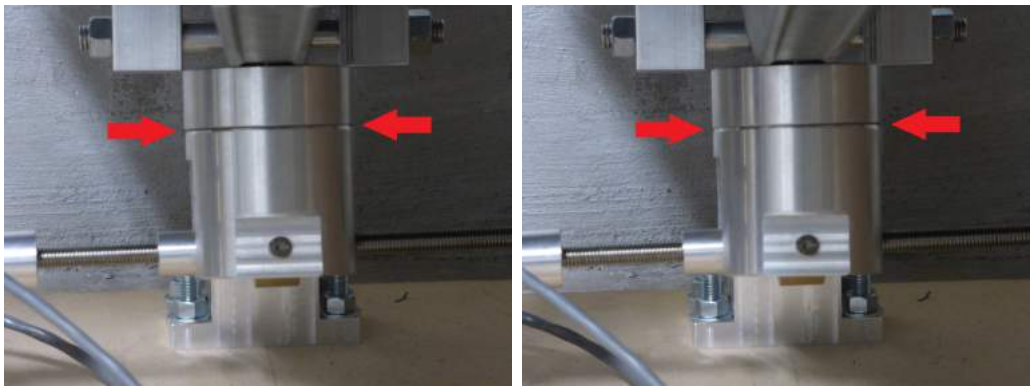


Figure 5.4: *Linear actuator #3's OTA rotator joint has visible mechanical backlash when external load is applied to telescope mount.*

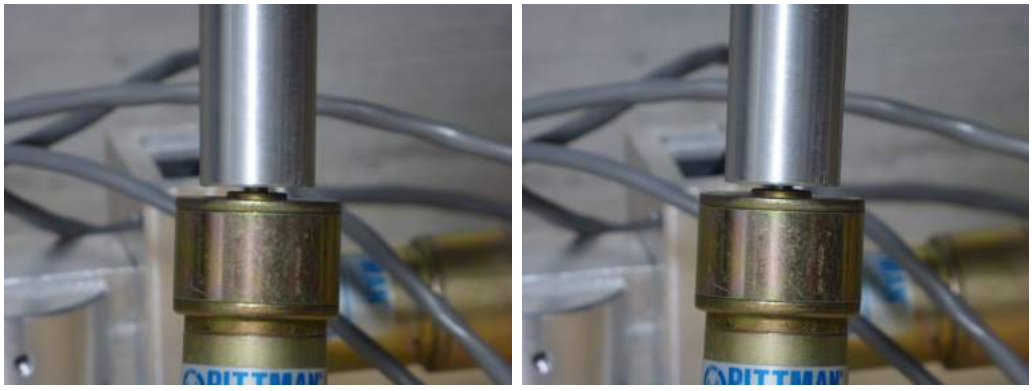


Figure 5.5: *Motor output shaft capable of small undesirable deflections.*

Table 5.2: *Repeatability test results.*

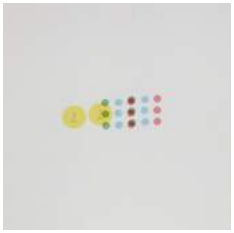
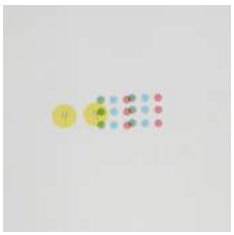
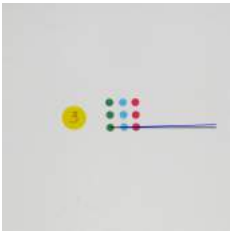
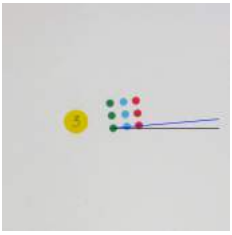
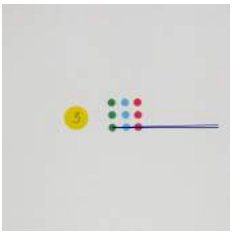
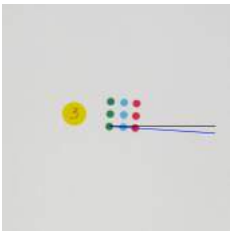
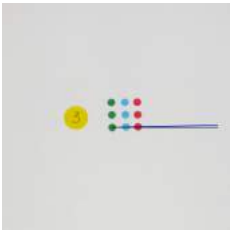
Point	Error Distance [cm]	Error Angle [deg]	Image
1	2.2	0.20°	
2	0.5	0.05°	
3	0.4	0.04°	
4	2.8	0.26°	
5	1.7	0.16°	

Table 5.3: *Point rotation test results.*

Point	Desired Angle	Actual	Normalized	Image
1	0°	1.6°	0°	
2	5°	4.8°	3.2°	
3	0°	1.4°	−0.2°	
4	−5°	−3.8°	−5.4°	
5	0°	1.1°	−0.5°	

BIBLIOGRAPHY

[1] *FAQ VLT/Paranal*. European Southern Observatory. <https://www.eso.org/public/about-eso/faq/faq-vlt-paranal/>.

[2] *Baker-Nunn Camera 001*. June 2009. <https://commons.wikimedia.org/wiki/File:Baker-Nunn_camera_001.JPG>.

[3] *Buckled column*. August 2009. <https://commons.wikimedia.org/wiki/File:Buckled_column.svg>.

[4] *Coordinate Systems*. University of Michigan Astronomy Department, December 2011. <https://dept.astro.lsa.umich.edu/ugactivities/Labs/coords/>.

[5] *Sidereal Time and Hour Angle Demonstrator*. September 2011. <http://astro.unl.edu/classaction/animations/200level/siderealTimeAndHourAngleDemo.html>.

[6] *RA and Dec on Celestial Sphere*. June 2012. <https://commons.wikimedia.org/wiki/File:Ra_and_dec_on_celestial_sphere.png>.

[7] *Big 3.5m Telescope*. AFRL, Kirtland Air Force Base, August 2014. <https://commons.wikimedia.org/wiki/File:Big3_5mtele.png>.

[8] R. G. Budynas and K. J. Nisbett. *Shigley's Mechanical Engineering Design With Additional Materials*. McGraw-Hill, 9th edition, 2011.

[9] K. Burnett. *Converting RA and DEC to ALT and AZ*. May 1998. <http://www.stargazing.net/kepler/altaz.html>.

[10] T. W. Carlson. *Azimuth-Altitude schematic*. May 2012. <https://commons.wikimedia.org/wiki/File:Azimuth-Altitude_schematic.svg>.

[11] B. Hartt, B. Gilchrist, and V. Truman. Hexapod linear actuators. Senior project, California Polytechnic State University, San Luis Obispo, 2012.

[12] R. C. Hibbeler. *Engineering Mechanics: Statics*. Prentice Hall, Upper Saddle River, NJ, 12th edition, 2010.

[13] K. Lancaster. *AMiBA 2*. June 2006. <https://en.wikipedia.org/wiki/File:AMiBA_2.jpg>.

[14] S. B. Niku. *Introduction to Robotics*. Wiley, Hoboken, N.J., 2nd edition, 2011.

[15] P. Rucks. *Stuetzmontierung*. 1990. <https://commons.wikimedia.org/wiki/File:Stuetzmontierung.jpg>.

[16] K. Russell, Q. Shen, and R. S. Sodhi. *Mechanism Design: Visual and Programmable Approaches*. CRC Press, 2014.

[17] V. S. Soni. *Mechanics and Relativity*. PHI Learning, 2nd edition, 2011.

[18] O. Trenouth. *Star Trails Over Umbria*. September 2011. <http://olitee.com/2011/09/star-trails-over-umbria/>.

[19] M. Yousry. *mbed LPC1768 Eagle library*. February 2012. <https://developer.mbed.org/users/MadVoltage/notebook/mbed-lpc1768-eagle-library/>.

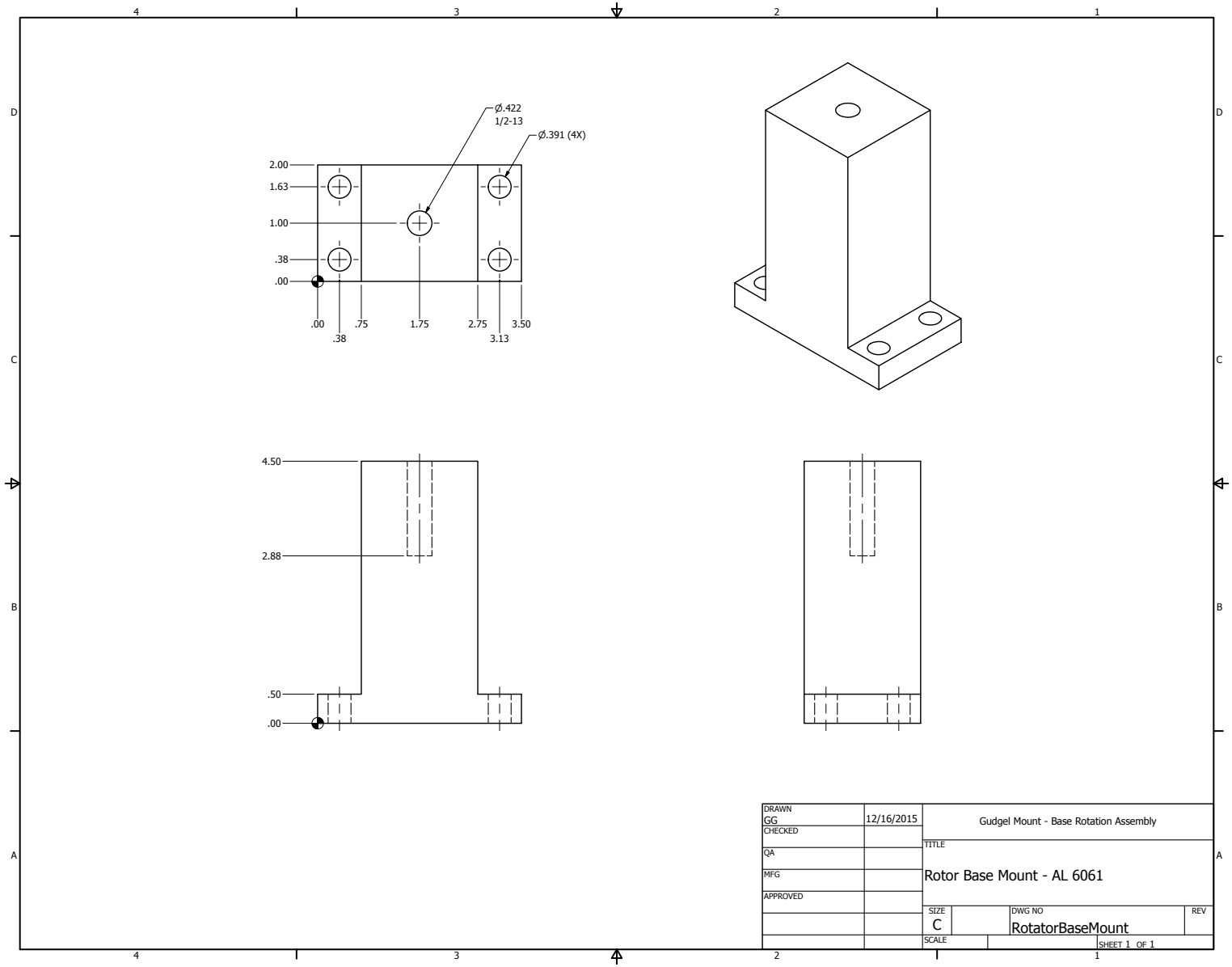
A.1 Mechanical

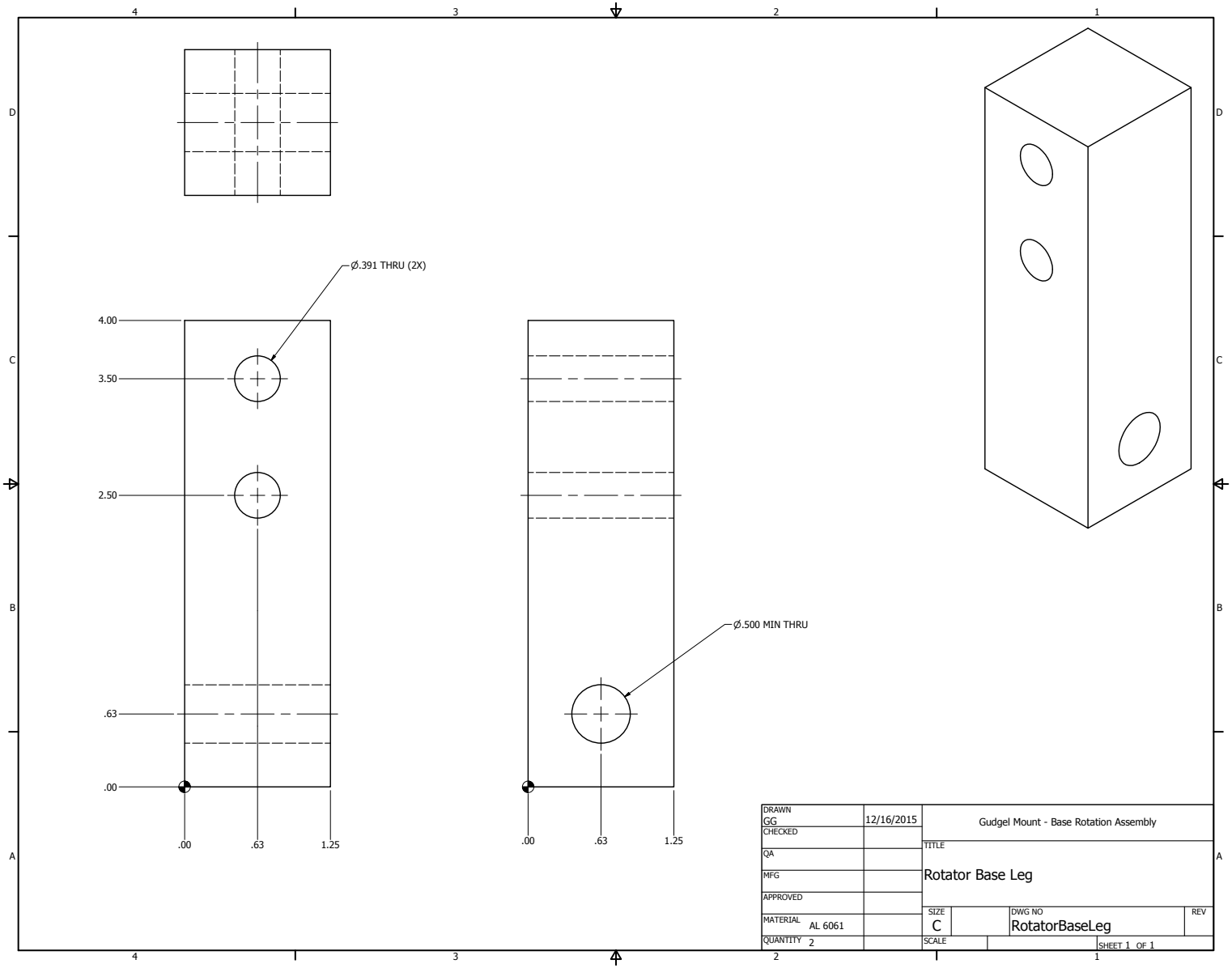
A.1.1 *OTA Base Rotator Assembly*

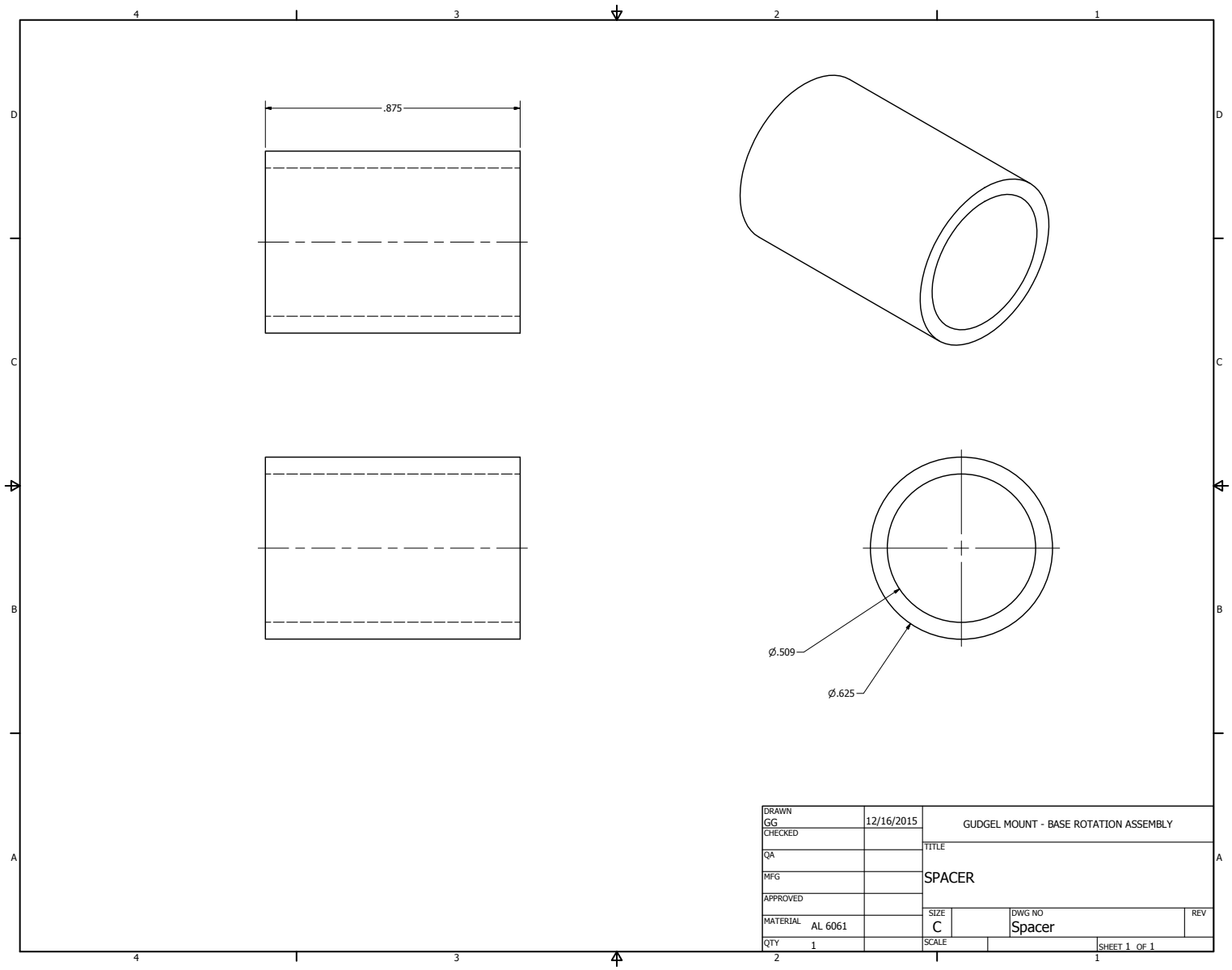
Figure A.1: *Ground to OTA base rotator assembly.*

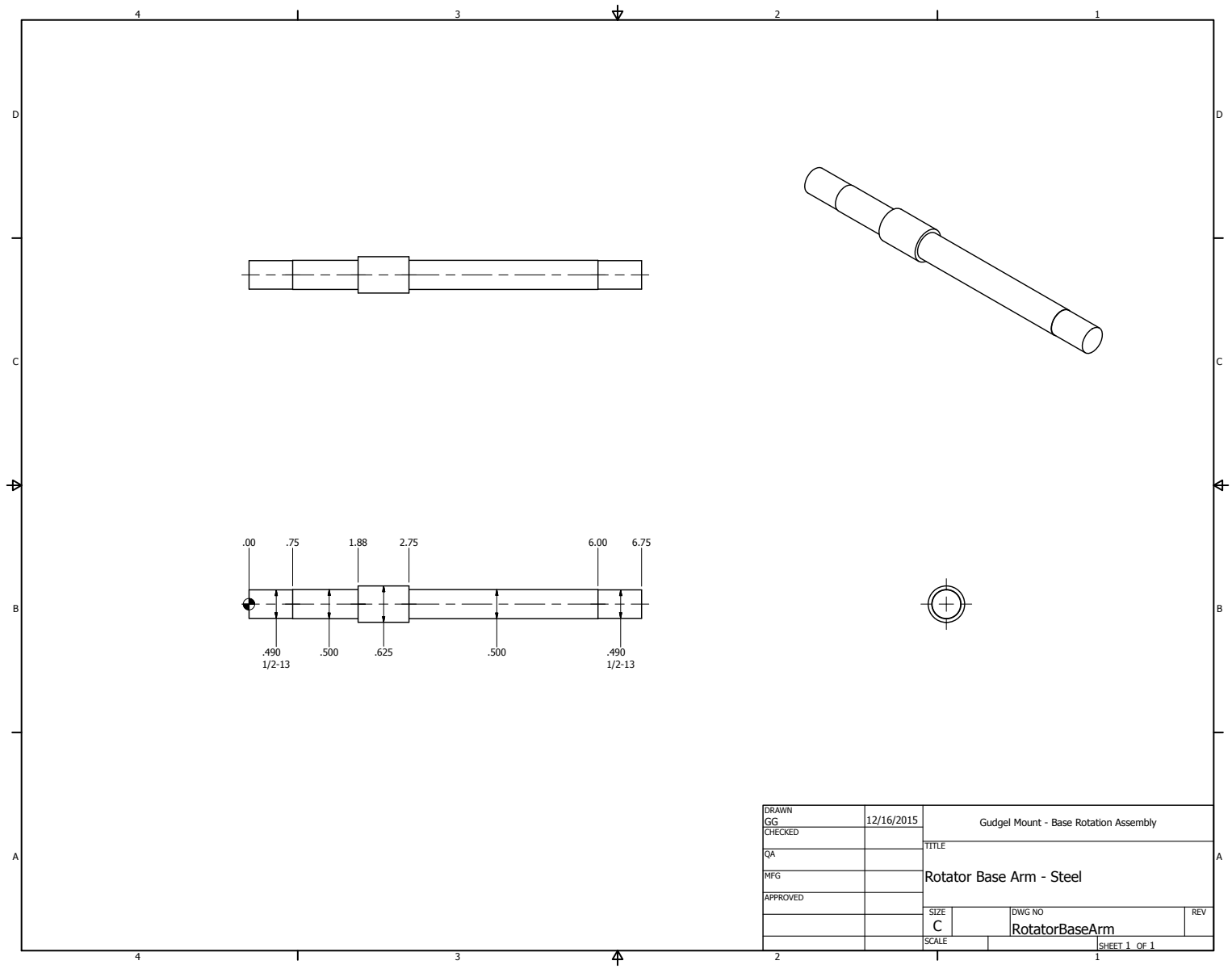
Table A.1: *Part list: Ground to OTA base rotator assembly.*

Part Name	**Quantity**
Rotor Base Mount	1
Ball Joint Rod End 6960T11	1
Rotator Base Leg	2
Spacer	1
Rotator Base Arm	1


Ø.422
1/2-13
Ø.391 (4X)
2.00
1.63
1.00
.38
.00
.00
.38
.75
1.75
2.75
3.13
3.50
4.50
2.88
.50
.00
DRAWN
GG
12/16/2015
CHECKED
QA
MFG
APPROVED
Gudgel Mount - Base Rotation Assembly
TITLE
Rotor Base Mount - AL 6061
SIZE
C
DWG NO
RotatorBaseMount
REV
SCALE
SHEET 1 OF 1


Ø.391 THRU (2X)
4.00
3.50
2.50
.63
.00
.00
.63
1.25
Ø.500 MIN THRU
.00
.63
1.25
DRAWN
GG
12/16/2015
CHECKED
QA
MFG
APPROVED
MATERIAL
AL 6061
QUANTITY
2
Gudgel Mount - Base Rotation Assembly
TITLE
Rotator Base Leg
SIZE
C
DWG NO
RotatorBaseLeg
REV
SCALE
SHEET 1 OF 1


.875
Ø.509
Ø.625
DRAWN
GG
12/16/2015
CHECKED
QA
MFG
APPROVED
MATERIAL
AL 6061
QTY
1
GUDGEL MOUNT - BASE ROTATION ASSEMBLY
TITLE
SPACER
SIZE
C
DWG NO
Spacer
REV
SCALE
SHEET 1 OF 1



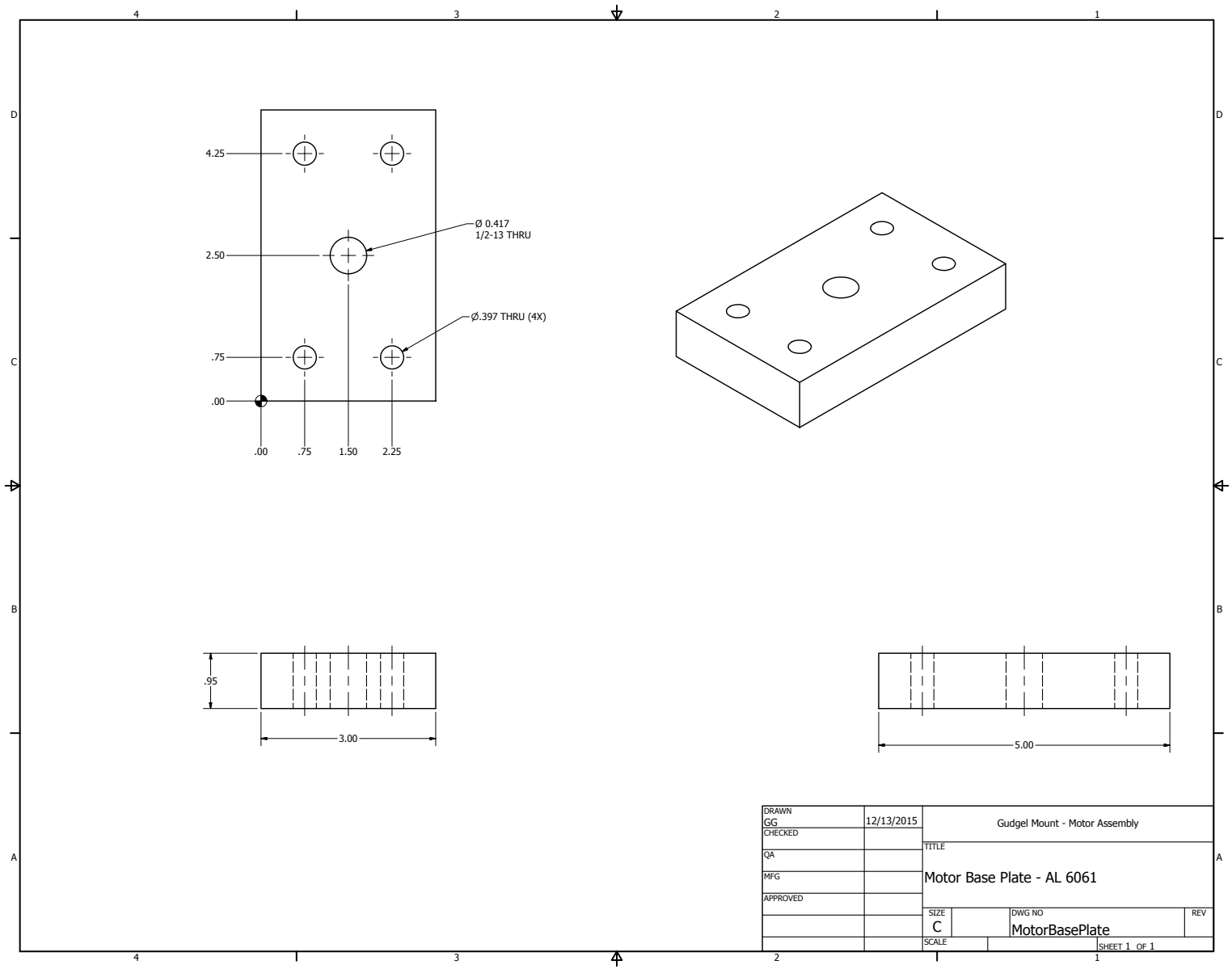
DRAWN GG	12/16/2015	Gudgel Mount - Base Rotation Assembly	
CHECKED		TITLE	
QA		Rotator Base Arm - Steel	
MFG			
APPROVED		SIZE C	DWG NO RotatorBaseArm / REV
		SCALE	SHEET 1 OF 1

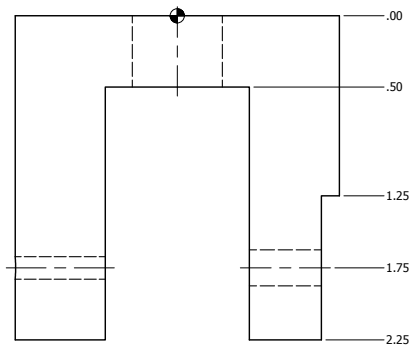
A.1.2 *Actuator Base Rotator Assembly*

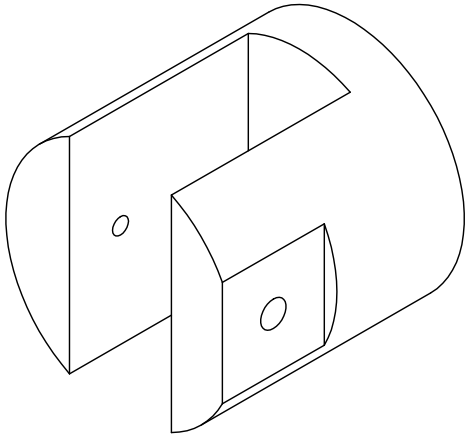
Figure A.2: *Ground to actuator base rotator assembly.*

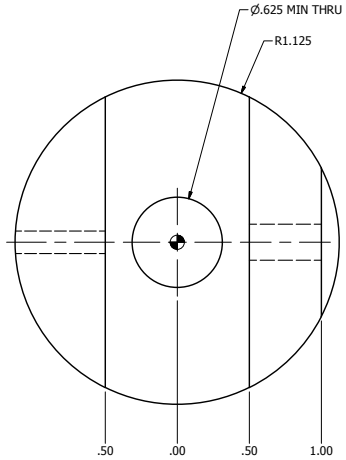
Table A.2: *Part list: Ground to actuator base rotator assembly.*

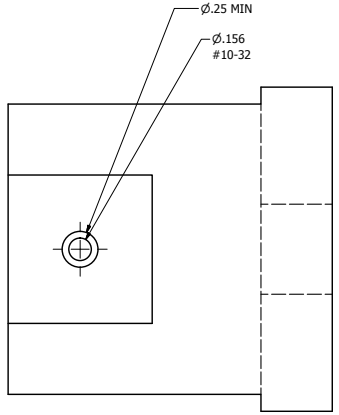
Part Name	**Quantity**
Motor Base Plate	1
Motor Rotator	1
Shoulder Screw 91327A304	1
Motor Hinge	1
Shoulder Screw 93996A540	1
Motor Casing	1
Motor Hold	1
Motor Shaft Coupler	1


4.25
2.50
.75
.00
Ø 0.417
1/2-13 THRU
Ø.397 THRU (4X)
.00
.75
1.50
2.25
.95
3.00
5.00
DRAWN
GG
12/13/2015
CHECKED
QA
MFG
APPROVED
Gudgel Mount - Motor Assembly
TITLE
Motor Base Plate - AL 6061
SIZE
C
DWG NO
MotorBasePlate
REV
SCALE
SHEET 1 OF 1

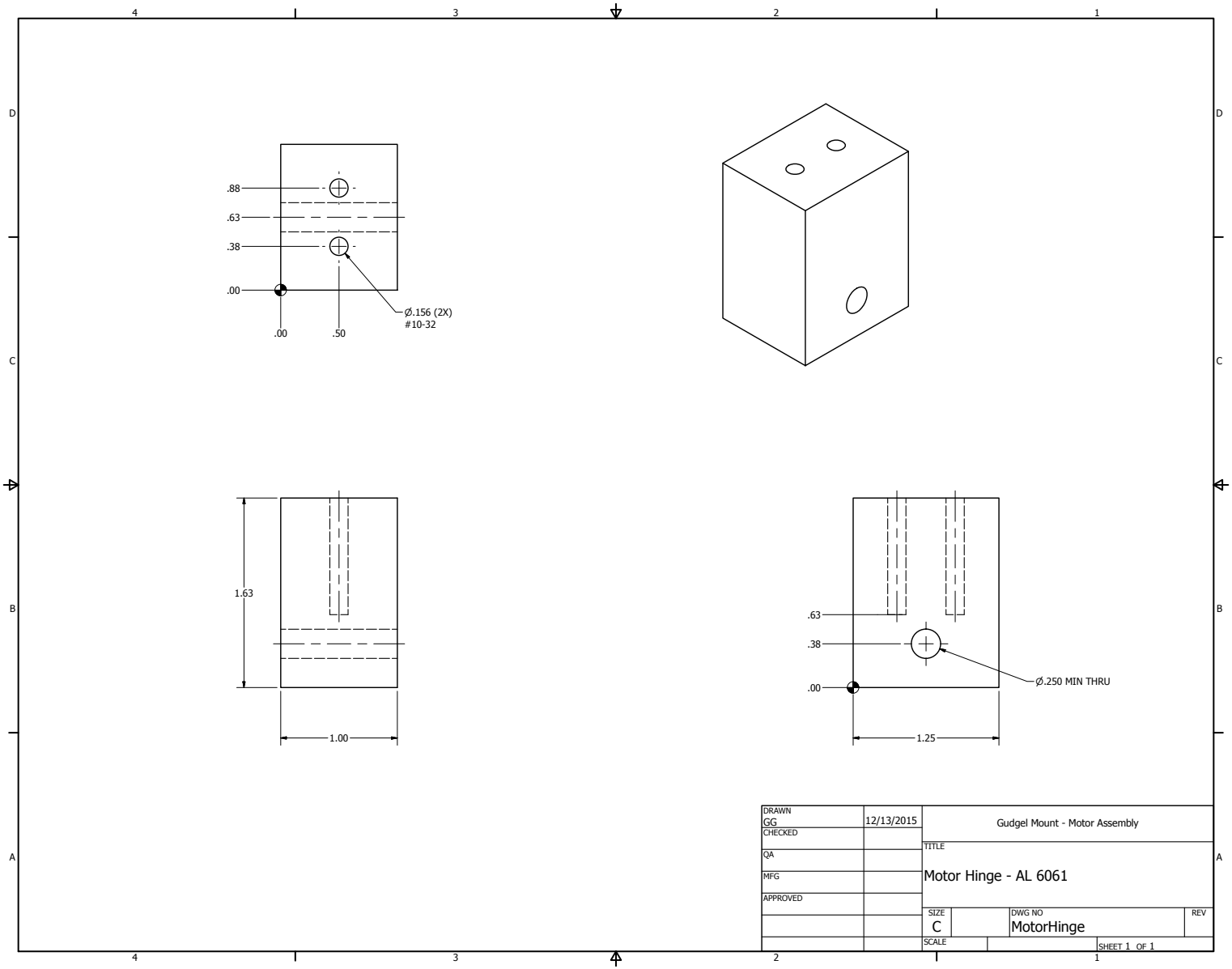


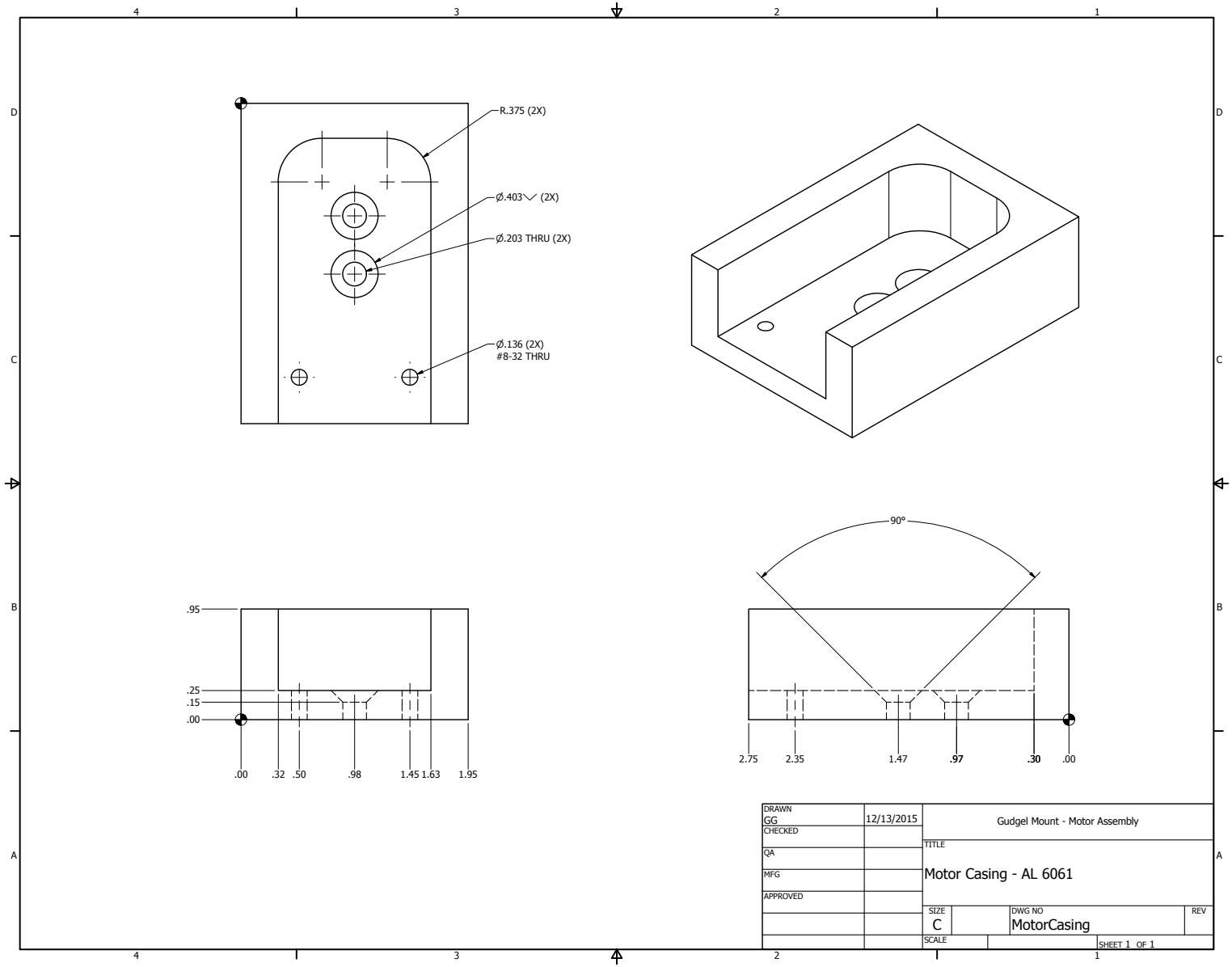


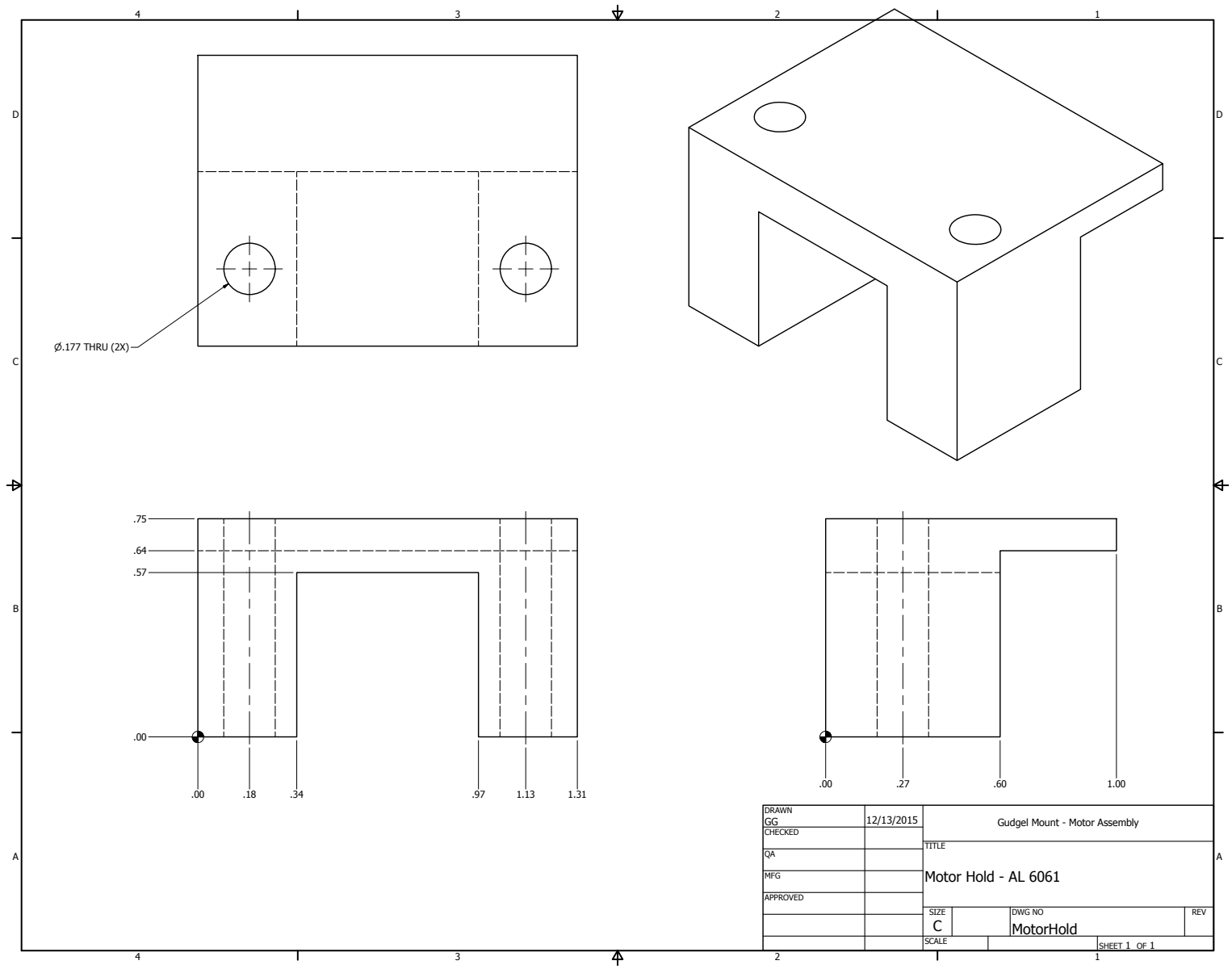


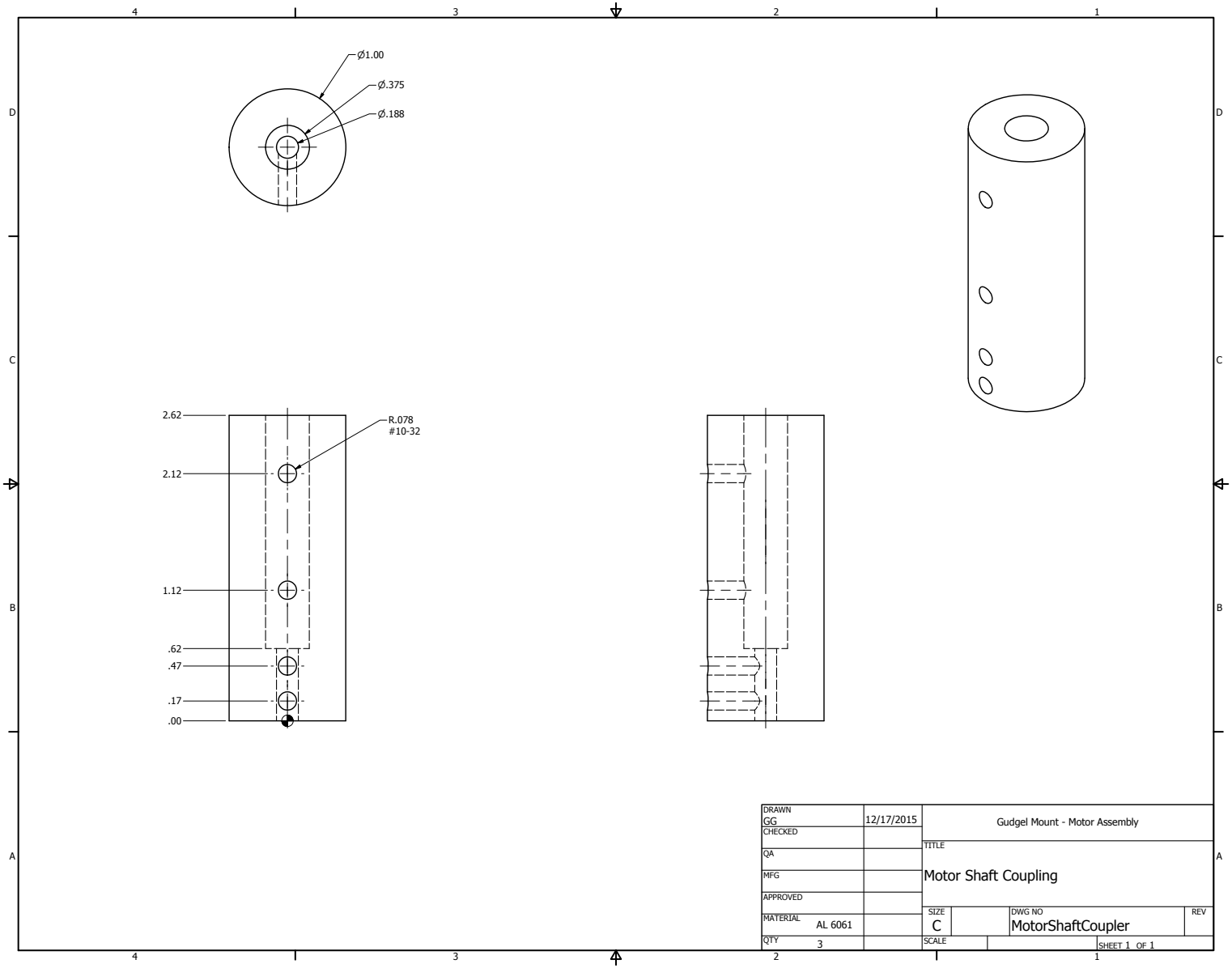


DRAWN GG	12/13/2015	Gudgel Mount - Motor Rotator	
CHECKED		TITLE	
QA		Motor Rotator - AL 6061	
MFG			
APPROVED		SIZE C	DWG NO MotorRotator REV
		SCALE	SHEET 1 OF 1


.88
.63
.38
.00
.00
.50
Ø.156 (2X)
#10-32
1.63
1.00
.63
.38
.00
Ø.250 MIN THRU
1.25
DRAWN
GG
12/13/2015
CHECKED
QA
MFG
APPROVED
Gudgel Mount - Motor Assembly
TITLE
Motor Hinge - AL 6061
SIZE
C
DWG NO
MotorHinge
REV
SCALE
SHEET 1 OF 1


R.375 (2X)
Ø.403 (2X)
Ø.203 THRU (2X)
Ø.136 (2X)
#8-32 THRU
90°
.95
.25
.15
.00
.00 .32 .50 .98 1.45 1.63 1.95
2.75 2.35 1.47 .97 .30 .00
DRAWN
GG
12/13/2015
CHECKED
QA
MFG
APPROVED
Gudgel Mount - Motor Assembly
TITLE
Motor Casing - AL 6061
SIZE
C
DWG NO
MotorCasing
REV
SCALE
SHEET 1 OF 1


Ø.177 THRU (2X)
.75
.64
.57
.00
.00
.18
.34
.97
1.13
1.31
.00
.27
.60
1.00
DRAWN
GG
12/13/2015
CHECKED
QA
MFG
APPROVED
Gudgel Mount - Motor Assembly
TITLE
Motor Hold - AL 6061
SIZE
C
DWG NO
MotorHold
REV
SCALE
SHEET 1 OF 1


Ø1.00
Ø.375
Ø.188
2.62
2.12
1.12
.62
.47
.17
.00
R.078
#10-32
DRAWN
GG
12/17/2015
CHECKED
QA
MFG
APPROVED
MATERIAL
AL 6061
QTY
3
Gudgel Mount - Motor Assembly
TITLE
Motor Shaft Coupling
SIZE
C
DWG NO
MotorShaftCoupler
REV
SCALE
SHEET 1 OF 1

A.1.3 *Actuator to OTA Rotator Assembly*

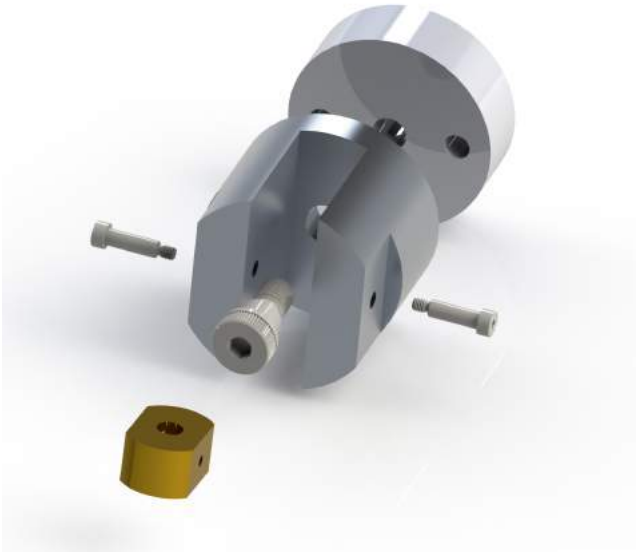
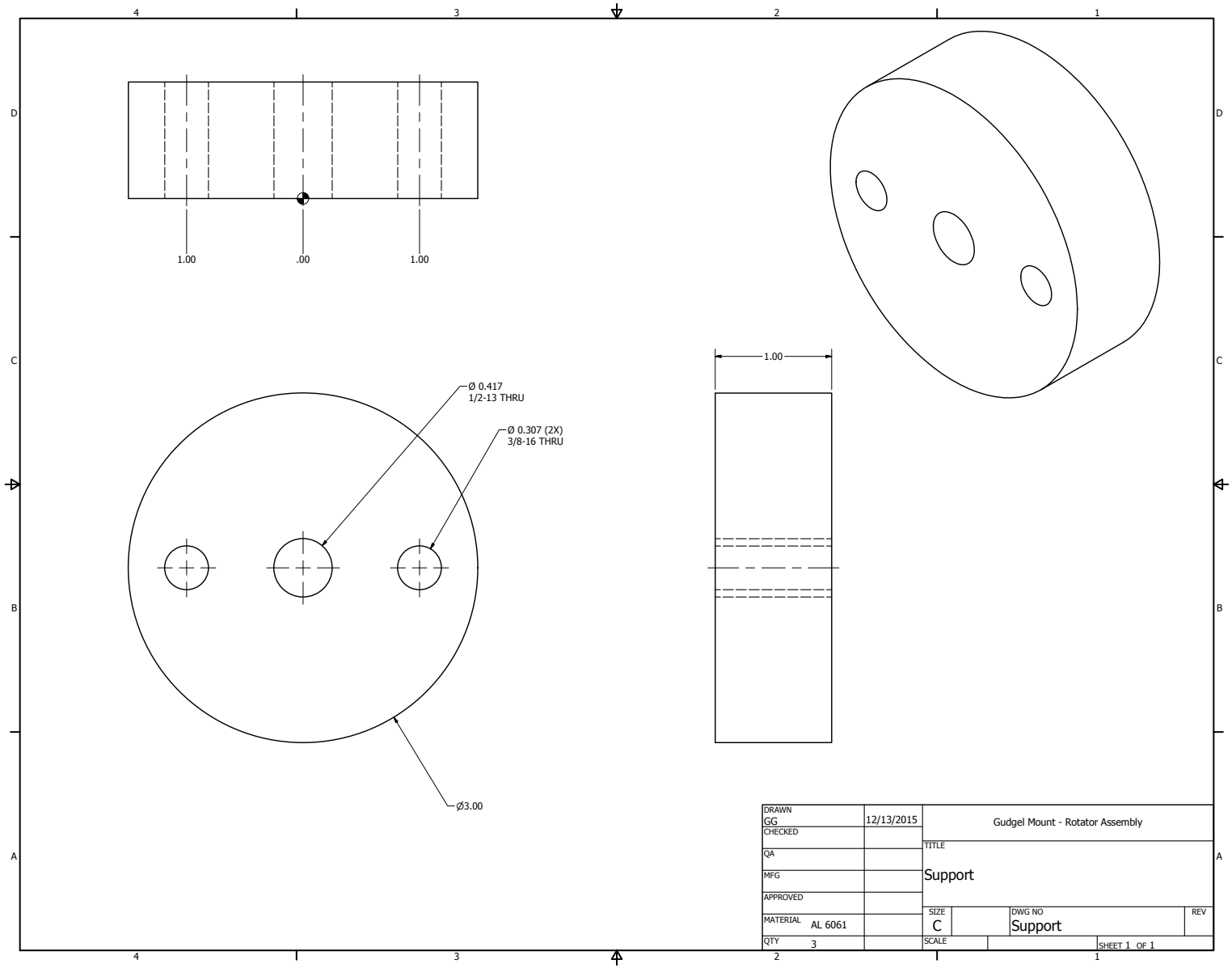
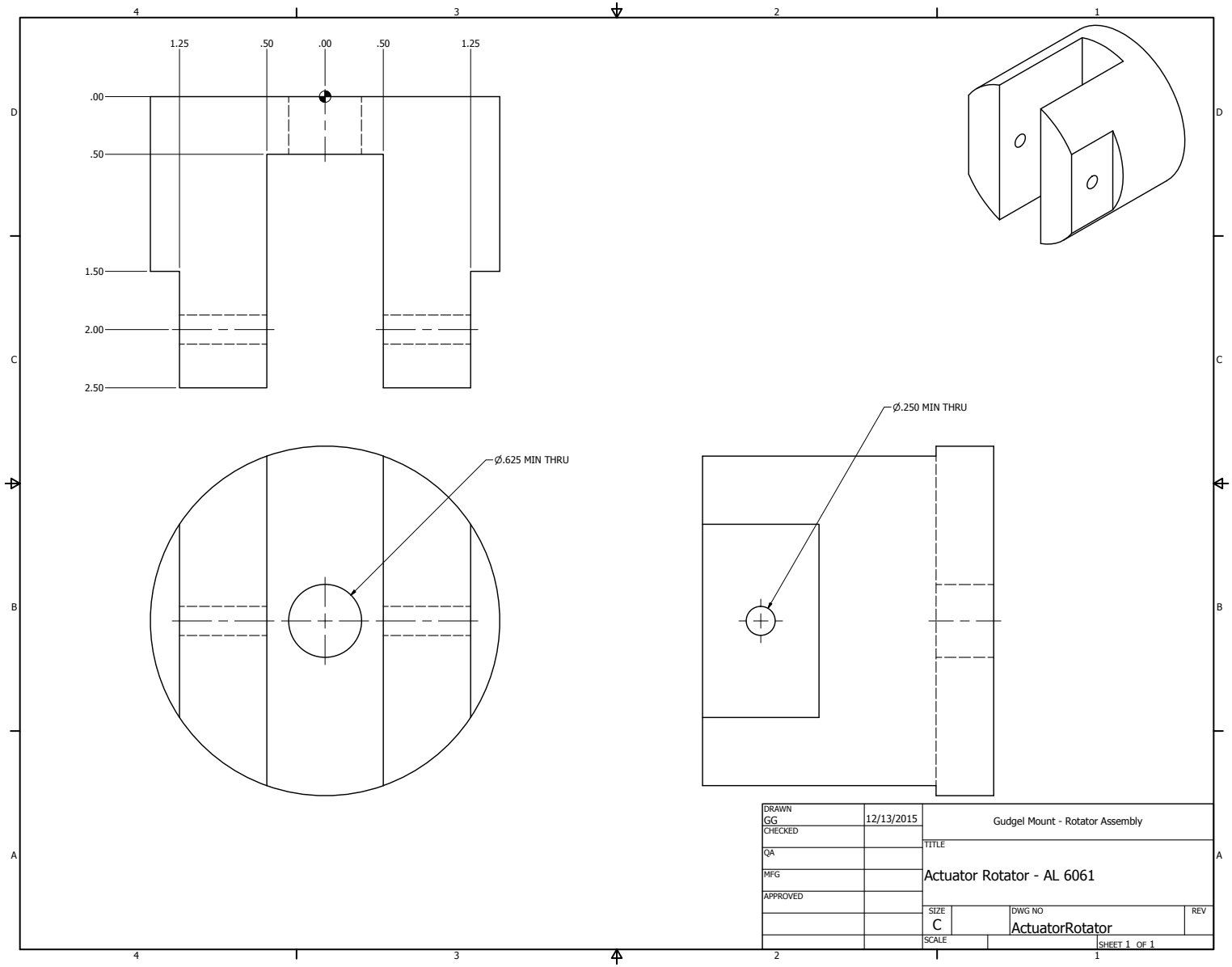


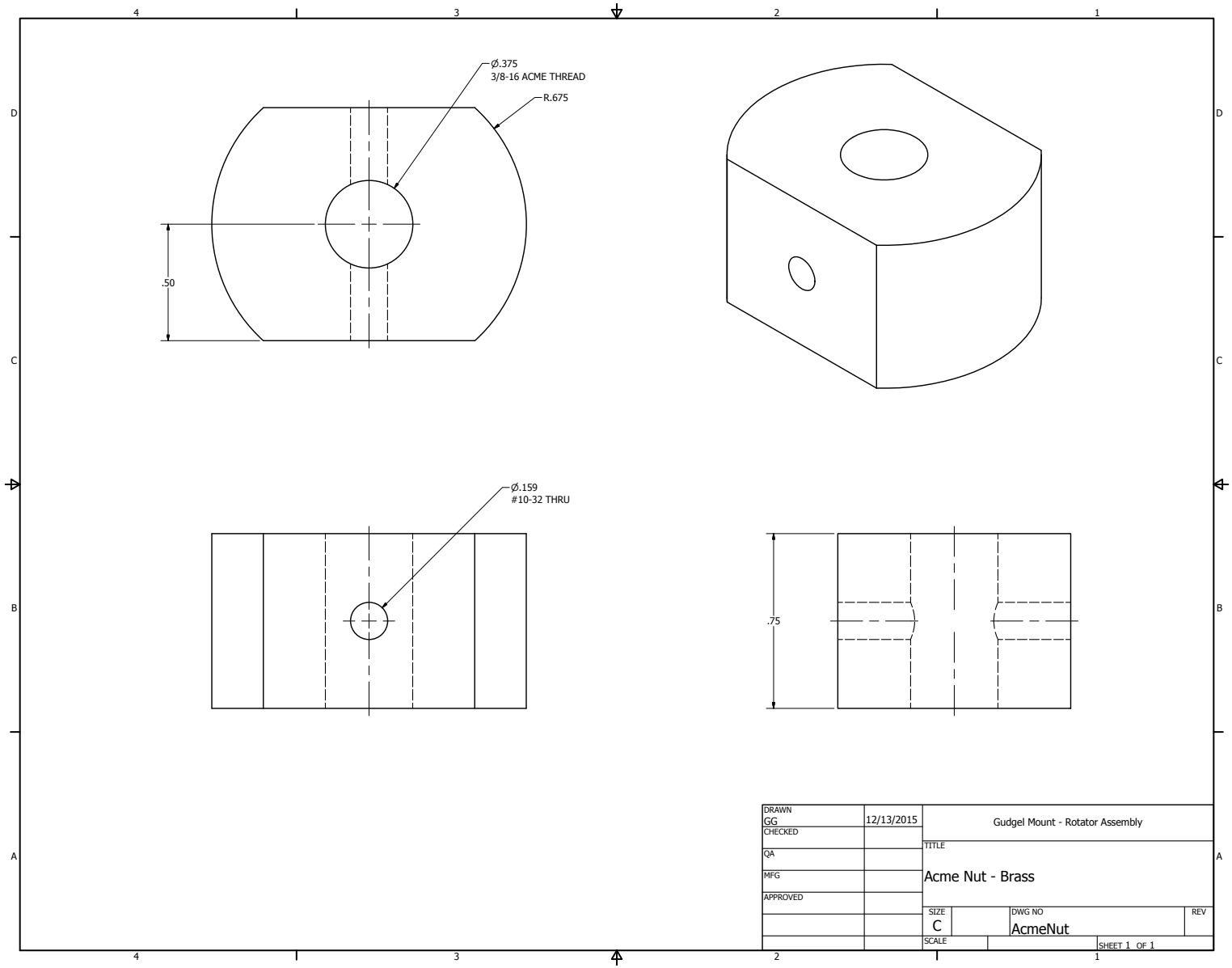
Figure A.3: *Actuator to OTA rotator assembly.*

Table A.3: *Part list: Actuator to OTA rotator assembly.*

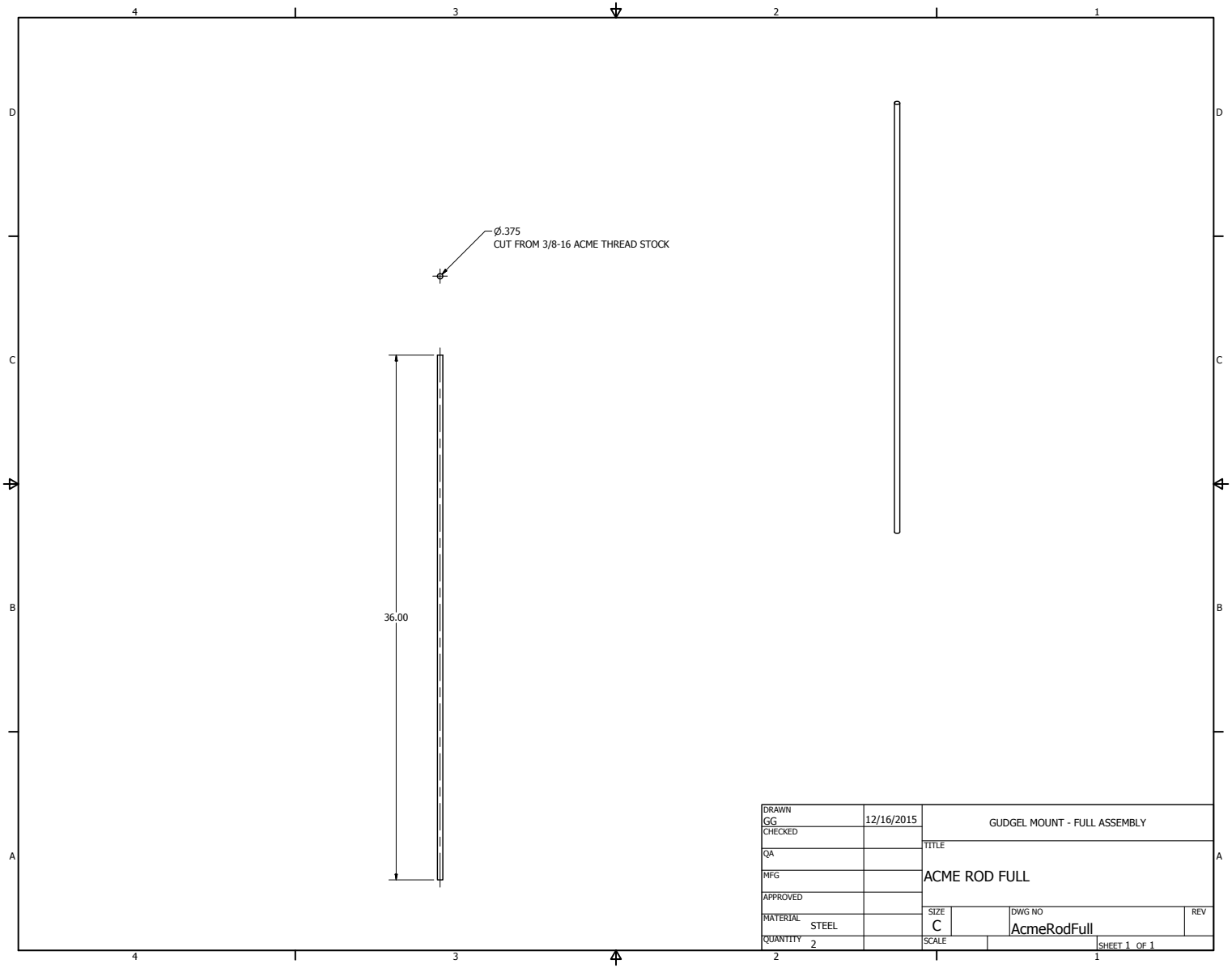
Part Name	**Quantity**
Support	1
Actuator Rotator	1
Shoulder Screw 91327A304	1
Acme Nut	1
Shoulder Screw 93996A510	2
Acme Rod Full	2
Acme Rod Short	1
Motor Shaft Coupler	1

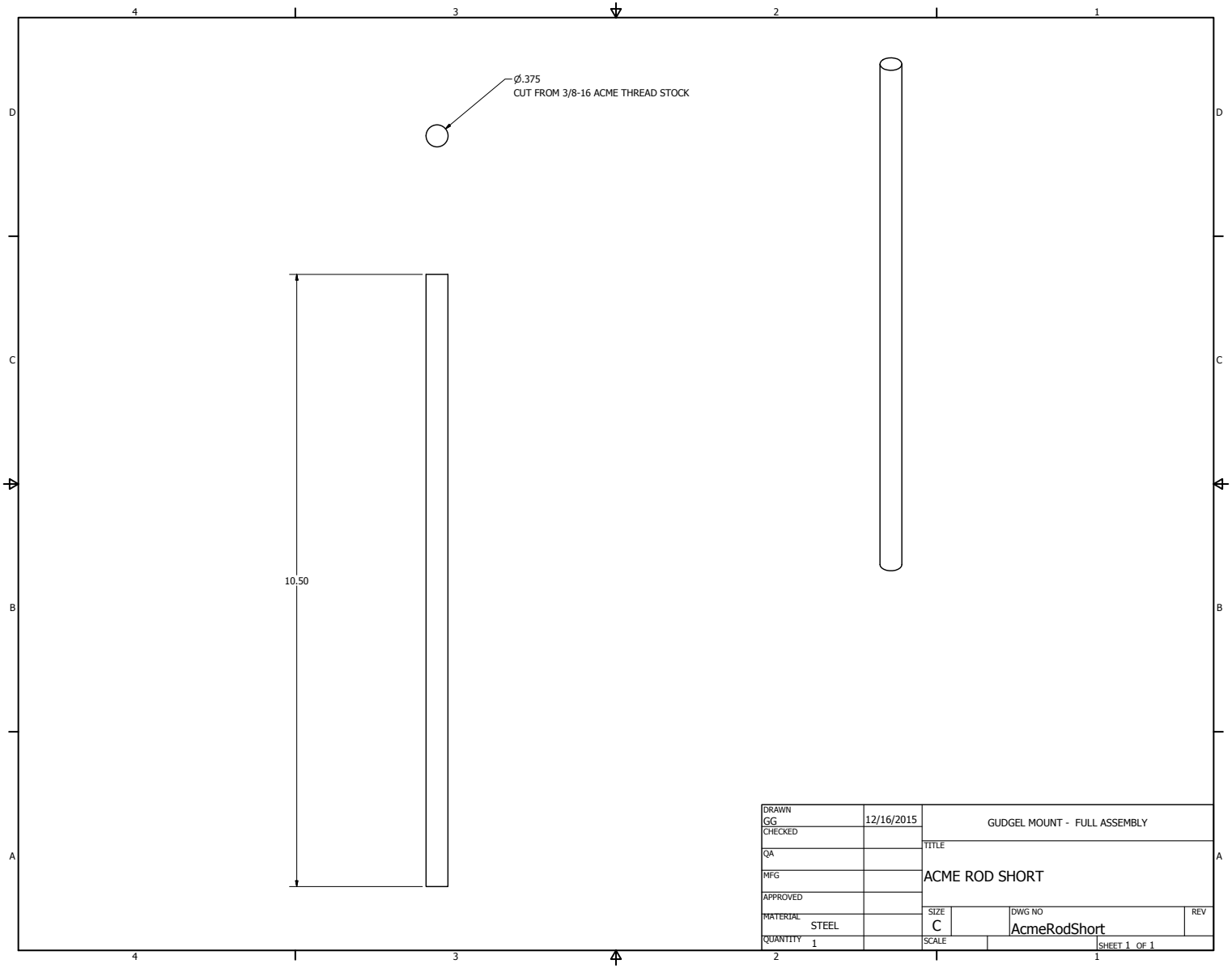

1.00
.00
1.00
Ø 0.417
1/2-13 THRU
Ø 0.307 (2X)
3/8-16 THRU
Ø3.00
1.00
DRAWN
GG
12/13/2015
CHECKED
QA
MFG
APPROVED
MATERIAL
AL 6061
QTY
3
Gudgel Mount - Rotator Assembly
TITLE
Support
SIZE
C
DWG NO
Support
REV
SCALE
SHEET 1 OF 1


1.25
.50
.00
.50
1.25
.00
.50
1.50
2.00
2.50
Ø.625 MIN THRU
Ø.250 MIN THRU
DRAWN
GG
12/13/2015
CHECKED
QA
MFG
APPROVED
Gudgel Mount - Rotator Assembly
TITLE
Actuator Rotator - AL 6061
SIZE
C
DWG NO
ActuatorRotator
REV
SCALE
SHEET 1 OF 1



DRAWN GG	12/13/2015	Gudgel Mount - Rotator Assembly	
CHECKED		TITLE	
QA		Acme Nut - Brass	
MFG			
APPROVED		SIZE C	DWG NO AcmeNut / REV
		SCALE	SHEET 1 OF 1


Ø.375
CUT FROM 3/8-16 ACME THREAD STOCK
36.00
DRAWN
GG
12/16/2015
CHECKED
QA
MFG
APPROVED
MATERIAL
STEEL
QUANTITY
2
GUDGEL MOUNT - FULL ASSEMBLY
TITLE
ACME ROD FULL
SIZE
C
DWG NO
AcmeRodFull
REV
SCALE
SHEET 1 OF 1


Ø.375
CUT FROM 3/8-16 ACME THREAD STOCK
10.50
DRAWN
GG
12/16/2015
CHECKED
QA
MFG
APPROVED
MATERIAL
STEEL
QUANTITY
1
GUDGEL MOUNT - FULL ASSEMBLY
TITLE
ACME ROD SHORT
SIZE
C
DWG NO
AcmeRodShort
REV
SCALE
SHEET 1 OF 1

A.1.4 *Full Assembly*

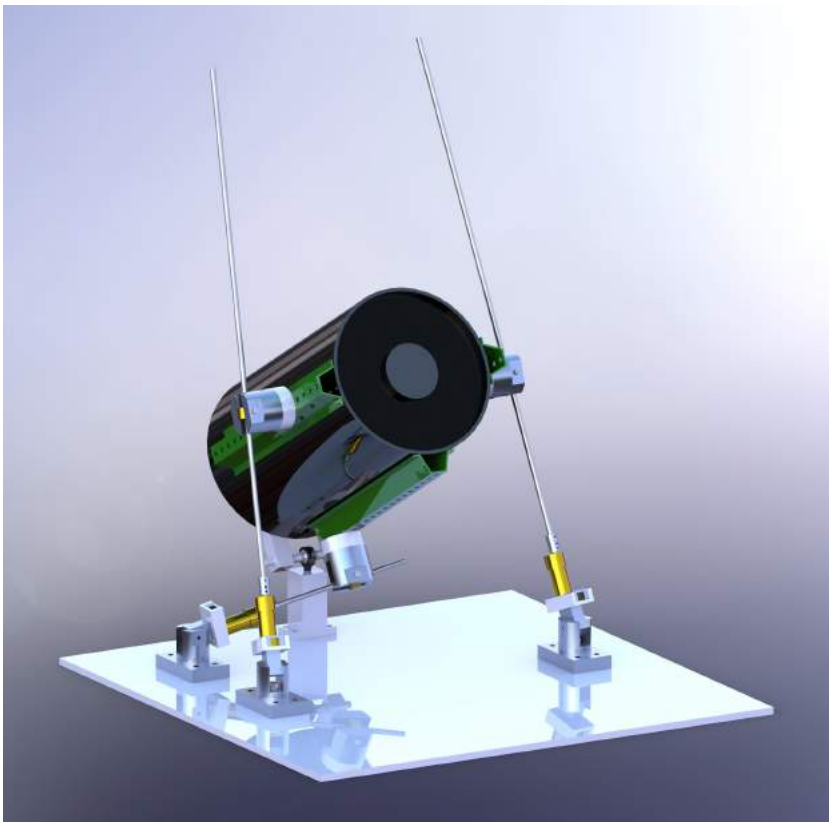
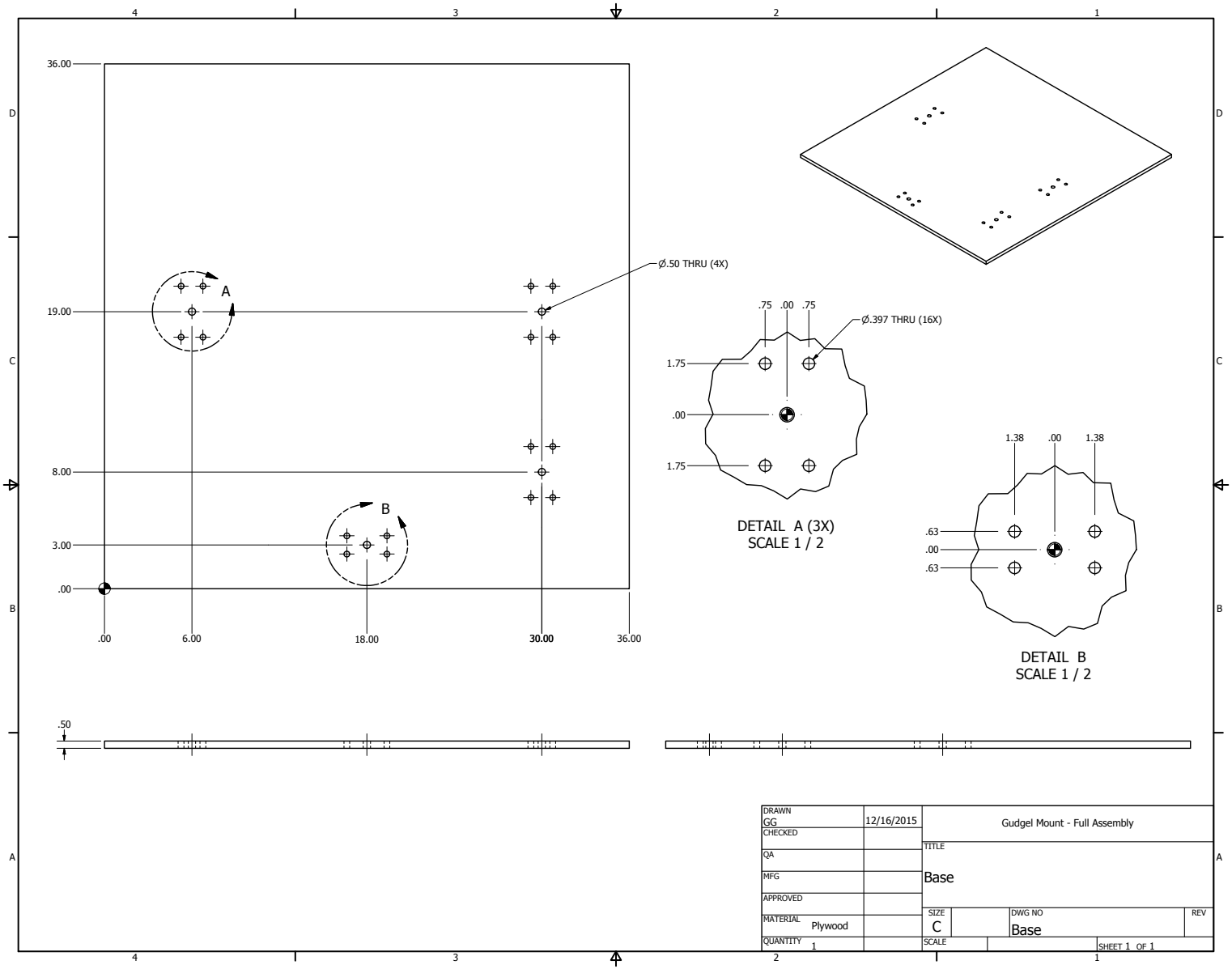


Figure A.4: *Complete mechanical assembly.*

Table A.4: *Assembly list: Complete mechanical assembly.*

Assembly Name	**Quantity**
Base	1
OTA Base Rotator Assembly	1
Actuator Base Rotator Assembly	3
Actuator to OTA Rotator Assembly	3
OTA with Rails	1


36.00
19.00
8.00
3.00
.00
.00
6.00
18.00
30.00
36.00
A
B
Ø.50 THRU (4X)
.75 .00 .75
Ø.397 THRU (16X)
1.75
.00
1.75
DETAIL A (3X)
SCALE 1 / 2
1.38 .00 1.38
.63
.00
.63
DETAIL B
SCALE 1 / 2
.50
DRAWN GG 12/16/2015
CHECKED
QA
MFG
APPROVED
MATERIAL Plywood
QUANTITY 1
Gudgel Mount - Full Assembly
TITLE
Base
SIZE C
DWG NO Base
REV
SCALE
SHEET 1 OF 1

A.2 Electrical

A.2.1 *Triple Motor-Encoder Driver Board*

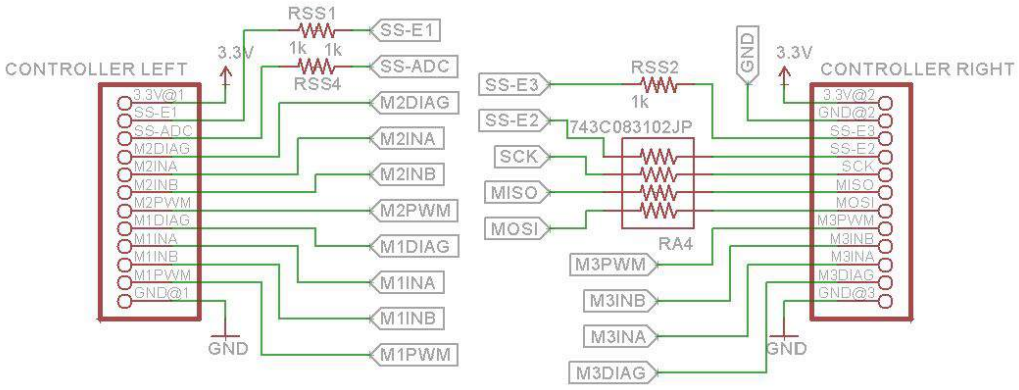


Figure A.5: *Triple Motor/Encoder Driver Board: Board IO pinout.*

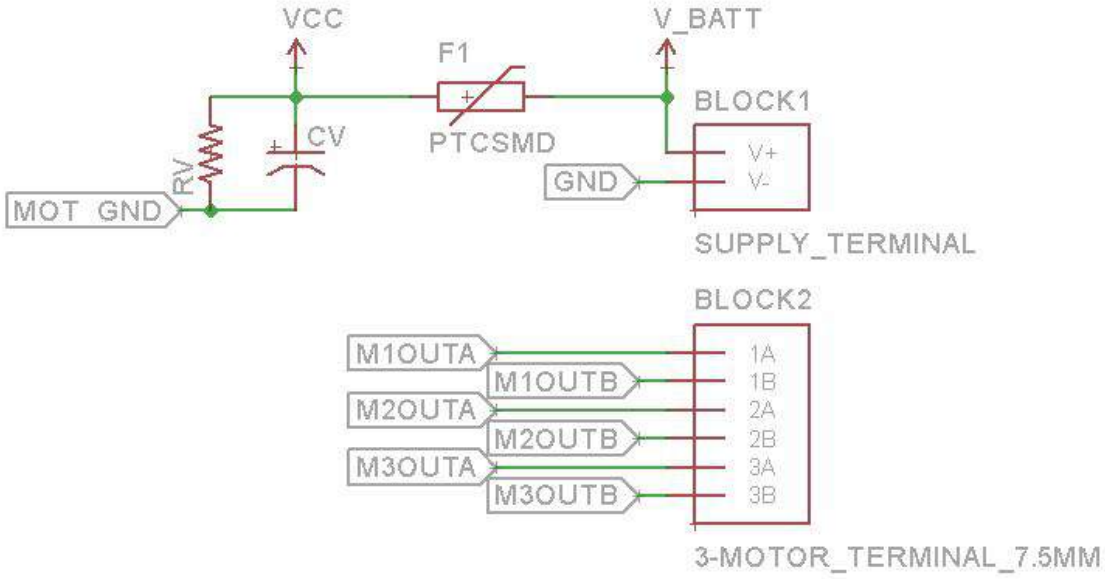


Figure A.6: *Triple Motor/Encoder Driver Board: Motor output terminals and voltage supply input terminals.*

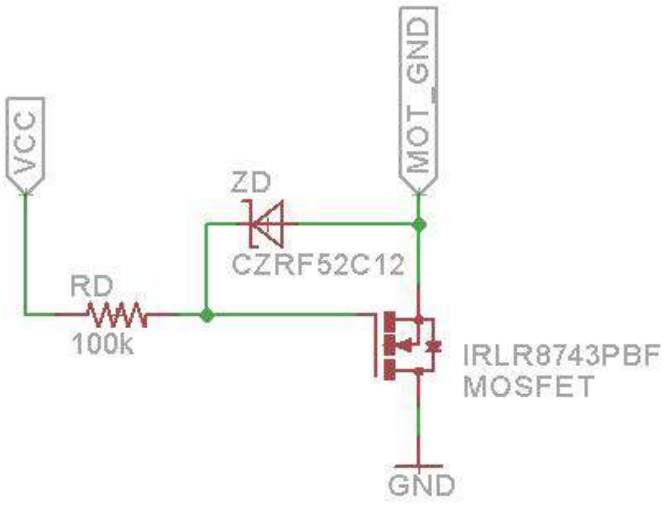


Figure A.7: *Triple Motor/Encoder Driver Board: Reverse voltage protection circuit.*

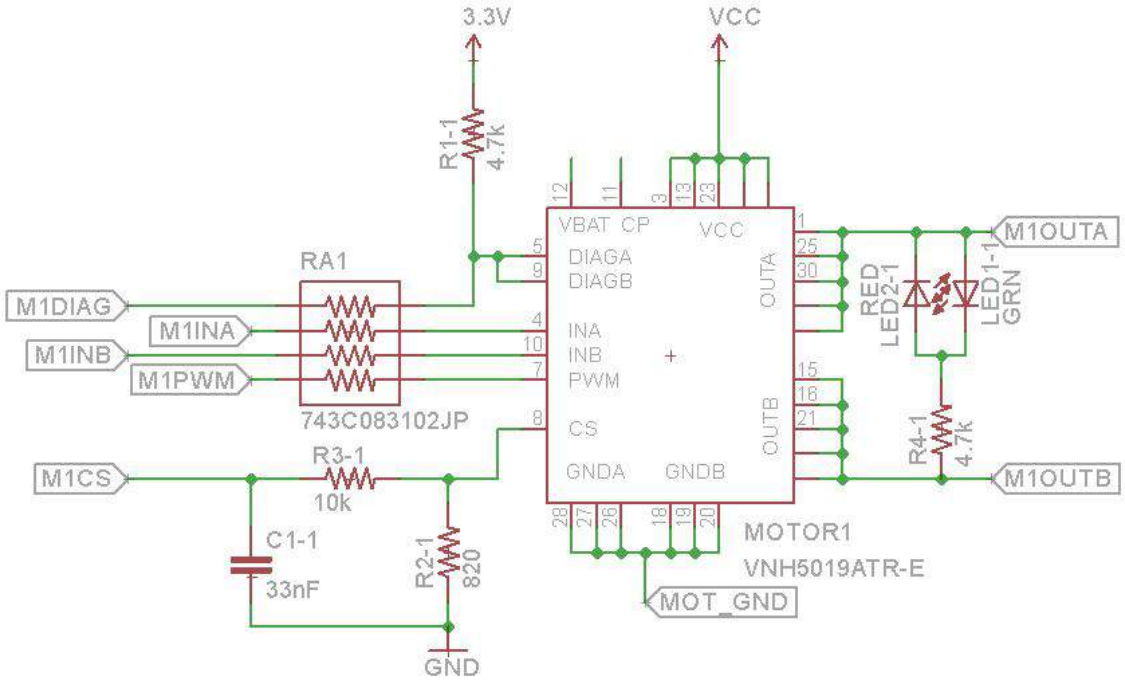


Figure A.8: *Triple Motor/Encoder Driver Board: Motor 1 voltage driver circuit.*

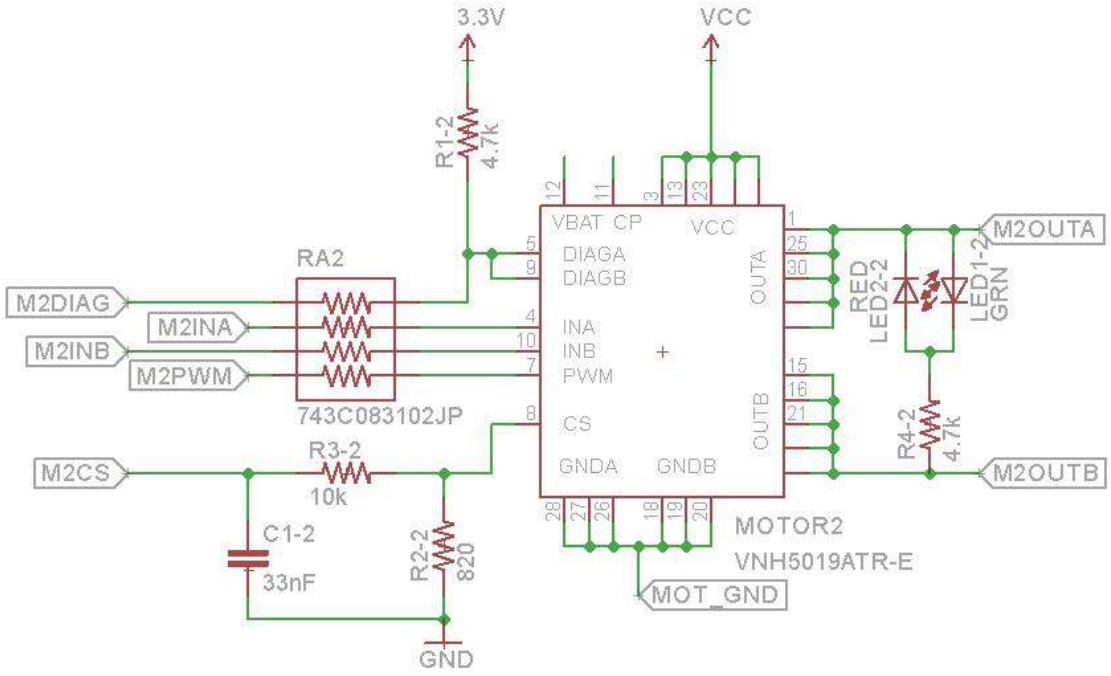


Figure A.9: *Triple Motor/Encoder Driver Board: Motor 2 voltage driver circuit.*

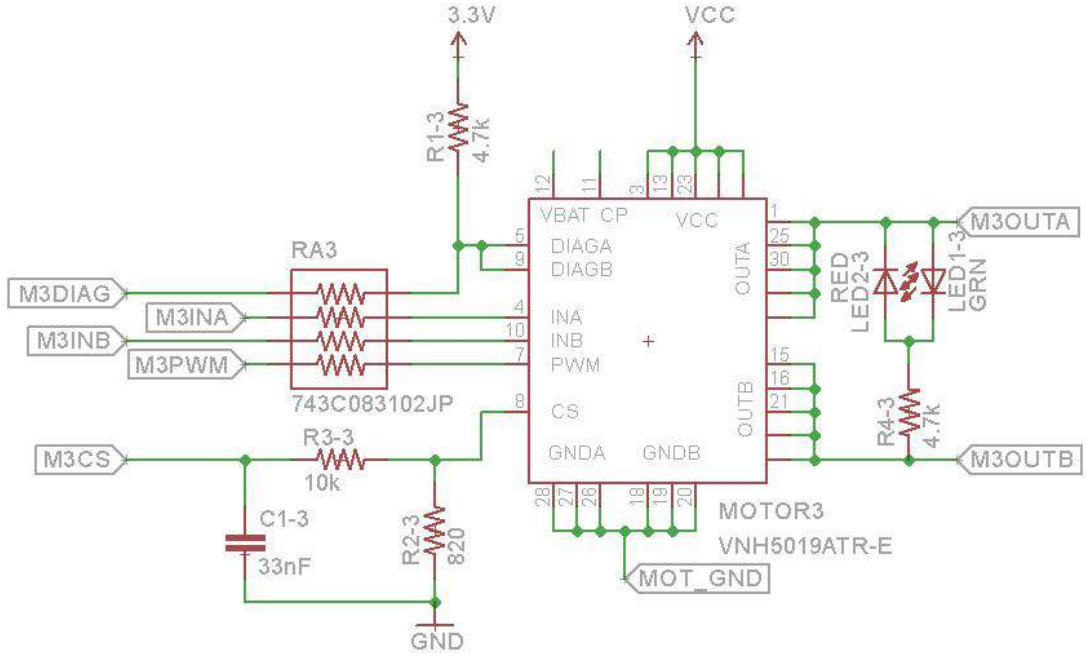


Figure A.10: *Triple Motor/Encoder Driver Board: Motor 3 voltage driver circuit.*

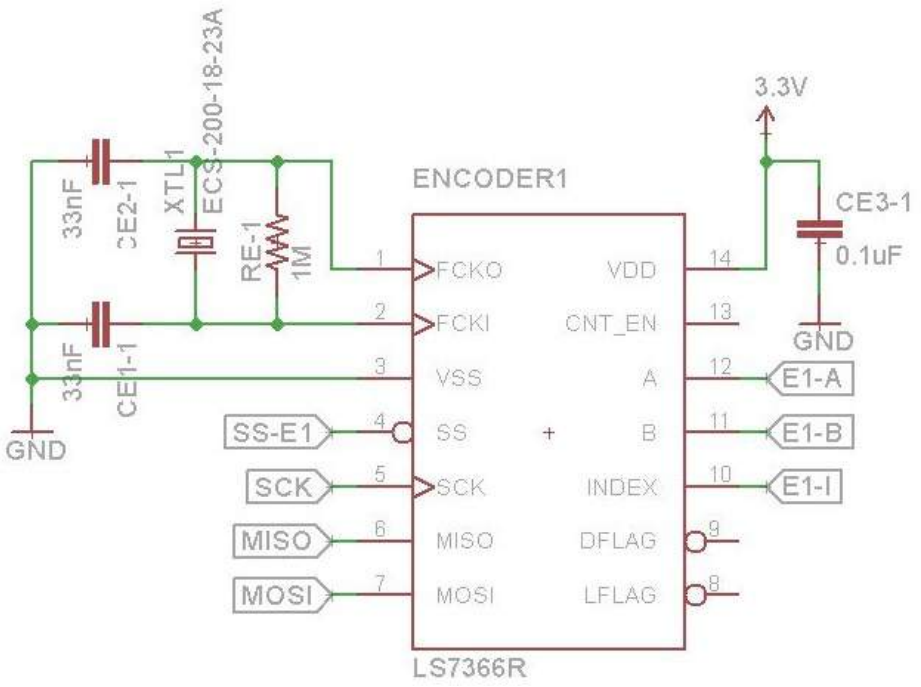


Figure A.11: *Triple Motor/Encoder Driver Board: Encoder 1 quadrature counter circuit.*

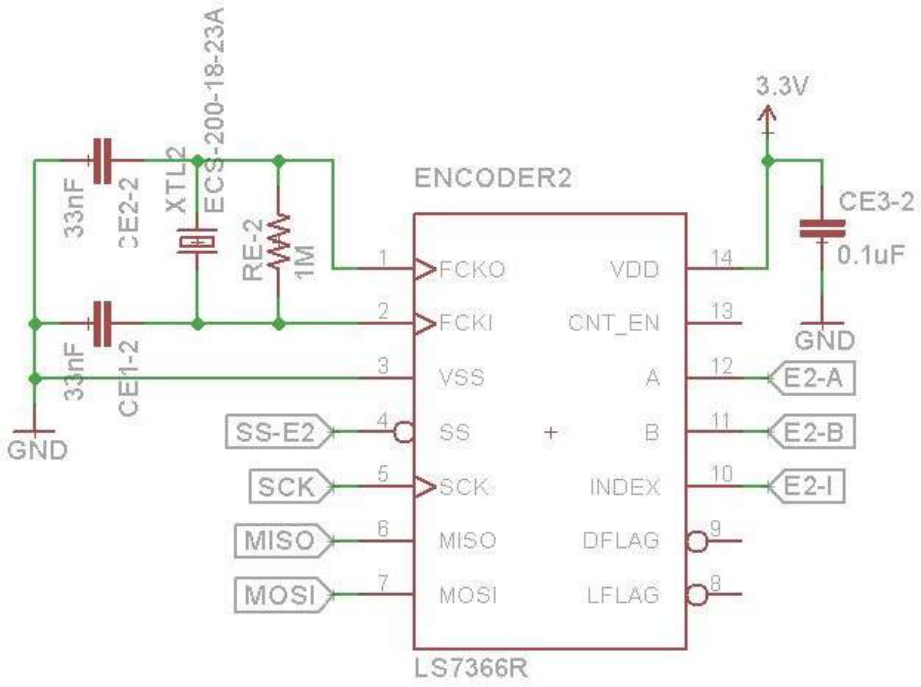


Figure A.12: *Triple Motor/Encoder Driver Board: Encoder 2 quadrature counter circuit.*

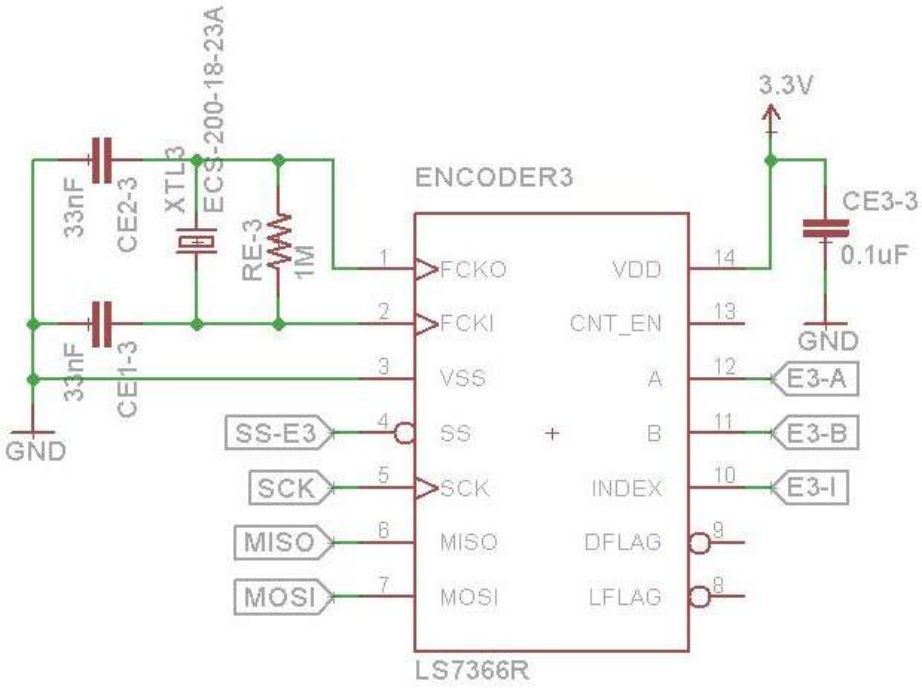


Figure A.13: *Triple Motor/Encoder Driver Board: Encoder 3 quadrature counter circuit.*

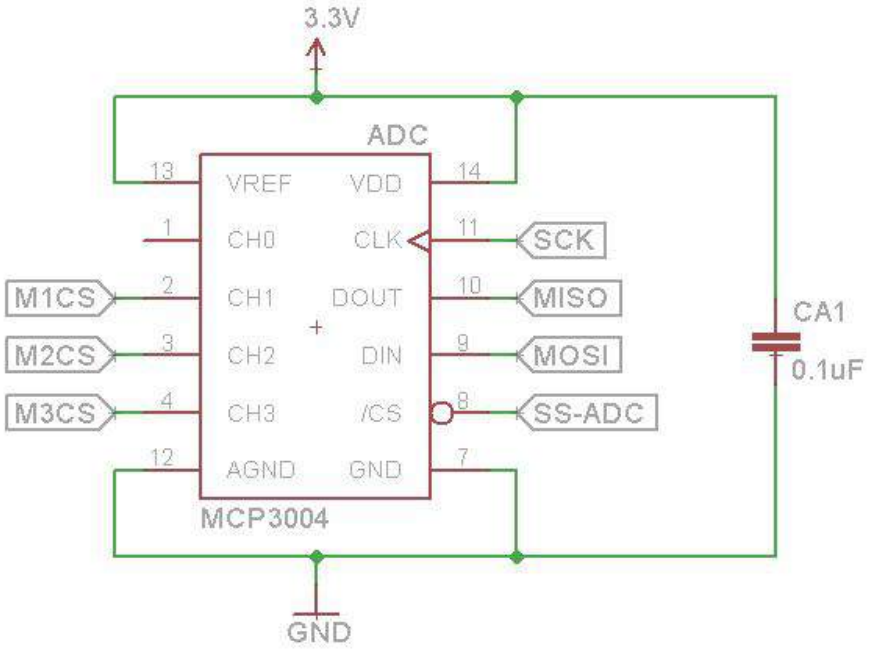


Figure A.14: *Triple Motor/Encoder Driver Board: Motor current sensing ADC circuit.*

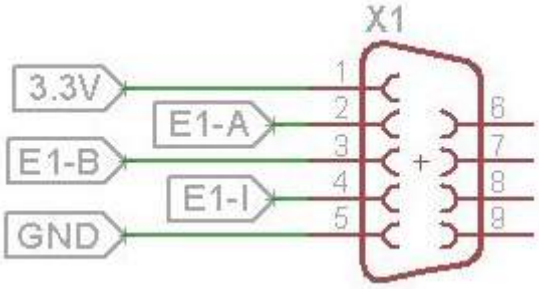


Figure A.15: *Triple Motor/Encoder Driver Board: Encoder 1 DB9 connector.*

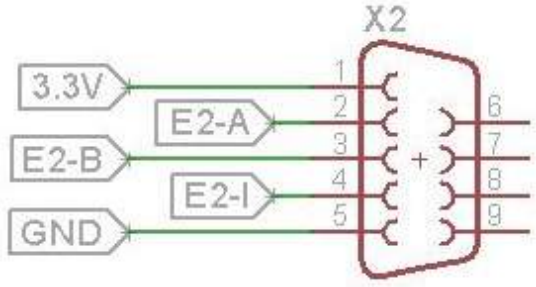


Figure A.16: *Triple Motor/Encoder Driver Board: Encoder 2 DB9 connector.*

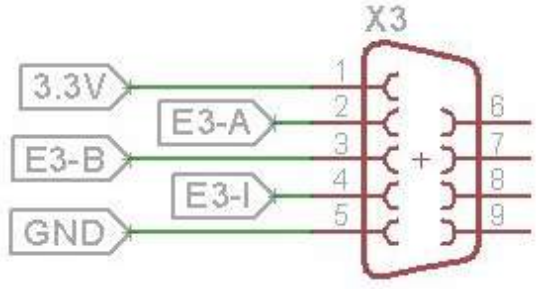


Figure A.17: *Triple Motor/Encoder Driver Board: Encoder 3 DB9 connector.*

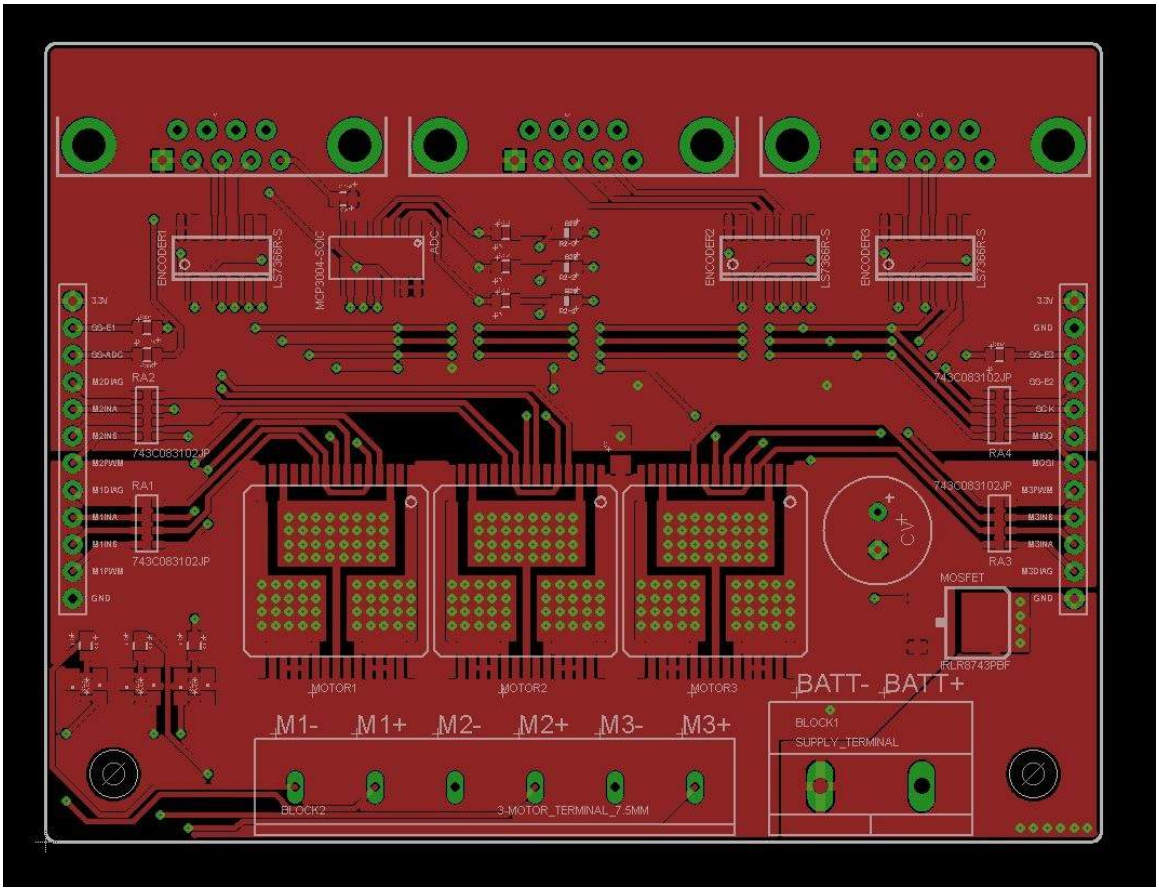


Figure A.18: *Triple Motor/Encoder Driver Board: Board top layer.*

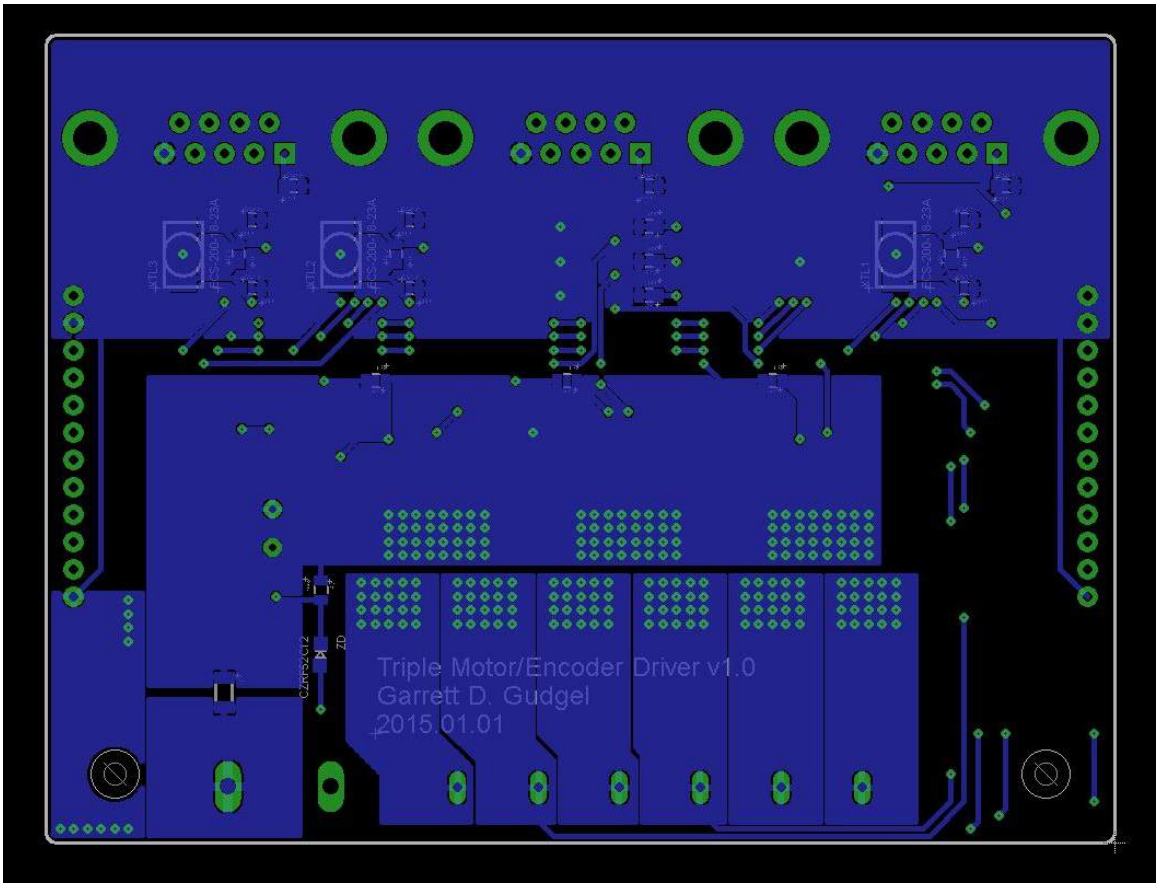


Figure A.19: *Triple Motor/Encoder Driver Board: Board bottom layer.*

A.2.2 *Triple Motor-Encoder Shield: Mbed Control*

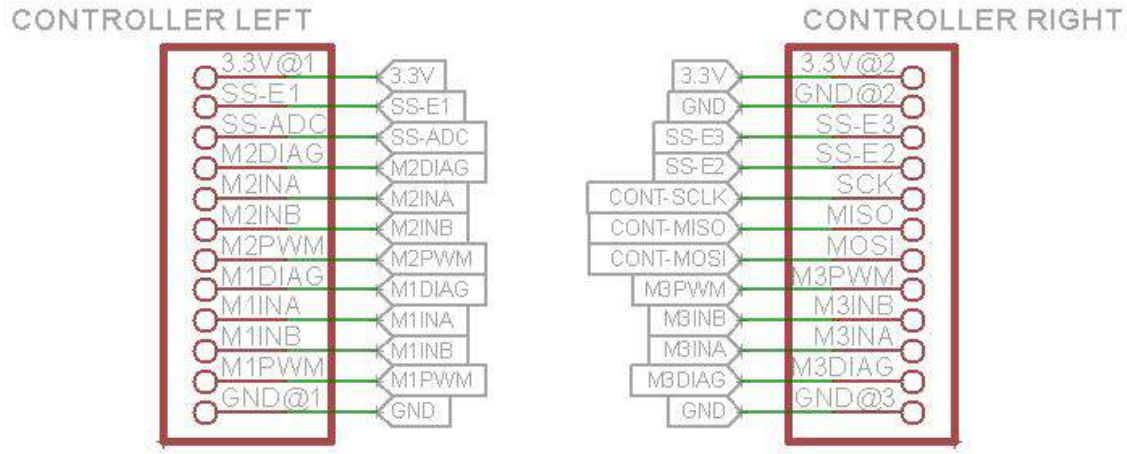


Figure A.20: *Mbed Shield: Triple Motor-Encoder pinout.*

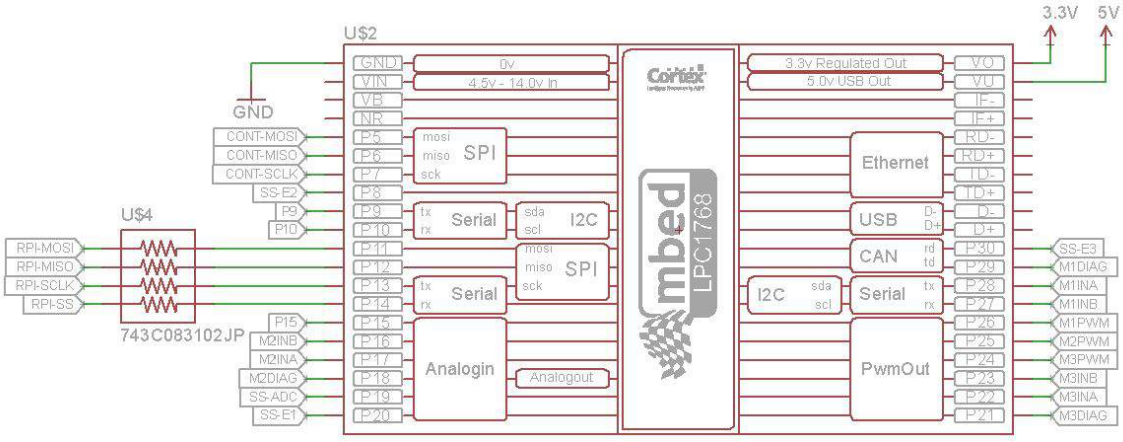


Figure A.21: *Mbed Shield: Mbed drop pinout. [19]*

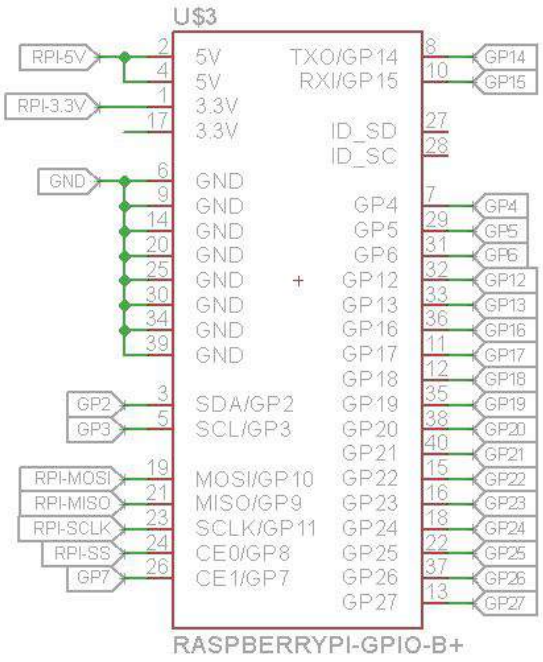


Figure A.22: *Mbed Shield: Raspberry Pi B+ ribbon pinout. Communication with Mbed over SPI if Raspberry Pi is used as primary computer, otherwise unused.*

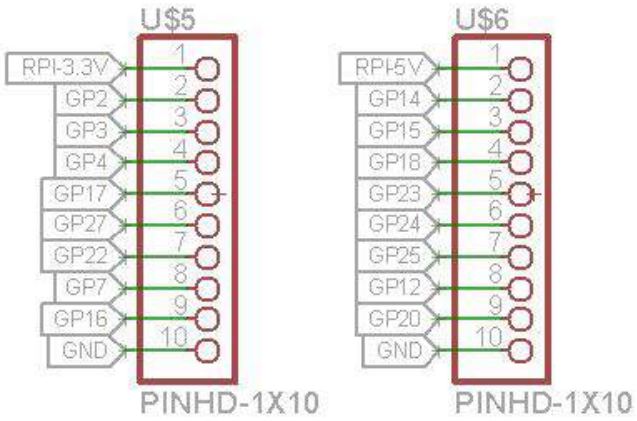


Figure A.23: *Mbed Shield: Raspberry Pi B+ additional GPIO.*

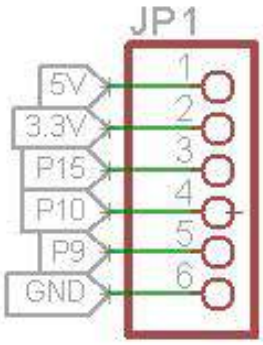


Figure A.24: *Mbed Shield: Mbed additional GPIO.*

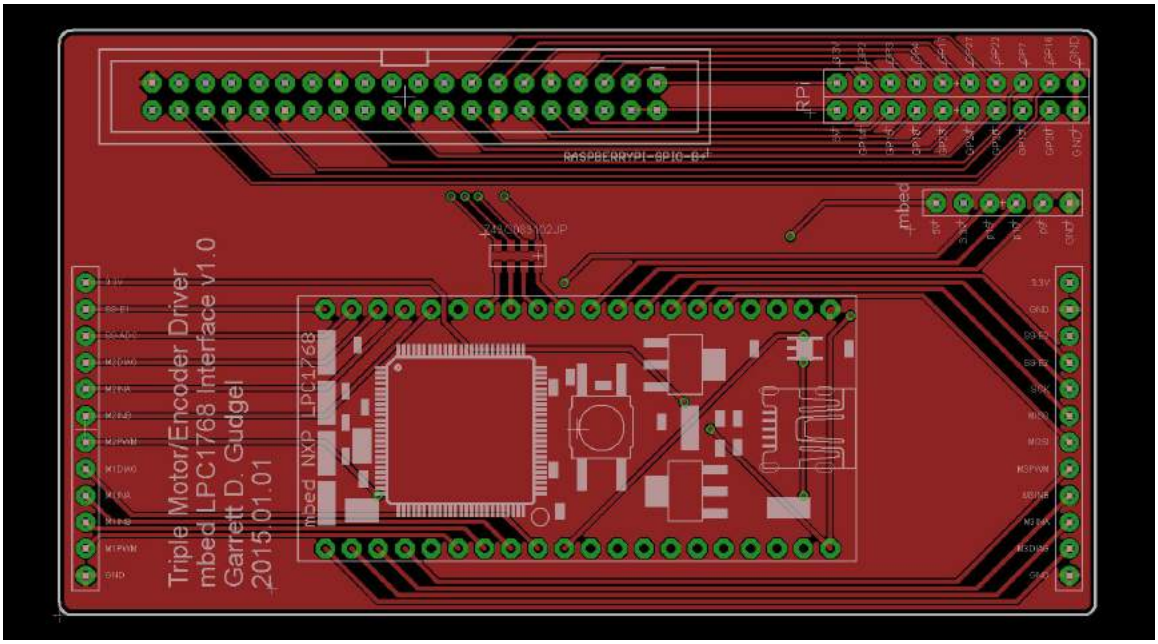


Figure A.25: *Mbed Shield: Board top layer.*

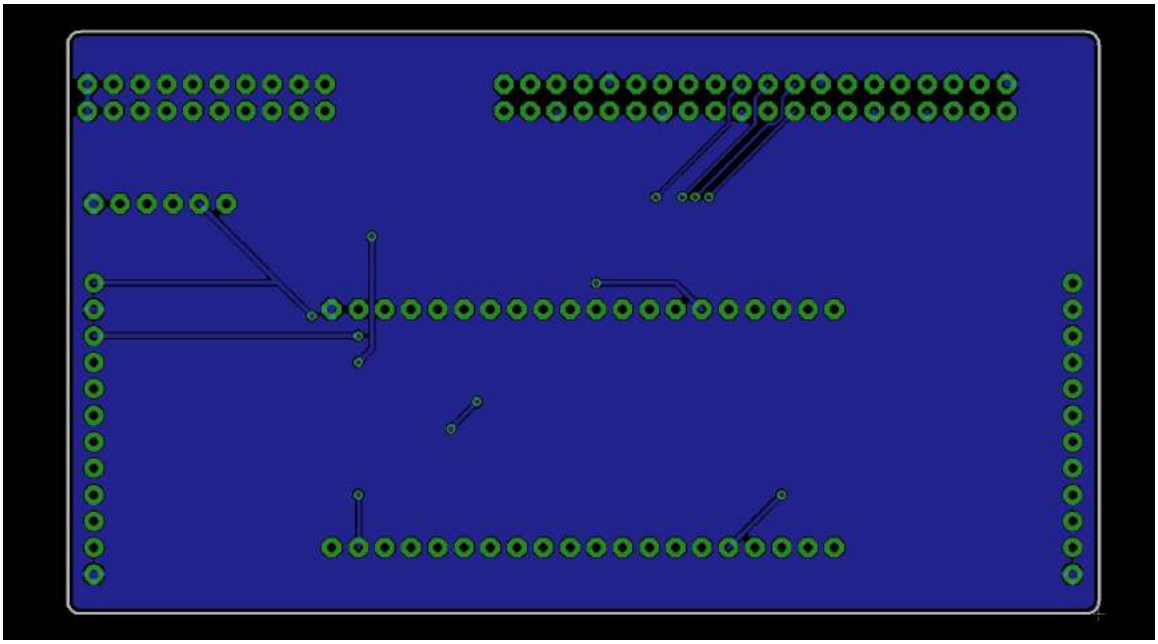


Figure A.26: *Mbed Shield: Board bottom layer.*

B.1 Python Scripts

B.1.1 *Test Mount Configuration*

File to run:

- *GudgelMount_DesignStudies.py*

Necessary additional libraries:

- *telescope_class.py*
- *telescope.csv*
- *tFunctions.py*
- numpy

```
import telescope_class
import numpy as np

class GudgelMount_DesignStudies():
    """
    Contains tests outlined in Section 3.3 to perform on the telescope
    configuration defined in .csv file FILENAME across the field-of-view
    determined by inputs THETA1_deg, THETA2_deg, and THETA3_deg.
    """
    def __init__(self, FILENAME='telescope.csv', THETA1_deg=range(-15,15+1,5),
                 THETA2_deg=range(30,100+1,5), THETA3_deg=range(-15,15+1,5),
                 L_MIN=[6.3,6.3,6.3], L_MAX=[36,36,20],
                 D=[0.375, 0.375, 0.375], SKY_RES_deg=1./3600):
        """
        FILENAME: Title of .csv file with desired telescope configuration.
        THETA1_deg: Range of theta1 values determining azimuth field-of-view.
        THETA2_deg: Range of theta2 values determining altitude field-of-view.
        THETA3_deg: Range of theta3 values determining desired telescope roll.
        L_MIN:      Array of actuator minimum lengths.
        L_MAX:      Array of actuator maximum lengths.
        D:          Array of actuator diameters.
        SKY_RES:    Desired resolution of telescope direction.

        Default values are representative of prototype telescope.
        """
        self.OTA = telescope_class.telescope('telescope.csv')
        self.theta1range_deg = THETA1_deg
        self.theta2range_deg = THETA2_deg
        self.theta3range_deg = THETA3_deg
        self.length_min      = L_MIN
        self.length_max      = L_MAX
        self.sky_resolution  = SKY_RES_deg
        self.actDiameter     = D

    def Test_Length(self):
        """
        Iterate through field-of-view and determine if telescope configuration
        can operate within actuator minimum and maximum lengths.
        """
        MIN = self.length_max*1
        MIN_LOC = np.matrix( np.zeros(shape=[3,3], dtype=float) )
        MAX = [0, 0, 0]
        MAX_LOC = np.matrix( np.zeros(shape=[3,3], dtype=float) )

        for theta1 in self.theta1range_deg:
            for theta2 in self.theta2range_deg:
                for theta3 in self.theta3range_deg:
                    lengths = self.OTA.calcActLength(theta1, theta2, theta3)
                    L = [lengths['L1'], lengths['L2'], lengths['L3']]
                    for i in range(3):
                        if L[i] < MIN[i]:
                            MIN[i] = L[i]
                            MIN_LOC[:,i] = np.matrix( [ [float(theta1)],
```

```
                                                    [float(theta2)],
                                                    [float(theta3)] ] )
                    if L[i] > MAX[i]:
                        MAX[i] = L[i]
                        MAX_LOC[:,i] = np.matrix( [ [float(theta1)],
                                                    [float(theta2)],
                                                    [float(theta3)] ] )
        PASS = True
        for i in range(3):
            PASS = PASS and (MIN[i] > self.length_min[i])
            PASS = PASS and (MAX[i] < self.length_max[i])
        return PASS, MIN, MIN_LOC, MAX, MAX_LOC
        pass

    def Test_Resolution(self):
        """
        Iterate through field-of-view and determine if a minimum of one encoder
        count can separate any given actuator configuration between those
        surrounding points of a distance of the desired sky resolution.
        """
        countNormMin = 10^1000
        countNormMin_LOC = [0,0,0]
        for theta1 in self.theta1range_deg:
            for theta2 in self.theta2range_deg:
                for theta3 in self.theta3range_deg:
                    if theta2 != 90:
                        o_count,s_count = self._Test_Resolution_Counts(theta1,
                                                                theta2, theta3)
                        for counts in s_count:
                            c1_diff = abs(o_count[0] - counts[0])
                            c2_diff = abs(o_count[1] - counts[1])
                            c3_diff = abs(o_count[2] - counts[2])
                            countNorm = np.linalg.norm(
                                [c1_diff,c2_diff,c3_diff] )
                            if countNorm < countNormMin:
                                countNormMin = countNorm
                                countNormMin_LOC = [theta1, theta2, theta3]
        if countNormMin < 1:
            PASS = False
        else:
            PASS = True
        return PASS, countNormMin, countNormMin_LOC

    def _Test_Resolution_Counts(self, THETA1, THETA2, THETA3):
        """
        Check all surrounding points using combinations of +/- SKY_RES in all
        three axes: THETA1, THETA2, and THETA3.  Return encoder counts for the
        original point and an array of encoder counts for each of 26
        surrounding points.
        Used by function Test_Resolution(...).
        """
        counts = self.OTA.calcActCounts(THETA1, THETA2, THETA3)
        o_count = [ counts['C1'], counts['C2'], counts['C3'] ]
```

```
        theta1_vals = [THETA1-self.sky_resolution, THETA1,
                       THETA1+self.sky_resolution]
        theta2_vals = [THETA2-self.sky_resolution, THETA2,
                       THETA2+self.sky_resolution]
        theta3_vals = [THETA3-self.sky_resolution, THETA3,
                       THETA3+self.sky_resolution]
        s_count = []
        for theta1 in theta1_vals:
            for theta2 in theta2_vals:
                for theta3 in theta3_vals:
                    if (theta1 != THETA1 or theta2 != THETA2 or
                                          theta3 != THETA3):
                        counts = self.OTA.calcActCounts(theta1, theta2, theta3)
                        s_count.append( [ counts['C1'], counts['C2'],
                                          counts['C3'] ] )
        return o_count, s_count

    def Test_Interference(self):
        """
        Iterate through field-of-view and determine if interference occurs
        between any two actuators or between any actuator and the OTA.
        """
        PASS = True
        interfere_LOC = [0,0,0]

        D1 = self.actDiameter[0]
        D2 = self.actDiameter[1]
        D3 = self.actDiameter[2]
        L1 = self.length_max[0]
        L2 = self.length_max[1]
        L3 = self.length_max[2]
        actGND = self.OTA.getActuatorGrounds()

        for theta1 in self.theta1range_deg:
            for theta2 in self.theta2range_deg:
                for theta3 in self.theta3range_deg:
                    info = self.OTA.calcActLength(theta1, theta2, theta3)
                    actOTA = info['newOTA']
                    pass12 = self._Test_Interference_Interfere(
                        actGND[:,0], actOTA[:,0], D1, L1,
                        actGND[:,1], actOTA[:,1], D2, L2)
                    pass23 = self._Test_Interference_Interfere(
                        actGND[:,1], actOTA[:,1], D2, L2,
                        actGND[:,2], actOTA[:,2], D3, L3)
                    pass13 = self._Test_Interference_Interfere(
                        actGND[:,0], actOTA[:,0], D1, L1,
                        actGND[:,2], actOTA[:,2], D3, L3)
                    if pass12 and pass23 and pass13:
                        pass
                    else:
                        PASS = False
                        interfere_LOC = [theta1, theta2, theta3]
        return PASS, interfere_LOC
```

```
    def _Test_Interference_Interfere(self,
                                     LN1_GND, LN1_EXT, LN1_DIAM, LN1_LENGTH,
                                     LN2_GND, LN2_EXT, LN2_DIAM, LN2_LENGTH):
        """
        Using points defining two skew lines, determine if interference occurs
        by finding the closest distance between both lines and comparing it to
        the safe distance between both lines as calculated by the diameters.
        If interference were to occur, determine if skewed line interference
        occurs within the stroke of the actuator as there would be no
        interference if the crossing point is outside the actuator range.
        Follows Section 3.3.3 procedure.
        Used by function Test_Interference(...).
        """
        O1 = LN1_GND
        u1 = LN1_EXT - LN1_GND
        O2 = LN2_GND
        u2 = LN2_EXT - LN2_GND

        n  = np.cross(u1, u2, axis=0)

        A = np.matrix( np.zeros(shape=[3,3], dtype=float) )
        A[:,0] = u1
        A[:,1] = -u2
        A[:,2] = n

        b = O2-O1

        x = np.linalg.inv( A ) * b

        P1 = O1 + x[0,0]*u1
        P2 = O2 + x[1,0]*u2

        closestDistance = np.linalg.norm(x[2,0]*n)

        testDiameterInterfere = closestDistance < ( (LN1_DIAM+LN2_DIAM)/2 )
        testWithinLength1     = np.linalg.norm(P1-O1) < LN1_LENGTH
        testWithinLength2     = np.linalg.norm(P2-O2) < LN2_LENGTH

        if testDiameterInterfere and testWithinLength1 and testWithinLength2:
            PASS = False
        else:
            PASS = True
        return PASS

if __name__ == '__main__':
    designStudy = GudgelMount_DesignStudies()
    print designStudy.Test_Length()
    print designStudy.Test_Resolution()
    print designStudy.Test_Interference()
```

B.1.2 *Operate Test GUI*

File to run:

- *GudgelMount_GUI.py*

Necessary additional libraries:

- *serial_thread.py*
- *telescope_thread.py*
- *shares.py*
- *myTk.py*
- *telescope_class.py*
- *telescope.csv*
- *tFunctions.py*
- pySerial
- numpy

GudgelMount_GUI.py

```
import Tkinter as tk
import logging
import time
import threading

import myTk
import serial_thread
import telescope_thread
import shares

buttonWidth = 10

class telescopeGUI():
    def __init__(self, PARENT, SHARES):
        """
        Setup GUI and initialize.
        """
        self.parent = PARENT
        self.Shares = SHARES

        tk.Label(self.parent, text='GUDGEL MOUNT 0.2', font=("bold")).grid(
            row=0, column=0, columnspan=2)

        tk.Label(self.parent, text='Master On/Off Switch:').grid(
            row=2, column=0, padx=5, pady=5)
        self.masterOnOff = myTk.ToggleButton(
            self.parent, command=self.turnOnOff, width=buttonWidth)
        self.masterOnOff.grid(row=2, column=1, padx=5, pady=5)

        tk.Label(self.parent, text='Desired State').grid(
            row=4, column=0, padx=5, pady=5)
        stateMenu = ['Hold','Calibrate','Run']
        self.stateVar = tk.StringVar()
        self.stateVar.set(stateMenu[0])
        self.stateMenu = tk.OptionMenu(
            self.parent, self.stateVar, *stateMenu, command=self.newState)
        self.stateMenu.grid(
            row=4, column=1, sticky=tk.NW+tk.SE, padx=3, pady=5)
        self.stateMenu.config(width=buttonWidth)

        tk.Label(self.parent, text='Update Setpoint').grid(
            row=5, column=0, padx=5, pady=5)
        tk.Button(self.parent, text='Go', command=self.goToPos,
                  width=buttonWidth).grid(row=5, column=1, sticky=tk.NW+tk.SE,
                                          padx=5, pady=5)
        self.parent.bind("<Return>", self.goToPos)

        tk.Label(self.parent, text='Resend Home').grid(
            row=6, column=0, padx=5, pady=5)
        tk.Button(self.parent, text='Send', command=self.resendHome,
                  width=buttonWidth).grid(row=6, column=1, sticky=tk.NW+tk.SE,
                  padx=5, pady=5)
        self.parent.bind("<Return>", self.goToPos)
```

```
        tk.Frame(width=3, bd=1, relief=tk.SUNKEN, background='black').grid(
            row=0, column=2, rowspan=15, sticky=tk.NW+tk.SE, padx=5, pady=5)
        tk.Frame(width=1, bd=1, relief=tk.SUNKEN, background='grey').grid(
            row=1, column=4, rowspan=15, sticky=tk.NW+tk.SE, padx=5, pady=5)
        tk.Frame(width=1, bd=1, relief=tk.SUNKEN, background='grey').grid(
            row=1, column=6, rowspan=15, sticky=tk.NW+tk.SE, padx=5, pady=5)
        tk.Frame(width=1, bd=1, relief=tk.SUNKEN, background='grey').grid(
            row=1, column=8, rowspan=15, sticky=tk.NW+tk.SE, padx=5, pady=5)
        tk.Frame(width=1, bd=1, relief=tk.SUNKEN, background='grey').grid(
            row=1, column=10, rowspan=15, sticky=tk.NW+tk.SE, padx=5, pady=5)
        tk.Frame(width=1, bd=1, relief=tk.SUNKEN, background='black').grid(
            row=0, column=12, rowspan=15, sticky=tk.NW+tk.SE, padx=5, pady=5)
        tk.Frame(width=1, bd=1, relief=tk.SUNKEN, background='grey').grid(
            row=1, column=14, rowspan=15, sticky=tk.NW+tk.SE, padx=5, pady=5)
        tk.Frame(width=1, bd=1, relief=tk.SUNKEN, background='grey').grid(
            row=1, column=16, rowspan=15, sticky=tk.NW+tk.SE, padx=5, pady=5)
        tk.Frame(height=1, bd=1, relief=tk.SUNKEN, background='black').grid(
            row=1, column=2, columnspan=17, sticky=tk.NW+tk.SE, padx=5, pady=5)
        tk.Frame(height=3, bd=1, relief=tk.SUNKEN, background='black').grid(
            row=3, column=2, columnspan=17, sticky=tk.NW+tk.SE, padx=5, pady=5)

        tk.Label(self.parent, text='Mount Direction [deg]').grid(
            row=0, column=3, columnspan=9)
        tk.Label(self.parent, text='Theta 1').grid(row=4, column=3)
        tk.Label(self.parent, text='Theta 2').grid(row=5, column=3)
        tk.Label(self.parent, text='Theta 3').grid(row=6, column=3)

        tk.Label(self.parent, text='Desired').grid(row=2, column=5)
        self.varDirectionDes = [tk.StringVar(), tk.StringVar(), tk.StringVar()]
        for i in range(3):
            self.varDirectionDes[i].set('0')
            tk.Entry(self.parent, textvariable=self.varDirectionDes[i],
                     width=10, justify=tk.RIGHT).grid(row=i+4, column=5)

        tk.Label(self.parent, text='Set Point').grid(row=2, column=7)
        self.varDirectionSet = [tk.StringVar(), tk.StringVar(), tk.StringVar()]
        for i in range(3):
            self.varDirectionSet[i].set('0')
            tk.Label(self.parent, textvariable=self.varDirectionSet[i],
                     width=10).grid(row=i+4, column=7)

        tk.Label(self.parent, text='Way Point').grid(row=2, column=9)
        self.varWayPoint = [tk.StringVar(), tk.StringVar(), tk.StringVar()]
        for i in range(3):
            self.varWayPoint[i].set('0')
            tk.Label(self.parent, textvariable=self.varWayPoint[i],
                     width=10).grid(row=i+4, column=9)

        tk.Label(self.parent, text='Actual').grid(row=2, column=11)
        self.varDirectionAct = [tk.StringVar(), tk.StringVar(), tk.StringVar()]
        for i in range(3):
            self.varDirectionAct[i].set('0')
```

```
            tk.Label(self.parent, textvariable=self.varDirectionAct[i],
                     width=10).grid(row=i+4, column=11)

        tk.Label(self.parent, text='L1').grid(row=4, column=13)
        tk.Label(self.parent, text='L2').grid(row=5, column=13)
        tk.Label(self.parent, text='L3').grid(row=6, column=13)

        tk.Label(self.parent, text='Actuator Lengths').grid(
            row=0, column=13, columnspan=5)

        tk.Label(self.parent, text='[in]').grid(row=2, column=15)
        self.varLen = [tk.StringVar(), tk.StringVar(), tk.StringVar()]
        for i in range(3):
            self.varLen[i].set('0')
            tk.Label(self.parent, textvariable=self.varLen[i], width=10).grid(
                row=i+4, column=15)

        tk.Label(self.parent, text='Actual\n[count]').grid(row=2, column=17)
        self.varCount = [tk.StringVar(), tk.StringVar(), tk.StringVar()]
        for i in range(3):
            self.varCount[i].set('0')
            tk.Label(self.parent, textvariable=self.varCount[i],
                     width=10).grid(row=i+4, column=17)

        tk.Label(self.parent, text='Desired\n[count]').grid(row=2, column=18)
        self.varDesCount = [tk.StringVar(), tk.StringVar(), tk.StringVar()]
        for i in range(3):
            self.varDesCount[i].set('0')
            tk.Label(self.parent, textvariable=self.varDesCount[i],
                     width=10).grid(row=i+4, column=18)

        self.statusMsg = tk.StringVar()
        self.Shares.set_GuiStatus('Init')
        tk.Label(self.parent, textvariable=self.statusMsg, bd=1,
                 relief=tk.SUNKEN, anchor=tk.W).grid(row=10, column=0,
                 columnspan=20, sticky=tk.NE+tk.SW)

        # Initialization
        self._after = None
        self.parent.protocol("WM_DELETE_WINDOW", self._windowClose)
        self.turnOnOff(self.masterOnOff.getStateVar())
        self.newState(self.stateVar.get())
        self.goToPos()
        self.updateValues()
        myTk.center(self.parent)

    def _windowClose(self):
        """
        Perform these tasks before the GUI closes.
        """
        if self._after is not None:
            self.parent.after_cancel(self._after)
            self._after = None
```

```
        self.turnOnOff('Off')
        self.parent.destroy()

    def turnOnOff(self, VALUE):
        """
        OnOff button pressed.
        """
        self.stateVar.set('Hold')
        self.newState('Hold')
        if (VALUE=='On'):
            self.Shares.set_GuiOnOff(True)
        else:
            self.Shares.set_GuiOnOff(False)
        print 'On? ',VALUE

    def newState(self, VALUE):
        """
        New control state selected, check that control is turned on.
        """
        onOffSwitch = self.masterOnOff.getValue()
        if onOffSwitch:
            self.Shares.set_GuiMode(shares.TelescopeMode[VALUE])
        else:
            self.stateVar.set('Hold')
            self.Shares.set_GuiMode(shares.TelescopeMode['Hold'])
        self.Shares.set_GuiStatus(self.Shares.get_GuiMode())
        print 'State',VALUE

    def goToPos(self, event=None):
        """
        New set of theta values selected.
        """
        desired = self.getDirectionDesired()
        AZ   = desired['AZ']
        ALT  = desired['ALT']
        ROLL = desired['ROLL']
        if AZ==-99 or ALT==-99 or ROLL==-99:
            myTk.errorWindow(self.parent, 'Values must be floats!')
        elif AZ<-5 or AZ>5:
            myTk.errorWindow(self.parent, 'Azimuth must be in range [-5,5]!')
        elif ALT<10 or ALT>90:
            myTk.errorWindow(self.parent, 'Altitude must be in range [10,90]!')
        elif ROLL<-5 or ROLL>5:
            myTk.errorWindow(self.parent, 'Roll must be in range [-5,5]!')
        else:
            self.setDirectionSetPoint(AZ, ALT, ROLL)
            self.Shares.set_GuiTarget(AZ, ALT, ROLL)
            self.Shares.set_GuiNewTarget(True)
            if self.stateVar.get()=='Run' and self.masterOnOff.getValue():
                self.Shares.set_GuiMode(shares.TelescopeMode['Run'])
```

```
    def resendHome(self, event=None):
        """
        Home button pressed.  Send values of homing positions to controller.
        """
        self.Shares.set_GuiSendHome(True)

    def updateValues(self):
        """
        GUI update function.  Called at 10Hz.
        """
        waypoint = self.Shares.get_GuiWaypoint()
        self.setWayPoint(waypoint['AZ'], waypoint['ALT'], waypoint['ROLL'])
        actual = self.Shares.get_GuiCurrentDirection()
        self.setDirectionActual(actual['AZ'], actual['ALT'], actual['ROLL'])
        l_inch = self.Shares.get_GuiLengthInch()
        self.setLengthInch(l_inch['L1'], l_inch['L2'], l_inch['L3'])
        l_count = self.Shares.get_GuiLengthCount()
        self.setLengthCount(l_count['L1'], l_count['L2'], l_count['L3'])
        status = self.Shares.get_GuiStatus()
        self.setStatusMsg(status)
        tx = self.Shares.get_TxValues()
        self.setLengthCountDes(tx[1], tx[2], tx[3])
        self._after = self.parent.after(100, self.updateValues)

    def getDirectionDesired(self):
        """
        Read from desired direction entries.
        """
        AZ   = self.varDirectionDes[0].get()
        ALT  = self.varDirectionDes[1].get()
        ROLL = self.varDirectionDes[2].get()
        try:
            AZ   = float(AZ)
            ALT  = float(ALT)
            ROLL = float(ROLL)
        except ValueError:
            AZ   = -99
            ALT  = -99
            ROLL = -99
        return {'AZ':AZ, 'ALT':ALT, 'ROLL':ROLL}

    def setDirectionDesired(self, AZ, ALT, ROLL):
        """
        Set desired direction entries.
        """
        self.varDirectionDes[0].set(str(AZ))
        self.varDirectionDes[1].set(str(ALT))
        self.varDirectionDes[2].set(str(ROLL))

    def getDirectionSetPoint(self):
        """
        Read the current set point label.
        """
```

```
        AZ   = self.varDirectionSet[0].get()
        ALT  = self.varDirectionSet[1].get()
        ROLL = self.varDirectionSet[2].get()
        try:
            AZ   = float(AZ)
            ALT  = float(ALT)
            ROLL = float(ROLL)
        except ValueError:
            AZ   = -99
            ALT  = -99
            ROLL = -99
        return {'AZ':AZ, 'ALT':ALT, 'ROLL':ROLL}

    def setDirectionSetPoint(self, AZ, ALT, ROLL):
        """
        Set point label is set according to direction desired entry.
        """
        self.varDirectionSet[0].set(str(AZ))
        self.varDirectionSet[1].set(str(ALT))
        self.varDirectionSet[2].set(str(ROLL))

    def setWayPoint(self, AZ, ALT, ROLL):
        """
        Set the waypoint label.
        """
        self.varWayPoint[0].set(str(AZ))
        self.varWayPoint[1].set(str(ALT))
        self.varWayPoint[2].set(str(ROLL))

    def setDirectionActual(self, AZ, ALT, ROLL):
        """
        Set the actual direction label.
        """
        self.varDirectionAct[0].set(str(AZ))
        self.varDirectionAct[1].set(str(ALT))
        self.varDirectionAct[2].set(str(ROLL))

    def setLengthInch(self, L1, L2, L3):
        """
        Set the actuator length label.
        """
        self.varLen[0].set(float(L1))
        self.varLen[1].set(float(L2))
        self.varLen[2].set(float(L3))

    def setLengthCount(self, L1, L2, L3):
        """
        Set the actuator count label.
        """
        self.varCount[0].set(int(L1))
        self.varCount[1].set(int(L2))
        self.varCount[2].set(int(L3))
```

```
    def setLengthCountDes(self, L1, L2, L3):
        """
        Set the desired actuator count label.
        """
        self.varDesCount[0].set(int(L1))
        self.varDesCount[1].set(int(L2))
        self.varDesCount[2].set(int(L3))

    def setStatusMsg(self, STATUS):
        """
        Set the current status/debug message.
        """
        self.statusMsg.set(STATUS)

class Gui_Test_Thread (threading.Thread):
    """
    Thread simulating messages from the serial_thread to ensure front panel of
    GUI is functional.
    """
    def __init__ (self, SHARES):
        """
        Shares represents variables shared between threads.
        """
        threading.Thread.__init__ (self, name="GuiThread")
        self.event = threading.Event()
        self.kill_thread = False
        self.Shares = SHARES

    def run (self):
        """
        No loop, test only requires a single run through.
        """
        for i in range(5):
            time.sleep(1)
            self.Shares.set_GuiWaypoint(i, i, i)
            self.Shares.set_GuiCurrentDirection(i, i, i)
            self.Shares.set_GuiLengthInch(i, i, i)
            self.Shares.set_GuiLengthCount(i, i, i)
            self.Shares.set_GuiStatus('This is my debug ' + str(i))

    def close_thread (self):
        """
        Match format of other threads, not necessary for test.
        """
        self.kill_thread = True


if __name__ == '__main__':
    self_test = False

    if self_test:
        Shares = shares.Shares()
        tester = Gui_Test_Thread(Shares)
```

```
        tester.daemon = False
        tester.start ()
        time.sleep(0.5)

        root = tk.Tk()
        telescopeGUI(root, Shares)
        root.mainloop()

        logging.debug('Upon closeout, kill it')
        tester.close_thread()
        tester.join()
        print 'Closed'
        time.sleep(3) # to view prompt
        print 'Fin'

    else:
        Shares = shares.Shares()

        """ Initiate serial thread first """
        serialThread = serial_thread.Serial_Thread(0.1, Shares)
        serialThread.daemon = False
        serialThread.start ()

        """ Initiate motor task thread second"""
        telescopeThread = telescope_thread.Telescope_Thread(0.05, Shares)
        telescopeThread.daemon = False
        telescopeThread.start ()

        """ Initiate GUI, which is the mainloop """
        root = tk.Tk()

        telescopeGUI(root, Shares)

        root.mainloop()

        time.sleep(1) # Allow time to send off command

        logging.debug('Upon closeout, kill it')
        serialThread.close_thread()
        serialThread.join()
        telescopeThread.close_thread()
        telescopeThread.join()
        print 'Closed'
        time.sleep(1) # to view prompt
        print 'Fin'
```

B.1.3 *Remaining Classes*

telescope_class.py

```
import math as m
import numpy as np
import tFunctions as tf

class telescope():
    """
    Contains values and calculations for a specific configuration of mount as
    defined by values in the .csv file.
    Important Internal Variables:
    OTA:              3x3 Actuator-OTA connection points
    GND:              3x3 Actuator-GND connection points
    Hardstops:        3x1 Lengths at which the hardstops are present
    phi1/phi2/phi3:   Telescope mount offset variables.
    invPhi1/invPhi23: Inverse transformation matrices used in equation 3.12
    count_per_inch:   Conversion between counts and inches.
    """
    def __init__(self, csvfile):
        """
        File csvfile contains information necessary to define mount locations.
        """
        zero_matrix = np.matrix( np.zeros(shape=[3,3], dtype=float) )
        self.count_per_inch = 500.*4.*31.*16
        self.OTA    = np.matrix(zero_matrix)
        self.invOTA = np.matrix(zero_matrix)
        self.GND    = np.matrix(zero_matrix)
        self.Hardstops = np.zeros(shape=[3,1], dtype=float)
        self.phi1 = 0.
        self.phi2 = 0.
        self.phi3 = 0.
        self.invPhi1  = np.matrix(np.identity(3, dtype=float))
        self.invPhi23 = np.matrix(np.identity(3, dtype=float))
        try:
            my_data = np.genfromtxt(csvfile, delimiter=',', dtype=float)
            my_data = np.delete(my_data, 0, 0)
            my_data = np.delete(my_data, 0, 1)
            my_data = np.transpose(my_data)
            my_data = np.matrix(my_data)
            self.GND[:,0] = my_data[:,1] - my_data[:,0]
            self.GND[:,1] = my_data[:,3] - my_data[:,0]
            self.GND[:,2] = my_data[:,5] - my_data[:,0]
            self.OTA[:,0] = my_data[:,2] - my_data[:,0]
            self.OTA[:,1] = my_data[:,4] - my_data[:,0]
            self.OTA[:,2] = my_data[:,6] - my_data[:,0]
            self.Hardstops = my_data[:,8]
            self.hardpoints = np.matrix(my_data)
            self.invOTA = np.linalg.inv( self.OTA )
        except IOError:
            print 'Failure...'
            return
        self._initLookupTable()
```

```
    def _initLookupTable(self):
        """
        Create lookup table of telescope directions and actuator counts for
        faster calculating of telescope direction from actautor counts.
        Performed during initialization.
        """
        self.lookupTable = []
        for i in range(-30, 30+1, 5):
            for ii in range(0, 105+1, 5):
                for iii in range(-15, 15+1, 5):
                    counts = self.calcActCounts(i, ii, iii)
                    data = [counts['C1'], counts['C2'], counts['C3'],
                            i, ii, iii]
                    self.lookupTable.append(data)

    def findClosestCounts(self, C1, C2, C3):
        """
        Return closest telescope direction based on actuator counts.
        Return -1 if fault occurs.
        """
        closest = -1
        distance = 99999999
        for i in range(len(self.lookupTable)):
            current = self.lookupTable[i]
            newDistance = ( abs(C1-current[0]) + abs(C2-current[1]) +
                            abs(C3-current[2]) )
            if newDistance < distance:
                distance = newDistance
                closest = i
        if closest != -1:
            theta1 = self.lookupTable[closest][3]
            theta2 = self.lookupTable[closest][4]
            theta3 = self.lookupTable[closest][5]
            return theta1, theta2, theta3
        else:
            return closest

    def getHardstopCounts(self):
        """
        Return hardstop position in actuator counts.
        """
        C1 = self.calcInch2Enc( self.Hardstops.item(0) + 0.75/2 )
        C2 = self.calcInch2Enc( self.Hardstops.item(1) + 0.75/2 )
        C3 = self.calcInch2Enc( self.Hardstops.item(2) + 0.75/2 )
        return C1, C2, C3

    def calcActLength(self, X_deg, Y_deg, Z_deg):
        """
        Return actuator lengths necessary to direct the telescope to direction
        theta1 = X_deg, theta2 = Y_deg, theta3 = Z_deg.  Also return actuator
        positions holding the OTA in this new orientation.
        """
        T = tf.RotX(X_deg) * tf.RotY(Y_deg) * tf.RotZ(Z_deg)
```

```
        T = self.invPhi1 * T * self.invPhi23
        newOTA = T*self.OTA
        length1 = np.linalg.norm( newOTA[:,0] - self.GND[:,0] )
        length2 = np.linalg.norm( newOTA[:,1] - self.GND[:,1] )
        length3 = np.linalg.norm( newOTA[:,2] - self.GND[:,2] )
        return {'L1':length1, 'L2':length2, 'L3':length3, 'newOTA':newOTA}

    def calcActCounts(self, X_deg, Y_deg, Z_deg):
        """
        Return actuator counts necessary to direct the telescope to direction
        theta1 = X_deg, theta2 = Y_deg, theta3 = Z_deg.
        """
        length = self.calcActLength(X_deg, Y_deg, Z_deg)
        count1 = int(self.calcInch2Enc(length['L1']))
        count2 = int(self.calcInch2Enc(length['L2']))
        count3 = int(self.calcInch2Enc(length['L3']))
        return {'C1':count1, 'C2':count2, 'C3':count3}

    def calcInch2Enc(self, length):
        """
        Conversion from inches to actuator counts.
        """
        return int(length * self.count_per_inch)

    def calcEnc2Inch(self, count):
        """
        Convertion from actuator counts to inches.
        """
        count = float(count)
        return count / self.count_per_inch

    def updatePhi(self, phi1, phi2, phi3):
        """
        Update telescope offset angles.
        """
        self.phi1 = phi1
        self.phi2 = phi2
        self.phi3 = phi3
        self.invPhi1  = np.linalg.inv( tf.RotX(phi1) )
        self.invPhi23 = np.linalg.inv( tf.RotX(phi2) * tf.RotY(phi3) )

    def calcEnc2Angles(self, C1, C2, C3, X0_deg=0., Y0_deg=45., Z0_deg=0.):
        """
        Calculate mount theta angles from the current actuator counts. Calls on
        function calcTelescopeAngles().
        """
        L1 = self.calcEnc2Inch(C1)
        L2 = self.calcEnc2Inch(C2)
        L3 = self.calcEnc2Inch(C3)

        angles = self.calcTelescopeAngles(L1, L2, L3, X0_deg, Y0_deg, Z0_deg)

        return angles
```

```
    def calcTelescopeAngles(self, L1, L2, L3, X0_deg=0., Y0_deg=45.,
                            Z0_deg=0.):
        """
        Calculate mount theta angles from current actuator lengths.
        Follow procedure from Section 3.2.5.
        """
        if Y0_deg==90:
            Y0_deg=91.

        initialGuess = self.calcActLength(X0_deg, Y0_deg, Z0_deg)
        act10 = initialGuess['newOTA'][:,0]
        act20 = initialGuess['newOTA'][:,1]
        act30 = initialGuess['newOTA'][:,2]
        x1 = np.concatenate( (act10, act20, act30) )
        G1 = np.matrix( np.zeros(shape=[9,1], dtype=float) )
        J1 = np.matrix( np.zeros(shape=[9,9], dtype=float) )

        act1GND = self.GND[:,0]
        act2GND = self.GND[:,1]
        act3GND = self.GND[:,2]
        D12 = np.linalg.norm(act10-act20)
        D13 = np.linalg.norm(act10-act30)
        D23 = np.linalg.norm(act20-act30)
        D1b = np.linalg.norm(act10)
        D2b = np.linalg.norm(act20)
        D3b = np.linalg.norm(act30)

        for i in range(9):
            act11 = x1[0:3,0]
            act21 = x1[3:6,0]
            act31 = x1[6:9,0]

            dist11_1GND = np.linalg.norm(act11-act1GND)
            dist21_2GND = np.linalg.norm(act21-act2GND)
            dist31_3GND = np.linalg.norm(act31-act3GND)
            dist11_21   = np.linalg.norm(act11-act21)
            dist11_31   = np.linalg.norm(act11-act31)
            dist21_31   = np.linalg.norm(act21-act31)
            dist11_b    = np.linalg.norm(act11)
            dist21_b    = np.linalg.norm(act21)
            dist31_b    = np.linalg.norm(act31)

            G1[0,0] = dist11_1GND - L1
            G1[1,0] = dist21_2GND - L2
            G1[2,0] = dist31_3GND - L3
            G1[3,0] = dist11_21 - D12
            G1[4,0] = dist11_31 - D13
            G1[5,0] = dist21_31 - D23
            G1[6,0] = dist11_b - D1b
            G1[7,0] = dist21_b - D2b
            G1[8,0] = dist31_b - D3b
```

```
            # Actuator Lengths
            J1[0,0] = (act11[0]-act1GND[0])/dist11_1GND
            J1[0,1] = (act11[1]-act1GND[1])/dist11_1GND
            J1[0,2] = (act11[2]-act1GND[2])/dist11_1GND

            J1[1,3] = (act21[0]-act2GND[0])/dist21_2GND
            J1[1,4] = (act21[1]-act2GND[1])/dist21_2GND
            J1[1,5] = (act21[2]-act2GND[2])/dist21_2GND

            J1[2,6] = (act31[0]-act3GND[0])/dist31_3GND
            J1[2,7] = (act31[1]-act3GND[1])/dist31_3GND
            J1[2,8] = (act31[2]-act3GND[2])/dist31_3GND

            # Distance between hardpoints
            J1[3,0] = (act11[0]-act21[0])/dist11_21
            J1[3,3] = -J1[3,0]
            J1[3,1] = (act11[1]-act21[1])/dist11_21
            J1[3,4] = -J1[3,1]
            J1[3,2] = (act11[2]-act21[2])/dist11_21
            J1[3,5] = -J1[3,2]

            J1[4,0] = (act11[0]-act31[0])/dist11_31
            J1[4,6] = -J1[4,0]
            J1[4,1] = (act11[1]-act31[1])/dist11_31
            J1[4,7] = -J1[4,1]
            J1[4,2] = (act11[2]-act31[2])/dist11_31
            J1[4,8] = -J1[4,2]

            J1[5,3] = (act21[0]-act31[0])/dist21_31
            J1[5,6] = -J1[5,3]
            J1[5,4] = (act21[1]-act31[1])/dist21_31
            J1[5,7] = -J1[5,4]
            J1[5,5] = (act21[2]-act31[2])/dist21_31
            J1[5,8] = -J1[5,5]

            # Distance to Base
            J1[6,0] = (act11[0])/dist11_b
            J1[6,1] = (act11[1])/dist11_b
            J1[6,2] = (act11[2])/dist11_b

            J1[7,3] = (act21[0])/dist21_b
            J1[7,4] = (act21[1])/dist21_b
            J1[7,5] = (act21[2])/dist21_b

            J1[8,6] = (act31[0])/dist31_b
            J1[8,7] = (act31[1])/dist31_b
            J1[8,8] = (act31[2])/dist31_b

            delta = np.linalg.lstsq(J1,G1)[0]
            x1 = x1 - delta

        newOTA = np.concatenate( ( x1[0:3,0], x1[3:6], x1[6:9] ), axis=1 )
        T  = newOTA * self.invOTA
```

```
        x2 = np.matrix( [ [float(X0_deg)], [float(Y0_deg)], [float(Z0_deg)] ] )
        G2 = np.matrix( np.zeros(shape=[9,1], dtype=float) )
        J2 = np.matrix( np.zeros(shape=[9,3], dtype=float) )

        x2[0,0] = m.radians(float(x2[0,0]))
        x2[1,0] = m.radians(float(x2[1,0]))
        x2[2,0] = m.radians(float(x2[2,0]))

        for i in range(15):
            th1 = x2[0,0]
            th2 = x2[1,0]
            th3 = x2[2,0]

            G2[0,0] = -T[0,0] + m.cos(th2)*m.cos(th3)
            G2[1,0] = -T[0,1] - m.cos(th2)*m.sin(th3)
            G2[2,0] = -T[0,2] + m.sin(th2)

            G2[3,0] =(-T[1,0] + m.cos(th1)*m.sin(th3) +
                       m.sin(th1)*m.sin(th2)*m.cos(th3) )
            G2[4,0] =(-T[1,1] + m.cos(th1)*m.cos(th3) -
                       m.sin(th1)*m.sin(th2)*m.sin(th3) )
            G2[5,0] = -T[1,2] - m.sin(th1)*m.cos(th2)

            G2[6,0] =(-T[2,0] + m.sin(th1)*m.sin(th3) -
                       m.cos(th1)*m.sin(th2)*m.cos(th3) )
            G2[7,0] =(-T[2,1] + m.sin(th1)*m.cos(th3) +
                       m.cos(th1)*m.sin(th2)*m.sin(th3) )
            G2[8,0] = -T[2,2] + m.cos(th1)*m.cos(th2)

            J2[0,0] =  0.
            J2[0,1] = -m.sin(th2)*m.cos(th3)
            J2[0,2] = -m.cos(th2)*m.sin(th3)

            J2[1,0] =  0.
            J2[1,1] =  m.sin(th2)*m.sin(th3)
            J2[1,2] = -m.cos(th2)*m.cos(th3)

            J2[2,0] =  0.
            J2[2,1] =  m.cos(th2)
            J2[2,2] =  0.

            J2[3,0] = -m.sin(th1)*m.sin(th3) + m.cos(th1)*m.sin(th2)*m.cos(th3)
            J2[3,1] =  m.sin(th1)*m.cos(th2)*m.cos(th3)
            J2[3,2] =  m.cos(th1)*m.cos(th3) - m.sin(th1)*m.sin(th2)*m.sin(th3)

            J2[4,0] = -m.sin(th1)*m.cos(th3) - m.cos(th1)*m.sin(th2)*m.sin(th3)
            J2[4,1] = -m.sin(th1)*m.cos(th2)*m.sin(th3)
            J2[4,2] = -m.cos(th1)*m.sin(th3) - m.sin(th1)*m.sin(th2)*m.cos(th3)

            J2[5,0] = -m.cos(th1)*m.cos(th2)
            J2[5,1] =  m.sin(th1)*m.sin(th2)
            J2[5,2] =  0.
```

```
            J2[6,0] =  m.cos(th1)*m.sin(th3) + m.sin(th1)*m.sin(th2)*m.cos(th3)
            J2[6,1] = -m.cos(th1)*m.cos(th2)*m.cos(th3)
            J2[6,2] =  m.sin(th1)*m.cos(th3) + m.cos(th1)*m.sin(th2)*m.sin(th3)

            J2[7,0] =  m.cos(th1)*m.cos(th3) - m.sin(th1)*m.sin(th2)*m.sin(th3)
            J2[7,1] =  m.cos(th1)*m.cos(th2)*m.sin(th3)
            J2[7,2] = -m.sin(th1)*m.sin(th3) + m.cos(th1)*m.sin(th2)*m.cos(th3)

            J2[8,0] = -m.sin(th1)*m.cos(th2)
            J2[8,1] = -m.cos(th1)*m.sin(th2)
            J2[8,2] =  0.

            if np.linalg.norm(G2) < 0.00000000001:
                delta = np.matrix( np.zeros(shape=[3,1], dtype=float) )
            else:
                delta = np.linalg.lstsq(J2,G2)[0]
            x2 = x2 - delta

            for ii in range(3):
                while x2[ii] > m.pi:
                    x2[ii] = x2[ii] - 2*m.pi
                while x2[ii] <= -m.pi:
                    x2[ii] = x2[ii] + 2*m.pi

            theta1theta3 = [0, 2]
            for ii in theta1theta3:
                while x2[ii] > m.pi/2:
                    x2[ii] = x2[ii] - m.pi
                while x2[ii] <= -m.pi/2:
                    x2[ii] = x2[ii] + m.pi

            if False:
                print i
                print x2
                print m.degrees(x2[0,0])
                print m.degrees(x2[1,0])
                print m.degrees(x2[2,0])


        THETA1 = round(m.degrees(x2[0,0]),5)
        THETA2 = round(m.degrees(x2[1,0]),5)
        THETA3 = round(m.degrees(x2[2,0]),5)

        THETA1,THETA2,THETA3 = self._redefineAngles(X0_deg, Y0_deg, Z0_deg,
                                                    THETA1, THETA2, THETA3)

        return {'THETA1':THETA1, 'THETA2':THETA2, 'THETA3':THETA3,
                'newOTA':newOTA}

    def _redefineAngles(self, GUESS1, GUESS2, GUESS3, THETA1, THETA2, THETA3):
        """
        Mount angle definitions show degeneracy if theta2=90 because theta1 and
```

```
        theta 3 will share the same rotational vector.  If theta2=90, function
        alters theta1 and theta3 such that theta1 is the desired guess and
        theta3 is the remainder.
        """
        if THETA2 == 90:
            THETA3 = THETA1 - GUESS1 + THETA3
            THETA1 = GUESS1

        return THETA1, THETA2, THETA3

    def getActuatorGrounds(self):
        """
        Return actuator ground points.
        """
        return self.GND

def _Example():
    """
    Example of module functions.
    """
    import time

    print "Init\n"

    OTA = telescope('telescope.csv')

    theta1 = 15
    theta2 = 90
    theta3 = -10

    guess1 = 20
    guess2 = 90
    guess3 = -10

    lengths = OTA.calcActLength(theta1,theta2,theta3)

    startT = time.clock()
    angles = OTA.calcTelescopeAngles(lengths['L1'],
                lengths['L2'], lengths['L3'],
                guess1, guess2, guess3)
    endT   = time.clock()

    ang1 = angles['THETA1']
    ang2 = angles['THETA2']
    ang3 = angles['THETA3']

    diff1 = abs(ang1-theta1)
    diff2 = abs(ang2-theta2)
    diff3 = abs(ang3-theta3)

    print "Actual: ", theta1, theta2, theta3
    print "Guess:  ", guess1, guess2, guess3
    print "Calced: ", ang1,   ang2,   ang3
```

```
    print "Diff:   ", diff1,  diff2,  diff3
    print "Time:   ", endT-startT

    print "\n"

    hs = OTA.getHardstopCounts()

    print 'Hardstops: ', hs[0], hs[1], hs[2]

    print 'Calc Hardstop: ', OTA.calcEnc2Angles(hs[0], hs[1], hs[2], 0, 0, 0)

    theta1 = 30
    theta2 = 84
    theta3 = 14

    counts = OTA.calcActCounts(theta1, theta2, theta3)

    startT = time.clock()
    guesses = OTA.findClosestCounts(counts['C1'], counts['C2'], counts['C3'])
    endT   = time.clock()

    print "Actual: ", theta1, theta2, theta3
    print "Close:  ", guesses
    print "Time:   ", endT-startT

    lengths = [OTA.calcEnc2Inch(counts['C1']), OTA.calcEnc2Inch(counts['C2']),
               OTA.calcEnc2Inch(counts['C3'])]

    startT = time.clock()
    angles = OTA.calcTelescopeAngles(lengths[0],
                lengths[1], lengths[2],
                guesses[0], guesses[1], guesses[2])
    endT   = time.clock()

    ang1 = angles['THETA1']
    ang2 = angles['THETA2']
    ang3 = angles['THETA3']

    diff1 = abs(ang1-theta1)
    diff2 = abs(ang2-theta2)
    diff3 = abs(ang3-theta3)

    print "Actual: ", theta1, theta2, theta3
    print "Guess:  ", guesses
    print "Calced: ", ang1,   ang2,   ang3
    print "Diff:   ", diff1,  diff2,  diff3
    print "Time:   ", endT-startT

if __name__ == '__main__':
    _Example()
```

tFunctions.py

```
import numpy
import math

def RotX(deg):
    """
    Return Reference Frame Transformation about x-axis.
    """
    rad = math.radians(deg)
    T = numpy.matrix([(1., 0.,             0.),
                      (0., math.cos(rad), -math.sin(rad)),
                      (0., math.sin(rad),  math.cos(rad))])
    return T

def RotY(deg):
    """
    Return Reference Frame Transformation about y-axis.
    """
    rad = math.radians(deg)
    T = numpy.matrix([(math.cos(rad),  0., math.sin(rad)),
                      (0.,             1., 0.),
                      (-math.sin(rad), 0., math.cos(rad))])
    return T

def RotZ(deg):
    """
    Return Reference Frame Transformation about z-axis.
    """
    rad = math.radians(deg)
    T = numpy.matrix([(math.cos(rad), -math.sin(rad), 0.),
                      (math.sin(rad),  math.cos(rad), 0.),
                      (0.,             0.,            1.)])
    return T

def getWaypoints(current, desired, increment):
    """
    Return path between current and desired locations using increment.
    """
    desiredAz   = desired[0]
    desiredAlt  = desired[1]
    desiredRoll = desired[2]
    nextAz   = current[0]
    nextAlt  = current[1]
    nextRoll = current[2]
    waypoints = []
    nextWaypoint = []

    condition = True
    while condition:
        nextAz   = _getWaypointValue(nextAz,   desiredAz,   increment)
        nextAlt  = _getWaypointValue(nextAlt,  desiredAlt,  increment)
        nextRoll = _getWaypointValue(nextRoll, desiredRoll, increment)
        nextWaypoint = [nextAz, nextAlt, nextRoll]
        waypoints.append(nextWaypoint)
```

```
        condition = (math.fabs(desiredAz   - nextAz)   > 0. or
                     math.fabs(desiredAlt  - nextAlt)  > 0. or
                     math.fabs(desiredRoll - nextRoll) > 0.)
    return waypoints

def _getWaypointValue(current, desired, increment):
    """
    Find next waypoint between current and desired using maximum increment.
    Used only by function getWaypoints(...).
    """
    if desired != current:
        if desired > current:
            if desired - current > increment:
                waypoint = (current+increment)
            else:
                waypoint = (desired)
        else:
            if current - desired > increment:
                waypoint = (current-increment)
            else:
                waypoint = (desired)
    else:
        waypoint = (desired)
    return waypoint

def _Example():
    """
    Example to demonstrate included functions.
    """
    import tFunctions

    X1 = tFunctions.RotX(30.)
    X2 = tFunctions.RotX(-30.)
    Y1 = tFunctions.RotY(30.)
    Y2 = tFunctions.RotY(120.)
    Z1 = tFunctions.RotZ(15.)
    Z2 = tFunctions.RotZ(-15.)

    print "X1:\n",X1
    print "X2:\n",X2
    print "Y1:\n",Y1
    print "Y2:\n",Y2
    print "Z1:\n",Z1
    print "Z2:\n",Z2
    print "T1:\n",X1*Y1*Z1
    print "T2:\n",X2*Y2*Z2
    print tFunctions.getWaypoints( [0,0,0], [5,4,-1], 1)


if __name__ == "__main__":
    _Example()
```

myTk.py

```
import Tkinter as tk
import time
import sys

"""
Regularly used custom Tkinter commands.
"""

def center(win):
    """
    Center the window on screen."
    """
    win.update_idletasks()
    width = win.winfo_width()
    frm_width = win.winfo_rootx() - win.winfo_x()
    win_width =  width + 2 * frm_width
    height = win.winfo_height()
    titlebar_height = win.winfo_rooty() - win.winfo_y()
    win_height = height + titlebar_height + frm_width
    x = win.winfo_screenwidth() // 2 - win_width // 2
    y = win.winfo_screenheight() // 2 - win_height // 2
    win.geometry('{}x{}+{}+{}'.format(width, height, x, y))
    win.geometry('')
    win.deiconify()

class diologue():
    """
    Dialogue box object.  Displays message until closed.
    """
    def __init__(self, PARENT, WIN_NAME, MESSAGE):
        self.top = tk.Toplevel(PARENT)
        self.top.wm_title(WIN_NAME)
        self.top.transient(PARENT)
        self.top.grab_set()
        self.top.focus_set()
        tk.Label(self.top, text=MESSAGE).pack()
        center(self.top)
        self.top.resizable(width=False, height=False)
        self.top.bind("<Return>", self.close)
        self.top.bind("<Escape>", self.close)

    def close(self, event=None):
        self.top.destroy()

class errorWindow():
    """
    Error window object.  Displays error message until closed.
    """
    def __init__(self, parent, MESSAGE):
        self.top = tk.Toplevel(parent)
        self.top.wm_title('Error')
        self.top.transient(parent)
        self.top.grab_set()
```

```
        self.top.focus_set()
        tk.Label(self.top, text=MESSAGE).pack()
        center(self.top)
        self.top.resizable(width=False, height=False)
        self.top.bind("<Return>", self.close)
        self.top.bind("<Escape>", self.close)

    def close(self, event=None):
        self.top.destroy()

def goodbye(WINDOW, MESSAGE):
    """
    Optional close out command ensuring GUI and threads close.
    """
    print MESSAGE
    ### Destroy GUI to prevent startum errors on next run ###
    WINDOW.destroy()
    ### Wait 5 seconds to allow user to read message before closing ###
    time.sleep(5)
    sys.exit('Goodbye')

class ToggleButton():
    """
    Button which toggles value after button release.  On and off.
    """
    def __init__(self, master, width=None, height=None, command=None):
        self.parent = master
        self.state  = False
        self.stateVar = tk.StringVar()
        self.stateVar.set('Off')
        self.button = tk.Button(master, textvariable=self.stateVar,
                            command=self.__toggle, width=width, height=height)
        self.command = command

    def grid(self, row=None, column=None, rowspan=1, columnspan=1,
             sticky=tk.NW+tk.SE, padx=0, pady=0):
        self.button.grid(row=row, column=column, rowspan=rowspan,
                         columnspan=columnspan, sticky=sticky, padx=padx,
                         pady=pady)

    def pack(self, side=None, anchor=None):
        self.button.pack(side=side, anchor=anchor)

    def setSwitch(self, newState):
        self.state = newState
        if newState:
            self.stateVar.set('On')
        else:
            self.stateVar.set('Off')

    def getValue(self):
        return self.state
```

```
    def getStateVar(self):
        return self.stateVar.get()

    def __toggle(self):
        self.state = not self.state
        if self.state:
            self.stateVar.set('On')
        else:
            self.stateVar.set('Off')
        self.command(self.stateVar.get())

    def destroy(self):
        self.button.destroy()
```

shares.py

```
import threading

TelescopeMode = {'Off':0, 'Calibrate':1, 'CalibrateDone':2, 'Run':3, 'Hold':4}

class Shares():
    """
    Single instance is shared between all threads and internal variables are
    thread protected allowing safe data sharing.
    """
    def __init__(self):
        """
        Initialize all shared variables.
        """
        # GUI Output Variables-------------------------------------------------
        # GuiOnOff
        self.onOffSwitch = True
        self.GuiOnOff_dataLock = threading.Lock()

        # GuiMode
        self.guiMode    = 0
        self.GuiMode_dataLock = threading.Lock()

        # GuiTarget
        self.targetAz   = 0.0
        self.targetAlt  = 0.0
        self.targetRoll = 0.0
        self.GuiTarget_dataLock = threading.Lock()

        # GuiNewTarget
        self.newTargetFlag = False
        self.GuiNewTarget_dataLock = threading.Lock()

        # GuiSendHome
        self.sendHomeFlag = False
        self.GuiSendHome_dataLock = threading.Lock()

        # GUI Input Variables--------------------------------------------------
        # GuiWaypoint
        self.waypointAz           = 0.0
        self.waypointAlt          = 0.0
        self.waypointRoll         = 0.0
        self.GuiWaypoint_dataLock = threading.Lock()

        # GuiCurrentDirection
        self.currentAz   = 0.0
        self.currentAlt  = 0.0
        self.currentRoll = 0.0
        self.GuiCurrentDirection_dataLock = threading.Lock()

        # GuiLengthInch
        self.currentLength1_inch   = 0.0
        self.currentLength2_inch   = 0.0
        self.currentLength3_inch   = 0.0
```

```
        self.GuiLengthInch_dataLock = threading.Lock()

        # GuiLengthCount
        self.currentLength1_count = 0
        self.currentLength2_count = 0
        self.currentLength3_count = 0
        self.GuiLengthCount_dataLock = threading.Lock()

        # GuiStatus
        self.guiStatus = ""
        self.GuiStatus_dataLock = threading.Lock()

        # Tx Desired Mode and Actuator Values
        self.modeTx = 0
        self.act1Tx = 0
        self.act2Tx = 0
        self.act3Tx = 0
        self.Tx_dataLock = threading.Lock()

        # Tx Hardstop points
        self.hardstop1Tx = 0
        self.hardstop2Tx = 0
        self.hardstop3Tx = 0
        self.TxHardstop_dataLock = threading.Lock()

        # Rx Mode, Calibration Flag, and Actuator Values
        self.modeRx       = 0
        self.calibratedRx = 0
        self.homeRx       = 0
        self.act1Rx       = 0
        self.act2Rx       = 0
        self.act3Rx       = 0
        self.Rx_dataLock = threading.Lock()

    ##The following get and set functions are to be used, not the variables

    # GuiOut-------------------------------------------------------------------
    # GuiOnOff
    def get_GuiOnOff (self):
        onOffSwitch = 'Error'
        try:
            self.GuiOnOff_dataLock.acquire()
            onOffSwitch = self.onOffSwitch
        finally:
            self.GuiOnOff_dataLock.release()
        return onOffSwitch

    def set_GuiOnOff (self, onOffSwitch):
        try:
            self.GuiOnOff_dataLock.acquire()
            self.onOffSwitch = bool(onOffSwitch)
        finally:
            self.GuiOnOff_dataLock.release()
```

```
    # GuiMode
    def get_GuiMode (self):
        guiMode    = 'Error'
        try:
            self.GuiMode_dataLock.acquire()
            guiMode = self.guiMode
        finally:
            self.GuiMode_dataLock.release()
        return guiMode

    def set_GuiMode (self, guiMode):
        try:
            self.GuiMode_dataLock.acquire()
            self.guiMode = int(guiMode)
        finally:
            self.GuiMode_dataLock.release()

    # GuiTarget
    def get_GuiTarget (self):
        targetAz   = 'Error'
        targetAlt  = 'Error'
        targetRoll = 'Error'
        try:
            self.GuiTarget_dataLock.acquire()
            targetAz   = self.targetAz
            targetAlt  = self.targetAlt
            targetRoll = self.targetRoll
        finally:
            self.GuiTarget_dataLock.release()
        return {'AZ':targetAz, 'ALT':targetAlt, 'ROLL':targetRoll}

    def set_GuiTarget (self, targetAz, targetAlt, targetRoll):
        try:
            self.GuiTarget_dataLock.acquire()
            self.targetAz   = float(targetAz)
            self.targetAlt  = float(targetAlt)
            self.targetRoll = float(targetRoll)
        finally:
            self.GuiTarget_dataLock.release()

    # GuiNewTarget
    def get_GuiNewTarget (self):
        flag   = 'Error'
        try:
            self.GuiNewTarget_dataLock.acquire()
            flag   = self.newTargetFlag
        finally:
            self.GuiNewTarget_dataLock.release()
        return flag

    def set_GuiNewTarget (self, flag):
        try:
```

```
            self.GuiTarget_dataLock.acquire()
            self.newTargetFlag = bool(flag)
        finally:
            self.GuiTarget_dataLock.release()

    # GuiSendHome
    def get_GuiSendHome (self):
        flag   = 'Error'
        try:
            self.GuiSendHome_dataLock.acquire()
            flag   = self.sendHomeFlag
        finally:
            self.GuiSendHome_dataLock.release()
        return flag

    def set_GuiSendHome (self, flag):
        try:
            self.GuiSendHome_dataLock.acquire()
            self.sendHomeFlag = bool(flag)
        finally:
            self.GuiSendHome_dataLock.release()

    # GuiIn----------------------------------------------------------------------
    # GuiWaypoint----------------------------------------------------------------
    def get_GuiWaypoint (self):
        waypointAz           = 'Error'
        waypointAlt          = 'Error'
        waypointRoll         = 'Error'
        try:
            self.GuiWaypoint_dataLock.acquire()
            waypointAz   = self.waypointAz
            waypointAlt  = self.waypointAlt
            waypointRoll = self.waypointRoll
        finally:
            self.GuiWaypoint_dataLock.release()
        return {'AZ':waypointAz, 'ALT':waypointAlt, 'ROLL':waypointRoll}

    def set_GuiWaypoint (self, AZIM, ALTI, ROLL):
        try:
            self.GuiWaypoint_dataLock.acquire()
            self.waypointAz   = float(AZIM)
            self.waypointAlt  = float(ALTI)
            self.waypointRoll = float(ROLL)
        finally:
            self.GuiWaypoint_dataLock.release()

    # GuiCurrentDirection--------------------------------------------------------
    def get_GuiCurrentDirection (self):
        currentAz            = 'Error'
        currentAlt           = 'Error'
        currentRoll          = 'Error'
        try:
            self.GuiCurrentDirection_dataLock.acquire()
```

```
            currentAz           = self.currentAz
            currentAlt          = self.currentAlt
            currentRoll         = self.currentRoll
        finally:
            self.GuiCurrentDirection_dataLock.release()
        return {'AZ':currentAz, 'ALT':currentAlt, 'ROLL':currentRoll}

    def set_GuiCurrentDirection (self, AZIM, ALTI, ROLL):
        try:
            self.GuiCurrentDirection_dataLock.acquire()
            self.currentAz   = float(AZIM)
            self.currentAlt  = float(ALTI)
            self.currentRoll = float(ROLL)
        finally:
            self.GuiCurrentDirection_dataLock.release()

    # GuiLengthInch-------------------------------------------------------------
    def get_GuiLengthInch (self):
        currentLength1_inch  = 'Error'
        currentLength2_inch  = 'Error'
        currentLength3_inch  = 'Error'
        try:
            self.GuiLengthInch_dataLock.acquire()
            currentLength1_inch  = self.currentLength1_inch
            currentLength2_inch  = self.currentLength2_inch
            currentLength3_inch  = self.currentLength3_inch
        finally:
            self.GuiLengthInch_dataLock.release()
        return {'L1':currentLength1_inch,
                'L2':currentLength2_inch,
                'L3':currentLength3_inch}

    def set_GuiLengthInch (self, L1, L2, L3):
        try:
            self.GuiLengthInch_dataLock.acquire()
            self.currentLength1_inch = float(L1)
            self.currentLength2_inch = float(L2)
            self.currentLength3_inch = float(L3)
        finally:
            self.GuiLengthInch_dataLock.release()

    # GuiLengthCount------------------------------------------------------------
    def get_GuiLengthCount (self):
        currentLength1_count = 'Error'
        currentLength2_count = 'Error'
        currentLength3_count = 'Error'
        try:
            self.GuiLengthCount_dataLock.acquire()
            currentLength1_count = self.currentLength1_count
            currentLength2_count = self.currentLength2_count
            currentLength3_count = self.currentLength3_count
        finally:
            self.GuiLengthCount_dataLock.release()
```

```
        return {'L1':currentLength1_count,
                'L2':currentLength2_count,
                'L3':currentLength3_count}

    def set_GuiLengthCount (self, L1, L2, L3):
        try:
            self.GuiLengthCount_dataLock.acquire()
            self.currentLength1_count = float(L1)
            self.currentLength2_count = float(L2)
            self.currentLength3_count = float(L3)
        finally:
            self.GuiLengthCount_dataLock.release()

    # GuiStatus-------------------------------------------------------------------
    def get_GuiStatus (self):
        status = ''
        try:
            self.GuiStatus_dataLock.acquire()
            status = self.guiStatus
        finally:
            self.GuiStatus_dataLock.release()
        return status

    def set_GuiStatus (self, STATUS):
        try:
            self.GuiStatus_dataLock.acquire()
            self.guiStatus = STATUS
        finally:
            self.GuiStatus_dataLock.release()

    # TX Values-------------------------------------------------------------------
    def get_TxValues (self):
        mode = -1
        act1 = -1
        act2 = -1
        act3 = -1
        try:
            self.Tx_dataLock.acquire()
            mode = self.modeTx
            act1 = self.act1Tx
            act2 = self.act2Tx
            act3 = self.act3Tx
        finally:
            self.Tx_dataLock.release()
        return mode, act1, act2, act3

    def set_TxValues (self, MODE, ACT1_ENC, ACT2_ENC, ACT3_ENC):
        try:
            self.Tx_dataLock.acquire()
            self.modeTx = int(MODE)
            self.act1Tx = int(ACT1_ENC)
            self.act2Tx = int(ACT2_ENC)
            self.act3Tx = int(ACT3_ENC)
```

```
        finally:
            self.Tx_dataLock.release()

    # TxHardstop Values----------------------------------------------------------------
    def get_TxHardstopValues (self):
        act1 = -1
        act2 = -1
        act3 = -1
        try:
            self.TxHardstop_dataLock.acquire()
            act1 = self.hardstop1Tx
            act2 = self.hardstop2Tx
            act3 = self.hardstop3Tx
        finally:
            self.TxHardstop_dataLock.release()
        return act1, act2, act3

    def set_TxHardstopValues (self, ACT1_ENC, ACT2_ENC, ACT3_ENC):
        try:
            self.TxHardstop_dataLock.acquire()
            self.hardstop1Tx = int(ACT1_ENC)
            self.hardstop2Tx = int(ACT2_ENC)
            self.hardstop3Tx = int(ACT3_ENC)
        finally:
            self.TxHardstop_dataLock.release()

    # RX Values------------------------------------------------------------------
    def get_RxValues (self):
        mode       = -1
        calibrated = -1
        homeRec    = -1
        act1       = -1
        act2       = -1
        act3       = -1
        try:
            self.Rx_dataLock.acquire()
            mode       = self.modeRx
            calibrated = self.calibratedRx
            homeRec    = self.homeRx
            act1       = self.act1Rx
            act2       = self.act2Rx
            act3       = self.act3Rx
        finally:
            self.Rx_dataLock.release()
        return {'MODE':mode, 'CALIBRATED':calibrated, 'HOMERX':homeRec,
                'L1':act1, 'L2':act2, 'L3':act3}

    def set_RxValues (self, MODE, CALIBRATED, HOMERX,
                      ACT1_ENC, ACT2_ENC, ACT3_ENC):
        try:
            self.Rx_dataLock.acquire()
            self.modeRx       = int(MODE)
            self.calibratedRx = int(CALIBRATED)
```

```
            self.homeRx       = int(HOMERX)
            self.act1Rx       = int(ACT1_ENC)
            self.act2Rx       = int(ACT2_ENC)
            self.act3Rx       = int(ACT3_ENC)
        finally:
            self.Rx_dataLock.release()
```

telescope_thread.py

```
import time
import threading
import logging
import telescope_class
import tFunctions as tf
import shares
import math

_TelescopeThreadState  = {'Init':0, 'Kill':1, 'Hold':2, 'Calibrate':3, 'Run':4}
_rTelescopeThreadState = {0:'Init', 1:'Kill', 2:'Hold', 3:'Calibrate', 4:'Run'}

class Telescope_Thread (threading.Thread):
    """
    Thread responsible for telescope control.  Interprets user input from the
    GUI, directs the serial thread to communicate new positions and commands to
    the motor controller, and interprets feedback from the controller.
    """
    def __init__ (self, run_interval, Shares):
        threading.Thread.__init__ (self, name="TelscopeThread")

        self.event = threading.Event()

        self.kill_thread = False

        self.Shares = Shares

        self.telescope = telescope_class.telescope('telescope.csv')

        # Save the starting value of the time interval between data points
        self.run_interval = run_interval

        # Initialize thread input variables to be received
        self.rx_onOffSwitch  = False
        self.rx_modeDesired = -1
        self.rx_modeActual  = -1
        self.rx_calibrated  = False
        self.rx_homeRx      = False
        self.rx_count1      = -1
        self.rx_count2      = -1
        self.rx_count3      = -1
        self.rx_newTarget   = False

        # Record the starting time, and set it as the most recent
        # and next run times
        self.start_time = time.time ()
        self.next_run_time = self.start_time
        self.last_run_time = self.start_time

    def run (self):
        """
        Thread loop.
        """
        # Zero State - Initialization
```

```
        self.state = _TelescopeThreadState['Init']
        firstRun = True

        waypoints = []
        currentX  = 0.
        currentY  = 0.
        currentZ  = 0.
        hs = self.telescope.getHardstopCounts()
        stateStartTime = time.time()

        while (not self.kill_thread):
            # Check if it's time to do stuff; if so, do it and update the timer
            now_time = time.time ()
            if (now_time >= self.next_run_time):
                self.next_run_time += self.run_interval

                # Always update variables:
                self.update_inputs()

                if ( self.rx_count1>(hs[0]-100) and self.rx_count2>(hs[1]-100)
                     and self.rx_count3>(hs[2]-100) ):
                    L1 = self.telescope.calcEnc2Inch(self.rx_count1)
                    L2 = self.telescope.calcEnc2Inch(self.rx_count2)
                    L3 = self.telescope.calcEnc2Inch(self.rx_count3)

                    angles = self.telescope.calcTelescopeAngles(
                        L1, L2, L3, currentX, currentY, currentZ)
                    currentX = angles['THETA1']
                    currentY = angles['THETA2']
                    currentZ = angles['THETA3']

                # State Machine---------------------------------------------------

                # Init state:
                # Set hardstop locations, transition to kill state
                if self.state == _TelescopeThreadState['Init']:
                    if firstRun:
                        hs = self.telescope.getHardstopCounts()
                        self.Shares.set_TxHardstopValues(hs[0], hs[1], hs[2])
                        firstRun = False
                    elif self.rx_homeRx:
                        self.state = _TelescopeThreadState['Kill']
                        firstRun = True

                # Kill state:
                # Turn off motors
                elif self.state == _TelescopeThreadState['Kill']:
                    if firstRun:
                        self.Shares.set_TxValues(
                            shares.TelescopeMode['Off'], 0, 0, 0)
                        firstRun = False
                    elif self.rx_onOffSwitch:
                        self.state = _TelescopeThreadState['Hold']
```

```
                self.Shares.set_TxValues(
                    shares.TelescopeMode['Run'], self.rx_count1,
                    self.rx_count2, self.rx_count3)
                firstRun = True
            else:
                pass # Do nothing

        # Hold state:
        # Hold at location defined by previous state
        # From killed: hold current count
        # From calibrate: 1 revolution from calibration point (62000)
        # From Run: last target point
        elif self.state == _TelescopeThreadState['Hold']:
            if firstRun:
                firstRun = False
            elif not self.rx_onOffSwitch:
                self.state = _TelescopeThreadState['Kill']
                firstRun = True
            elif (self.rx_modeDesired ==
                  shares.TelescopeMode['Calibrate']):
                self.state = _TelescopeThreadState['Calibrate']
                firstRun = True
            elif (self.rx_calibrated and
                  self.rx_modeDesired == shares.TelescopeMode['Run']):
                self.state = _TelescopeThreadState['Run']
                firstRun = True

        # Calibrate state:
        # Slowly draw lead screws until all three stop moving
        elif self.state == _TelescopeThreadState['Calibrate']:
            if firstRun:
                self.Shares.set_TxValues(
                    shares.TelescopeMode['Calibrate'], 0, 0, 0)
                stateStartTime = time.time()
                firstRun = False
            elif not self.rx_onOffSwitch:
                self.state = _TelescopeThreadState['Kill']
                firstRun = True
            elif time.time() - stateStartTime < 1:
                pass
            elif self.rx_calibrated:
                self.state = _TelescopeThreadState['Hold']
                lengths = self.telescope.calcActLength(0, 10, 0)
                C1 = self.telescope.calcInch2Enc(lengths['L1'])
                C2 = self.telescope.calcInch2Enc(lengths['L2'])
                C3 = self.telescope.calcInch2Enc(lengths['L3'])
                self.Shares.set_TxValues(shares.TelescopeMode['Run'],
                                         C1, C2, C3)
                self.Shares.set_GuiMode(shares.TelescopeMode['Hold'])
                currentX = 0.
                currentY = 10.
                currentZ = 0.
                firstRun = True
```

```
            elif self.rx_modeDesired == shares.TelescopeMode['Hold']:
                self.state = _TelescopeThreadState['Hold']
                self.Shares.set_TxValues(
                    shares.TelescopeMode['Run'], self.rx_count1,
                    self.rx_count2, self.rx_count3)
                firstRun = True

        # Run state:
        # Calculate path
        elif self.state == _TelescopeThreadState['Run']:
            if firstRun:
                target = self.Shares.get_GuiTarget()
                desired = [target['AZ'], target['ALT'], target['ROLL']]
                currentLoc = [currentX, currentY, currentZ]
                waypoints = tf.getWaypoints(currentLoc, desired, 1.)
                tolerance = 0.01
                nextWaypoint = True
                print currentLoc
                self.Shares.set_GuiNewTarget(False)
                firstRun = False
            elif not self.rx_onOffSwitch:
                self.state = _TelescopeThreadState['Kill']
                firstRun = True
            elif (self.rx_modeDesired ==
                  shares.TelescopeMode['Calibrate']):
                self.state = _TelescopeThreadState['Calibrate']
                firstRun = True
            elif self.rx_newTarget:
                firstRun = True
            elif nextWaypoint:
                print waypoints
                nextWaypoint = False
                if waypoints:
                    x = waypoints[0][0]
                    y = waypoints[0][1]
                    z = waypoints[0][2]
                    lengths = self.telescope.calcActLength(x, y, z)
                    C1 = self.telescope.calcInch2Enc(lengths['L1'])
                    C2 = self.telescope.calcInch2Enc(lengths['L2'])
                    C3 = self.telescope.calcInch2Enc(lengths['L3'])
                    self.Shares.set_TxValues(
                        shares.TelescopeMode['Run'], C1, C2, C3)
                else:
                    self.state = _TelescopeThreadState['Hold']
                    self.Shares.set_TxValues(
                        shares.TelescopeMode['Run'], C1, C2, C3)
                    self.Shares.set_GuiMode(
                        shares.TelescopeMode['Hold'])
                    firstRun = True
            else:
                desired = waypoints[0]
                cond1 = math.fabs(desired[0] - currentX) < tolerance
                cond2 = math.fabs(desired[1] - currentY) < tolerance
```

```
                        cond3 = math.fabs(desired[2] - currentZ) < tolerance

                        if cond1 and cond2 and cond3:
                            nextWaypoint = True
                            del waypoints[0]

                    self.update_outputs(waypoints, currentX, currentY, currentZ)
                # Sleep until the next time this task is supposed to run
                time.sleep (self.run_interval / 10.0)

    def update_inputs (self):
        rx = self.Shares.get_RxValues()
        self.rx_onOffSwitch  = self.Shares.get_GuiOnOff()
        self.rx_modeDesired = self.Shares.get_GuiMode()
        self.rx_modeActual  = rx['MODE']
        self.rx_calibrated  = rx['CALIBRATED']
        self.rx_homeRx      = rx['HOMERX']
        self.rx_count1      = rx['L1']
        self.rx_count2      = rx['L2']
        self.rx_count3      = rx['L3']
        self.rx_newTarget   = self.Shares.get_GuiNewTarget()

    def update_outputs (self, WAYPOINT, X, Y, Z):
        if WAYPOINT:
            self.Shares.set_GuiWaypoint(WAYPOINT[0][0], WAYPOINT[0][1],
                                        WAYPOINT[0][2])
        else:
            self.Shares.set_GuiWaypoint(-1, -1, -1)
        self.Shares.set_GuiCurrentDirection(X, Y, Z)
        self.Shares.set_GuiStatus(_rTelescopeThreadState[self.state])
        self.Shares.set_GuiLengthCount(self.rx_count1, self.rx_count2,
                                       self.rx_count3)
        L1 = round(self.telescope.calcEnc2Inch(self.rx_count1),4)
        L2 = round(self.telescope.calcEnc2Inch(self.rx_count2),4)
        L3 = round(self.telescope.calcEnc2Inch(self.rx_count3),4)
        self.Shares.set_GuiLengthInch(L1, L2, L3)
        pass

    def close_thread (self):
        """
        Flag the thread to cease operations.
        """
        self.kill_thread = True

def _Example():
    """
    Example operation of thread.
    """

    def setRx(SHARES, OTA, X, Y, Z):
        lengths = OTA.calcActLength(X, Y, Z)
        C1 = OTA.calcInch2Enc(lengths['L1'])
        C2 = OTA.calcInch2Enc(lengths['L2'])
```

```
        C3 = OTA.calcInch2Enc(lengths['L3'])
        SHARES.set_RxValues(
            shares.TelescopeMode['Run'], True, True, C1, C2, C3)

    print 'Starting'
    Shares = shares.Shares()

    OTA = telescope_class.telescope('telescope.csv')
    Shares.set_RxValues(shares.TelescopeMode['Off'], False, False, 0, 0, 0)
    Shares.set_GuiOnOff(False)
    Shares.set_GuiMode(shares.TelescopeMode['Hold'])

    time.sleep(0.5)

    telescopeThread = Telescope_Thread (0.1, Shares)
    telescopeThread.daemon = False
    telescopeThread.start ()

    time.sleep(0.5)

    print 1, _rTelescopeThreadState[ telescopeThread.state ]

    Shares.set_GuiOnOff(True)
    time.sleep(0.5)
    print 2, _rTelescopeThreadState[ telescopeThread.state ]

    Shares.set_GuiMode(shares.TelescopeMode['Run'])
    time.sleep(0.5)
    print 3, _rTelescopeThreadState[ telescopeThread.state ]

    Shares.set_GuiMode(shares.TelescopeMode['Calibrate'])
    time.sleep(0.5)
    print 4, _rTelescopeThreadState[ telescopeThread.state ]

    hs = OTA.getHardstopCounts()
    Shares.set_RxValues(
        shares.TelescopeMode['Calibrate'], True, True, hs[0], hs[1], hs[2])
    time.sleep(0.5)
    print 5, _rTelescopeThreadState[ telescopeThread.state ]

    Shares.set_GuiMode(shares.TelescopeMode['Run'])
    time.sleep(0.5)
    print 6, _rTelescopeThreadState[ telescopeThread.state ],

    setRx(Shares, OTA, -11.805, 1.664, 0)
    time.sleep(0.5)
    print 7, _rTelescopeThreadState[ telescopeThread.state ]

    time.sleep(1)

    print 'Ending?'
    logging.debug('kill it')
    telescopeThread.close_thread()
```

```
    telescopeThread.join()
    time.sleep(3)
    print 'Fin'

if __name__ == '__main__':
    _Example()
```

serial_thread.py

```
import time
import threading
import logging
import serial
import shares

SOC = '['
EOC = ']'

RxParameterError = ['BadChar', 'Overflow', 'BufferNotComplete', 'NoParameter',
                    'ExtraParameter']
RxParameterError_list = list(enumerate(RxParameterError, -99))
RxParameterError_dict = {}
for number, name in RxParameterError_list:
    RxParameterError_dict[name] = number

class Serial_Thread (threading.Thread):
    """
    Thread responsible for communications between the host computer and Mbed
    motor controller via USB.
    """
    def __init__ (self, run_interval, Shares):
        """
        run_interval: Thread call rate in seconds.
        Shares:       Threadlocked data variables shared between threads.
        """
        threading.Thread.__init__ (self, name="SerialThread")
        self.event = threading.Event()
        self.kill_thread = False
        self.Shares = Shares
        self.rxBuffer = ''
        self.rxBuffering = False

        # Create pySerial instance for communication with mBed
        self.mBed = serial.Serial(port='COM8', baudrate=9600)

        # Save the starting value of the time interval between data points
        self.run_interval = run_interval

        # Record the starting time, and set it as the most recent and
        # next run times
        self.start_time = time.time ()
        self.next_run_time = self.start_time
        self.last_run_time = self.start_time

    def run (self):
        """
        Thread loop.
        """
        # Initialization
        self.Shares.set_GuiSendHome(True)
        time.sleep(1)
        hs = self.Shares.get_TxHardstopValues()
```

```
        # Begin Infinite Loop
        while (not self.kill_thread):
            # Check if it's time to take data yet;
            # if so, do it and update the timer
            now_time = time.time ()
            if (now_time >= self.next_run_time):
                self.next_run_time += self.run_interval
                # Send Home sequence if necessary
                if self.Shares.get_GuiSendHome():
                    output = ( '[1,' + str(hs[0]) + ',' + str(hs[1]) + ',' +
                               str(hs[2]) + ']' )
                    self.mBed.write(output)
                    self.Shares.set_GuiSendHome(False)
                # Tx as necessary
                self.TX_Command()
                time.sleep(0.1)
                # Rx as necessary
                self.RX_Command()
            # Sleep until the next time this task is supposed to runn
            time.sleep (self.run_interval / 10.0)
        # Remember to turn off motors, this means code is exiting
        self.mBed.close()

    def TX_Command (self):
        """
        TX Commands:
        0 - Set positions
        1 - Set home
        2 - Set mode
        3 - Request data
        """
        mode, act1, act2, act3 = self.Shares.get_TxValues()
        # Set Mode
        output = '[2,' + str(mode) + ']'
        self.mBed.write(output)
        # Goto Positions
        output = '[0,' + str(act1) + ',' + str(act2) + ',' + str(act3) + ']'
        self.mBed.write(output)
        # Request data
        self.mBed.write('[3]')

    def RX_Command (self):
        """
        Check serial line and record all values to buffer.  If EOC character is
        read, empty the buffer and interpret the contents.
        """
        out = ''
        while self.mBed.inWaiting() > 0:
            char = self.mBed.read(1)
            if char==SOC:
                self.rxBuffering = True
                self.rxBuffer    = ''
```

```
            elif self.rxBuffering and not char==EOC:
                self.rxBuffer += char
            elif char==EOC:
                self.rxBuffering = False
                self.RX_Parse(self.rxBuffer)
            out += char

    def RX_Parse (self, MESSAGE):
        """
        Interpret incoming serial buffer.
        """
        print 'Parse Message: ' + MESSAGE
        RxParameter_list = []
        for i in range(6): #Mode,Cal_Flag,Home_Flag,ACT1_ENC,ACT2_ENC,ACT3_ENC
            RxParameter_list.append(RxParameterError_dict['NoParameter'])

        for ii in range(len(RxParameter_list)):
            # First determine if any message left to parse
            if MESSAGE == '':
                break
            # Next, determine if more potential parameters or last parameter
            i = MESSAGE.find(',')
            #    More
            if i != -1:
                try:
                    RxParameter_list[ii] = int(MESSAGE[0:i])
                except:
                    RxParameter_list[ii] = RxParameterError_dict['BadChar']
                MESSAGE = MESSAGE[i+1:]
            #    Last
            else:
                try:
                    RxParameter_list[ii] = int(MESSAGE)
                except:
                    RxParameter_list[ii] = RxParameterError_dict['BadChar']
                MESSAGE = ''

        MODE      = RxParameter_list[0]
        CAL_FLAG  = RxParameter_list[1]
        HOME_FLAG = RxParameter_list[2]
        ACT1_ENC  = RxParameter_list[3]
        ACT2_ENC  = RxParameter_list[4]
        ACT3_ENC  = RxParameter_list[5]

        self.Shares.set_RxValues (MODE, CAL_FLAG, HOME_FLAG,
                                  ACT1_ENC, ACT2_ENC, ACT3_ENC)

    def close_thread (self):
        """
        Flag the thread to cease operations.
        """
        self.kill_thread = True
```

```
def _Example():
    """
    Example operation of thread.
    """
    logging.basicConfig(level=logging.DEBUG,
                        format='(%(threadName)-10s) %(message)s')
    print 'Starting'
    print RxParameterError_dict
    Shares = shares.Shares()
    serialThread = Serial_Thread (0.5, Shares)
    serialThread.daemon = False
    serialThread.start ()

    waitTime = 1;

    time.sleep(waitTime)
    Shares.set_TxValues(1, 20, 40, 60)
    time.sleep(waitTime)
    print Shares.get_RxValues()
    Shares.set_TxValues(2, 40, 80, 120)
    time.sleep(waitTime)
    print Shares.get_RxValues()
    Shares.set_TxValues(3, 80, 160, 240)
    time.sleep(waitTime)
    print Shares.get_RxValues()
    time.sleep(waitTime)
    print 'Ending?'


    logging.debug('kill it')
    serialThread.close_thread()
    serialThread.join()


if __name__ == '__main__':
    _Example()
```

B.1.4 *Example Telescope Configuration File*

Contents of used *telescope.csv* file. Note that point of reference can be anywhere, as all required points will be zeroed during initialization. Values are according to the telescope position when in zeroed orientation as described in Section 3.2.4.

Table B.1: *Example telescope.csv file, used for project prototype.*

Units in Inches	**X**	**Y**	**Z**
BASE	5.5	0	0
ACT1_GND	2.7	12	16
ACT1_0	12	10.3737	14.0938
ACT2_GND	2.7	-12	16
ACT2_0	12	-10.3737	14.0938
ACT3_GND	2.7	12	5
ACT3_0	1.6263	0	4.0938
	L1	**L2**	**L3**
Hardstop	10.75	10.75	10.75

B.2 Mbed C++ Scripts

Programming of the Mbed LPC1768 was written C++ in Mbed's online compiler. The Triple Motor/Encoder Driver operates as a simple PID controller for each of three motor assemblies with serial communication to the host computer over USB. Code has been uploaded to an online repository located at:

https://developer.mbed.org/users/gdgudgel/code/TripleMotorEncoderDriver/